CS 3100 – Models of Computation – Fall 2011 – Notes for L27
Halting Problem
Last Updated 10:01am, 12/01/11

Acceptance (A_{TM}) is undecidable (important!): We now prove this set to be *undecidable.* **Suppose** there exists a decider H for A_{TM}. H expects to be given a Turing machine M and a string w. Notice that "giving H a Turing machine" means "giving it a character string representing a Turing machine program."

Build a program called D as follows:

- D takes a single argument M.
- As its first step, D invokes H on $\langle M, M\rangle$.[1]
- If $H(\langle M, M\rangle)$ rejects, $D(\langle M\rangle)$ accepts.
- If $H(\langle M, M\rangle)$ accepts, $D(\langle M\rangle)$ rejects.

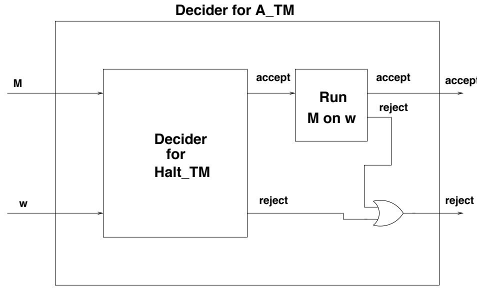
Now we can ask what $D(\langle D\rangle)$ will result in (to preserve the clarity of our arguments, the reader is invited to suppress any occurrence of $\langle \ldots \rangle$ in the text below):

- The $D(\langle D\rangle)$ "call" turns into an $H(\langle D, D\rangle)$ call.
- Suppose $H(\langle D, D\rangle)$ rejects. In that case, $D(\langle D\rangle)$ accepts.
- But, according to the advertised behavior of H — which is that it is a decider for A_{TM} — the fact that $H(\langle D, D\rangle)$ rejects means that D is *not* a Turing machine that will accept $\langle D\rangle$, or that $D(\langle D\rangle)$ rejects!
- Suppose $H(\langle D, D\rangle)$ accepts. In that case, $D(\langle D\rangle)$ rejects.
- But, according to the advertised behavior of H — which is that it is a decider for A_{TM} — the fact that $H(\langle D, D\rangle)$ accepts means that D *is* a Turing machine that accepts $\langle D\rangle$, or that $D(\langle D\rangle)$ accepts!

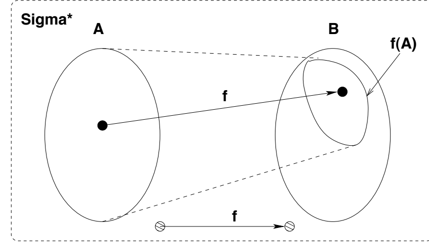
Therefore, we obtain a contradiction *in both cases.* Hence, the claimed decider H for A_{TM} cannot exist, or *the acceptance problem for Turing machines is undecidable.*

What does this have to do with diagonalization? We can conceptually build a table of machines that accept other machine descriptions (supposedly filled by the checker program `H`. Since `D` is a legal machine, it must exist in this table. Now, at the spot marked with `?`, `H` will run into a contradiction.

	<M1>	<M2>	<M3>	<M4>	..	<D>
M1	Y	N	N	Y		
M2						
M3						
M4						
..						
D						?



(a) A_{TM} to $Halt_{TM}$ reduction



(b) Illustration of mapping reduction from A to B

Halting ($Halt_{TM}$) is undecidable (important!) The golden rule of reduction is: **Reduce an existing ("old") undecidable problem to the given ("new") problem.** If we assume that the new problem is decidable, we would conclude that the existing undecidable problem is decidable (a contradiction, Figure 1(a) where the "*new problem*" is $Halt_{TM}$). By assuming that it is decidable, we can assume the existence of the decider $D_{Halt_{TM}}$, and using it, build a decider for A_{TM}, the "*old problem*," known undecidable; this obtains a contradiction. Hence, $D_{Halt_{TM}}$ cannot exist.

