Readings for this lecture

Chapters 7-8 of [Sipser 1996], 3rd edition.
Complexity measures

Definition (Time complexity of a Deterministic TM)
Let $M$ be a deterministic TM that halts on all inputs. The running time or time complexity of $M$ is a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that $f(n)$ is the maximum number of steps that $M$ uses on any input of length $n$.

For nondeterministic TMs we change the definition of $f(n)$ to be “the maximum number of steps that $N$ makes on any branch of its computation on any input of length $n.”

Definition (Space Complexity of a Deterministic TM)
Let $M$ be a deterministic Turing machine, DTM, that halts on all inputs. The space complexity of $M$ is the function $f : \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that $M$ scans on any input of length $n$.

For nondeterministic TMs we change the definition of $f(n)$ to be “the maximum number of tape cells that $M$ scans on any branch of its computation for any input of length $n.”
Asymptotic analysis

- Computing exact running time or used space of an algorithm is often a complex expression, we usually just estimate it.
- One convenient form of estimation is the so-called asymptotic analysis which determines cost w.r.t. large inputs.
- This is valid because the value of the highest order term dominates the value of the other terms on large inputs.

Consider the function \( f(n) = 6n^3 + 2n^2 + 10n + 100 \).

- Disregarding the coefficient 6, we say that \( f \) is asymptotically at most \( n^3 \).
- The asymptotic notation, or big-O notation, for describing this estimation is \( f(n) = \mathcal{O}(n^3) \).

- Big-O notation says that a function is asymptotically no more than another function.
- Small-O notation says that a function is asymptotically less than another function, e.g. \( \sqrt{n} = o(n) \).
How to show a language is in a given class?

- **P**: a polynomial time deterministic TM decider
- **NP**:  
  - a polynomial time nondeterministic TM decider, or  
  - a polynomial time deterministic verifier
- **NP-hard**: problems to which every $A \in \text{NP}$ problem can be polynomial time reduced

**Definition (NP-complete)**

A language $B$ is **NP-complete** if it is NP and NP-hard.

- **PSPACE**: A polynomial space deterministic TM decider
- **NPSPACE**: A polynomial space nondeterministic TM decider
- **PSPACE-hard**: problems to which every PSPACE problem can be polynomial space reduced

**Definition (PSPACE-complete)**

A language $B$ is **PSPACE-complete** if it is PSPACE and PSPACE-hard.
**Other classes**

$\mathbf{L}$ is the class of languages that are decidable in logarithmic space on a DTM with a read-only input tape, i.e.

$$ L = \text{SPACE}(\log n) $$

$\mathbf{NL}$ is the class of languages that are decidable in logarithmic space on a NTM with a read-only input tape, i.e.

$$ \mathbf{NL} = \text{NSPACE}(\log n) $$

In summary

$$ \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \text{PSPACE} = \text{NSPACE} \subseteq \text{EXPTIME} $$
P versus NP and NP-completeness

Is $P = NP$?

- This is one the greatest unsolved problems in theoretical computer science and contemporary mathematics.
- If $P$ is equal to $NP$ then any polynomially verifiable problem would be polynomially decidable.
- SAT is an NP-complete problem since it is NP and we can polynomially reduce any arbitrary NP problem into it.
- SAT was the first problem to be proven to be NP-complete.
- All problems in $NP$ are at most as difficult as SAT.
- Providing a polynomial algorithm to SAT solves the $P$ versus $NP$ dilemma.
## Closure properties

<table>
<thead>
<tr>
<th>Operation</th>
<th>Regular</th>
<th>Ctx-free</th>
<th>Det. Ctx-free</th>
<th>Decidable</th>
<th>Turing recog.</th>
<th>P</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>concatenation</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>star</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>intersection</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>intersect w/ regular</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>complement</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>reversal</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>shuffle</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
The \textit{PATH} problem

Does a directed path exists from two given nodes in a graph? Let

\[
\text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t \}\]

\textbf{Theorem}

\[\text{PATH} \in \mathbf{P}\]

\textbf{Proof idea}

Construct a polynomial time algorithm that decides \textit{PATH}. Note that the brute-force algorithm that examines all potential paths in \( G \) and determines whether any is a direct path from \( s \) to \( t \) would not suffice.
A polynomial time algorithm for \textit{PATH}

\[ M = \text{"On input string } \langle G, s, t \rangle \text{ where } G \text{ is a direct graph containing nodes } s \text{ and } t:\]

1. Place a mark on node \( s \)
2. Repeat until no additional nodes are marked:
   3. Scan all the edges of \( G \). If an edge \((a, b)\) is found going from a marked node \( a \) to an unmarked node \( b \), mark node \( b \)
4. If \( t \) is marked, \textit{accept}. Otherwise \textit{reject}."

\[ \Box \]

\( \text{Stage 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound}\)

\( \text{Stage 3 runs at most } m \text{ times because each time except the last it marks an additional node of } G. \text{ Each run scans all the edges, which can be done polynomially on the number of edges. Hence, stages 2 and 3 are also polynomially bound}\)

\( \text{Therefore } M \text{ is a polynomial time algorithm for } \textit{PATH}\)
A polynomial time algorithm for \textit{PATH}

