Last time we agreed that algorithmic solvability of a problem is equivalent to the solvability by a Turing Machine – this is the Church-Turing Thesis. Just to remind:

**Definition:** A language $L$ is called *Turing-decidable* (or just *decidable*), if there exists a Turing Machine $M$ such that on input $x$, $M$ accepts if $x \in L$, and $M$ rejects otherwise. $L$ is called *undecidable* if it is not decidable.

In light of the previous lecture, and in light of the fact that languages represent decision problems, we have

- Decidable languages correspond to algorithmically solvable decision problems.
- Undecidable languages correspond to algorithmically unsolvable decision problems.

For the rest of this course we will look at some of the decidable and some of the undecidable languages/problems.

**Decidable Languages**

First, some closure properties:

**Theorem:** The class of decidable languages is closed under complement.

**Proof:** Let $M$ be a TM that decides a language $L$. Construct a TM $M'$ that decides $\overline{L}$ in the following way:

$M' = \text{"On input } w:\"

1. Simulate $M$ on $w$.
2. *Accept* if $M$ rejects, *reject* if $M$ accepts”.

**Important:** for decidability we *always* have to check two things – $M'$ always halts, and $M'$ gives the correct result on every input.

$M'$ always halts because $M$ always halts. $M'$ gives the correct result, because if $w \in L$, $M$ will accept, and so $M'$ will reject, and if $w \notin L$, $M$ will reject, in which case $M'$ will accept.

Theorem:** The class of decidable languages is closed under union.

**Proof:** Let $M_1$ be a TM that decides a language $L_1$, and $M_2$ be a TM that decides $L_2$. Construct a TM $M$ that decides $L_1 \cup L_2$ in the following way:

$M = \text{"On input } w:\"

1. Simulate $M_1$ on $w$.
2. If $M_1$ accepts, accept.
2. If $M_1$ rejects, simulate $M_2$ on $w$.

3. Accept if $M_2$ accepts, reject if $M_2$ rejects.

$M$ always halts because $M_1$ and $M_2$ always halt. If $w \in L_1 \cup L_2$, then either $w \in L_1$ or $w \in L_2$. If $w \in L_1$, then $M_1$ accepts $w$, and so $M$ accepts $w$ in line 2. Otherwise $M_2$ accepts $w$, and so $M$ accepts $w$ in line 4. If $w \notin L_1 \cup L_2$, then both $M_1$ and $M_2$ reject $w$, and so $M$ rejects $w$.

**Theorem:** The class of decidable languages is closed under intersection.

**Proof:** $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$.

Now, let us look at some “interesting” solvable problems and the decidable languages that correspond to them.

- **Acceptance problem for DFAs:** given a DFA $D$ and the input string $w$, determine if $D$ accepts $w$.

  Is this problem solvable? Sure. Now, let us turn this problem into a language, and build a Turing Machine to decide it. We know that this machine exists – why?

  The language:

  $$A_{DFA} = \{ \langle D, w \rangle \mid D \text{ is a DFA that accepts } w \}.$$ 

  Explanation: $A_{DFA}$ is the language of all strings that represent a valid encoding of all pairs $D$ and $w$, where $D$ is an encoding of some DFA, $w$ is the input string, and $D$ accepts $w$.

  What is a “valid encoding” of an object $O$? Any string which contains enough information about $O$ so the Turing Machine can analyze some relevant properties of $O$. Notation: $\langle O \rangle$.

  Whenever a Turing Machine is given an input string that is supposed to be an encoding of some object, it is implied that the first step of a TM is to verify that the given string is indeed a correct encoding of the object. If not, TM immediately rejects.

  So, the machine to recognize $A_{TM}$:

  $M =$ “On input $\langle D, w \rangle$, where $D$ is a DFA and $w$ is a string

  1. Simulate $D$ on $w$.

  2. Accept, if the simulation ends in the accept state. Reject otherwise”.


Example of Encoding and Simulation:

**Theorem:** $A_{DFA}$ is decidable.

**Proof:** The Turing Machine $M$ constructed above decides $A_{DFA}$: $M$ halts on every input (if the input is not a valid encoding – reject right away; if it is then simulate a DFA, but a DFA always halts, so the simulation will halt too). $M$ accepts if and only if the simulation of $D$ on $w$ ends in the accepting state, i.e. a DFA encoded by $D$ accepts $w$.