We should not go by the English language meaning of the term "reduction" that can lead us astray. To see why, assume this wrong direction and see what it leads to. It will lead us to prove, by contradiction, the following futile fact: "IF the existing undecidable problem is decidable, THEN we would have shown the new problem to be decidable." However, this is a statement of the form "IF false, THEN assert X." Clearly, in this case, we cannot assert X. *So we must always reduce the known undecidable problem to a new problem.* Then the new problem, if shown decidable, will trigger a contradiction. Always draw a "boxes within boxes" diagram as in Figure 1(a). Here, the new problem is used to build a box to "solve" the old impossible-to-solve problem.

An introduction to mapping reduction: Define $Halt_{TM} = \{\langle M, w\rangle \mid TM\ M\ halts\ on\ w\}$. We show $Halt_{TM}$ to be undecidable as follows:

> Suppose not; i.e., there is a decider for $Halt_{TM}$. Let's now build a decider for A_{TM} (call it $D_{A_{TM}}$). $D_{A_{TM}}$'s design will be as follows:
>
> - $D_{A_{TM}}$ will first feed M and w to $D_{Halt_{TM}}$, the claimed decider for $Halt_{TM}$.
> - If $D_{Halt_{TM}}$ goes to $accept_{D_{Halt_{TM}}}$, $D_{A_{TM}}$ knows that it can safely run M on w, which it does.
> - If M goes to $accept_M$, $D_{A_{TM}}$ will go to $accept_{D_{A_{TM}}}$.
> - If M goes to $reject_M$, or if $D_{Halt_{TM}}$ goes to $reject_{D_{Halt_{TM}}}$, $D_{A_{TM}}$ will go to $reject_{D_{A_{TM}}}$.

Notice that we have labeled the accept and reject states of the two machines $D_{Halt_{TM}}$ and $D_{A_{TM}}$. After one becomes familiar with these kinds of proofs, higher-level proof sketches are preferred. *Here is such a higher-level proof sketch. By way of contradiction, assuming $D_{Halt_{TM}}$ exists, build $D_{A_{TM}}$. Accepts input $\langle M, w\rangle$ and run $D_{Halt_{TM}}$ on it. If this run accepts, then we can safely (without the fear of looping) deploy M on w, and return the accept/reject result that this run returns; else return "reject."*

[1]Basically, we feed $\langle M\rangle$ twice over, just to 'please' H that expects two arguments.

Mapping reductions: **Definition:** A *computable* function $f : \Sigma^* \to \Sigma^*$ is a mapping reduction from $A \subseteq \Sigma^*$ into $B \subseteq \Sigma^*$ if for all $x \in \Sigma^*$, $x \in A \Leftrightarrow f(x) \in B$.

See Figure 1(b) which illustrates the general situation that A maps into a subset denoted by $f(A)$ of B, and members of A map into $f(A)$ while non-members of A map outside of B (that means they map outside of even $B \setminus f(A)$). Also note that A and B need not be disjoint sets, although they often are. A mapping reduction can be (and usually is) a non-injection and non-surjection; *i.e.*, it can be many to one and not necessarily onto. It is denoted by $\leq_m$. By asserting $A \leq_m B$, the existence of an f as described above is also being asserted. Typically mapping reductions are used as follows:

- Let A be a language known to be undecidable ("old" or "existing" language).
- Let B be the language that must be shown to be undecidable ("new" language).
- Find a mapping reduction f from A into B.
- Now, if B has a decider D_B, then we can decide membership in A as follows:
  - On input z, in order to check if $z \in A$, find out if $D_B(f(z))$ accepts or not. If it accepts, then $z \in A$, and if it rejects, then $z \notin A$.
- The PCP is shown undecidable by exhibiting a mapping reduction from A_{TM} to PCP instances ("tiles").
- Grammar ambiguity is shown undecidable through a mapping reduction from PCP instances to CFGs.
- We can see how arguments chain: A_{TM} is reduced to PCP which in turn is reduced to $CFG_ambiguity$. Thus $CFG_ambiguity$ being decidable makes PCP decidable which makes A_{TM} decidable.