\( M = \) “On input string \( \langle G, s, t \rangle \) where \( G \) is a direct graph containing nodes \( s \) and \( t \):

1. Place a mark on node \( s \)

2. Repeat until no additional nodes are marked:
   
   3. Scan all the edges of \( G \). If an edge \((a, b)\) is found going from a marked node \( a \) to an unmarked node \( b \), mark node \( b \)

4. If \( t \) is marked, \textit{accept}. Otherwise \textit{reject}.”

\( \triangleright \) Stages 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound
A polynomial time algorithm for PATH

$M =$ “On input string $\langle G, s, t \rangle$ where $G$ is a direct graph containing nodes $s$ and $t$:

1. Place a mark on node $s$
2. Repeat until no additional nodes are marked:
   3. Scan all the edges of $G$. If an edge $(a, b)$ is found going from a marked node $a$ to an unmarked node $b$, mark node $b$
4. If $t$ is marked, accept. Otherwise reject.”

▷ Stages 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound

▷ Stage 3 runs at most $m$ times because each time except the last it marks an additional node of $G$. Each run scans all the edges, which can be done polynomially on the number of edges. Hence, stages 2 and 3 are also polynomially bound
A polynomial time algorithm for **PATH**

\[
M = \text{"On input string } \langle G, s, t \rangle \text{ where } G \text{ is a direct graph containing nodes } s \text{ and } t:\n\begin{enumerate}
\item Place a mark on node } s
\item Repeat until no additional nodes are marked:
\begin{enumerate}
\item Scan all the edges of } G. \text{ If an edge } (a, b) \text{ is found going from a marked node } a \text{ to an unmarked node } b, \text{ mark node } b
\end{enumerate}
\item \text{If } t \text{ is marked, accept. Otherwise reject."
\end{enumerate}
\]

▷ Stages 1 and 4 are executed only once and at most scan all the nodes, hence they are polynomially bound

▷ Stage 3 runs at most } m \text{ times because each time except the last it marks an additional node of } G. \text{ Each run scans all the edges, which can be done polynomially on the number of edges. Hence, stages 2 and 3 are also polynomially bound

▷ Therefore } M \text{ is a polynomial time algorithm for } \textit{PATH}
The HAMPATH problem

Hamiltonian path problem:

- A Hamiltonian path in a directed graph $G$ is a path that goes through each node of $G$ exactly once

- Hamiltonian path problem consists of testing whether a directed graph $G$ contains a Hamiltonian path connecting two specified nodes

$HAMPATH = \{ \langle G, s, t \rangle \mid G$ is a directed graph with a Hamiltonian path from $s$ to $t \}$

It is easy to obtain an exponential algorithm to solve HAMPATH, however nobody knows whether a polynomial one exists.
Polynomial Verifiability of $HAMPATH$

The $HAMPATH$ problem has a feature called *polynomial verifiability* which is important for understanding its complexity:

> If a Hamiltonian path in a graph $G$ is discovered (no matter how) we could easily convince someone else of its existence, simply by presenting it!

This is, *verifying* the existence of a Hamiltonian graph may be much easier than *determining* its existence.
An NTM deciding \textbf{HAMPATH}

\[ N_1 = \text{"On input string } \langle G, s, t \rangle \text{ where } G \text{ is a directed graph containing nodes } s, t:} \]

1. Write a list of \( m \) numbers \( p_1, \ldots, p_m \) where \( m \) is the number of nodes in \( G \). Each \( p_i \) is nondeterministically selected between 1 and \( m \).
2. Check for repetitions in the list. If any are found, \textit{reject}
3. Check whether \( s = p_1 \) and \( t = p_m \). If either fails, \textit{reject}
4. For each \( i, 1 \leq i \leq m \), check whether \( p_1, p_{i+1} \) is an edge of \( G \). If any are not, \textit{reject}. Otherwise, \textit{accept}.”

Each stage clearly runs in nondeterministic polynomial time, thus so does \( N_1 \).
Examples on Space Complexity

▷ SAT can be solved with the linear space algorithm $M_1$, which means it is in PSPACE:

$M_1 = \text{“On input } ⟨\varphi⟩, \text{ where } \varphi \text{ is a Boolean formula:}\\
1. \text{For each truth assignment to the variables } x_1, \ldots, x_n \text{ of } \varphi:\n   (a) \text{Evaluate } \varphi \text{ on that truth assignment; }\\   (b) \text{If } \varphi \text{ evaluates to 1, accept }\\
2. \text{If } \varphi \text{ never evaluates to 1, reject.”}\\

▷ $\overline{\text{ALL}}_{\text{NFA}} = \{⟨A⟩ | A \text{ is an NFA and } \mathcal{L}(A) \neq \Sigma^∗\}$

$\overline{\text{ALL}}_{\text{NFA}} \in \text{NSPACE}(n)$:

Proof idea

Construct $N$, a nondeterministic linear space algorithm that decides $\overline{\text{ALL}}_{\text{NFA}}$ by guessing a string that is rejected by NFA $A$ and uses a linear space to keep track of which states the NFA could be in at a particular time. Note, this language is not know to NP or coNP.