Other decidable problems about FAs (see Chapter 4.1 in the textbook for detailed constructions):

- **Acceptance problem for NFAs:** given a NFA $N$ and the input string $w$, determine if $M$ accepts $w$.

  The language:

  $$A_{NFA} = \{ \langle N, w \rangle \mid N \text{ is a NFA that accepts } w \}.$$  

  Idea for the Turing Machine for $A_{NFA}$: convert $N$ into the representation of equivalent DFA $D$ (we know the algorithm to do it, so, by the Church-Turing Thesis, Turing Machine can do it too), and run the machine for $A_{DFA}$, on $\langle D, w \rangle$.

  Alternative: just simulate the NFA $N$ directly.

- **Emptiness problem for DFAs:** given a DFA $D$ determine if $D$ accepts any strings at all, i.e. if $L(D) = \emptyset$.

  The language:

  $$E_{DFA} = \{ \langle D \rangle \mid D \text{ is a DFA and } L(D) = \emptyset \}.$$
Idea for the Turing Machine for $E_{DFA}$: check if any of the accept states are reachable from the start state.

**Equivalence problem for DFAs:** given two DFAs $D_1$, and $D_2$, determine if the recognize the same language, i.e. if $L(D_1) = L(D_2)$.

The language:

$$EQ_{DFA} = \{\langle D_1, D_2 \rangle \mid D_1, D_2 \text{ are DFAs and } L(D_1) = L(D_2)\}.$$

Idea 1 for the Turing Machine for $EQ_{DFA}$: construct a DFA that accepts only those strings that are accepted by $D_1$ or by $D_2$, but not both. Test if the new DFA for emptiness (see the textbook).

Idea 2 for the Turing Machine for $EQ_{DFA}$: ...

- Some decidable problems about CFGs (see Chapter 4.1 in the textbook for detailed constructions).

**Acceptance problem for CFGs:** given a CFG $G$ and the input string $w$, determine if $G$ generates $w$.

The language:

$$A_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates } w\}.$$

Algorithm for the Turing Machine for $A_{CFG}$:

1. Find all non-terminals that generate $\varepsilon$. If $S$ is among those, and $w = \varepsilon$, accept.  
   **Exercise:** give an algorithm to find all non-terminals that can generate $\varepsilon$.

2. Otherwise, let $B$ be any of such terminals. Then, for every rule of the form $C \rightarrow uBv \ (uv \neq \varepsilon)$, add a new rule $C \rightarrow uv$. Then, remove all the rules of the form $A \rightarrow \varepsilon$.

3. Now we have grammar $G'$, such that $L(G') = L(G) - \varepsilon$. We need to check if $w$ can be generated by $G'$. To do this, do a depth-first search on all left-most derivations starting from the start symbol. Of course, there is an infinite number of these, but, because there is no $\varepsilon$-rules, on every step of the derivation the length of the current string (of terminals and non-terminal) is either growing or staying the same. Thus, we can stop searching a particular branch the moment it produces a string that is longer than $|w|$. Also, to cover the possibility of loops (for example, $S \Rightarrow A \Rightarrow S$) we need to remember the strings that we already saw.

Note: the textbook gives an algorithm for this problem which uses Chomski Normal Form for the grammar. The algorithm here does not require CNF.
Emptiness problem for CFGs: given a CFG $G$ determine if it generates any string, i.e. if $L(G) = \emptyset$.

The language:

$$E_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG, and } L(G) = \emptyset \}.$$ 

Idea for the Turing Machine for $E_{CFG}$: mark the terminal symbols. Repeat: for each rule that has all symbols in the right-hand side marked, mark its left-hand side. If $S$ is marked, reject, otherwise accept.

- **Exercises**: textbook 4.2, 4.3, 4.4, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.19.