Rice's Theorem

Every non-trivial partitioning of TM codes based on the language of the TMs is undecidable. A trivial partitioning is either "all TM codes" or "no TM code." These are decidable sets. A non-trivial partitioning attempts to classify $\langle M \rangle$ based on the *language* that the TM accepts. Here are some non-trivial partitionings (any such language is undecidable):

- The set of all TMs (TM codes) that accept a particular string
- The set of all TMs whose language is regular
- The set of all TMs which accept ϵ
- The set of all TMs which accept a given string w

A brief overview of NP-completeness

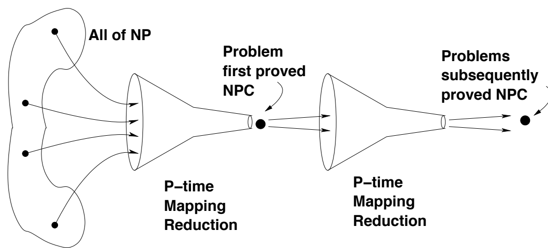
Class NP: The class NP is those problems for which there are non-deterministic algorithms that "perform" in polynomial time. That is, there is an NDTM that has one execution path (among the potential exponentially branching set of non-deterministic selections) that takes a polynomial number of steps.

Alternately, an NP problem is one for which given a "certificate" (a claimed answer) which is also polynomially long, a DTM can check the answer for correctness in polynomial time.

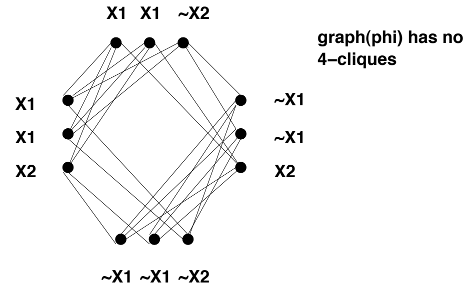
Examples of ND languages:

- $SAT = \{\Gamma \mid \Gamma \; is\ a\ Boolean\ formula\ that\ is\ satisfiable\}$. This is in NP because of either reason below:
  - One can develop a SAT checker that picks one assignment for the Boolean variables (the "guess generation phase") and checks the formula for satisfiability.
  - Given a formula and a claimed assignment, one can check whether the formula is true under the assignment.
- $3SAT = \{\Gamma \mid \Gamma \; is\ a\ 3CNF\ Boolean\ formula\ that\ is\ satisfiable\}$. This is in NP for the same reasons.

- *Clique* = $\{G \mid G\ is\ a\ graph\ with\ a\ \ clique\ of\ size\ k\}$. This is in NP because of either reason below:
  - The NDTM can generate a collection of k nodes and then check.
  - One can present a collection of k nodes and design a DTM that checks whether these form a clique.



(c) Show problem in NP; then do a mapping reduction.



(d) The Proof that *Clique* is NPH using an example formula $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (x_1 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_2)$

NP-Completeness: We don't know whether all ND languages have deterministic polynomial time algorithms. That is whether the language family NP and P are the same. To attack this annoying "gap" between P and NP, scientists have come up with the *hardest* members of NP called NP-complete.

A language L is NP-complete if

- (In NP) L is in NP
- (NP-hard) Every other language in NP can be poly-time mapping reduced to L

Note that there are undecidable NP-hard languages (e.g. Diophantines language!). Thus we must show the "In NP" part for any meaningful dialog in this space!!

The first problem shown to be NP-complete is 3SAT. This is done by showing that the "work of any NDTM can be represented as Boolean formula satisfaction."

Now for any other language (e.g. Clique), we show it NP-complete by

- Showing that Clique is in NP
- Showing that there is a mapping reduction from 3SAT to Clique

A diagram showing this mapping reduction chain is in Figure 1(c). Here, all of NPC was mapping-reduced to 3SAT. Then problems such as Clique are shown NPC by the second mapping-reduction whose details are illustrated in Figure 1(d). If Clique were to have a P-time algorithm, then *all of NP* will have a P-time algorithm—thought to be highly unlikely!

Significance of NPC

While NP-complete problems are solved using algorithms with exponential time complexity, with well-chosen heuristics, they often run in polynomial time. Complexity classifications are essential for theoretical CS advance by linking problems together into complexity classes, and developing a common understanding.