MIT 6.840: Theory of Computation

Taught by Michael Sipser
Notes by Leonard Tang and Austin Li

Fall 2020

The course was taught by Michael Frederic Sipser. The lectures were on Tuesdays and Thursdays at 2:30–4pm EST. The course had biweekly problem sets, one midterm and one final. The course assistants were Fadi Atieh, Damian Barabonkov, Di-Chia Chueh, Alexander Dimitrakakis, Thomas Xiong, Abbas Zeitoun, and Emily Liu. Additional material can be found on the course website.

Contents

1 Regular Languages **3**
1.1 Key Definitions 3
1.2 Key Results 3
1.3 Proof Concepts and Examples 4
1.4 Problem Set Results 4

2 Context-Free Languages **5**
2.1 Key Definitions 5
2.2 Key Results 6
2.3 Proof Concepts and Examples 6
2.4 Problem Set Results 6

3 The Church-Turing Thesis **7**
3.1 Key Definitions 7
3.2 Key Results 7
3.3 Proof Concepts and Examples 8
3.4 Problem Set Results 8

4 Decidability **9**
4.1 Key Definitions 9
4.2 Key Results 9
4.3 Proof Concepts and Examples 10
4.4 Problem Set Results 10

5 Reducibility **11**
5.1 Key Definitions 11
5.2 Key Results 11
5.3 Proof Concepts and Examples 11
5.4 Problem Set Results 11

6 Advanced Topics in Computability Theory **12**
6.1 Key Definitions 12
6.2 Key Results 12

6.3 Proof Concepts and Examples . . . 12
6.4 Problem Set Results . . . 12

7 Time Complexity **13**
7.1 Key Definitions . . . 13
7.2 Key Results . . . 13
7.3 Proof Concepts and Examples . . . 15
7.4 Recitation Results . . . 17
7.5 Problem Set Results . . . 19

8 Space Complexity **22**
8.1 Key Definitions . . . 22
8.2 Key Results . . . 22
8.3 Recitation Results . . . 28
8.4 Proof Concepts and Examples . . . 29
8.5 Problem Set Results . . . 29

9 Intractability **31**
9.1 Key Definitions . . . 31
9.2 Key Results . . . 31
9.3 Proof Concepts and Examples . . . 33
9.4 Problem Set Results . . . 33

10 Advanced Topics in Complexity Theory **34**
10.1 Key Definitions . . . 34
10.2 Key Results . . . 34
10.3 Problem Set Results . . . 35

11 Class Relations, Closure Properties, and Useful Problems **36**

1 Regular Languages

1.1 Key Definitions

Definition 1.1. A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set called the **states**,
2. Σ is a finite set called the **alphabet**,
3. $\delta : Q \times \Sigma \to Q$ is the **transition function**,
4. q_0 is the **start state**, and
5. $F \subseteq Q$ is the **set of accept states**.

Definition 1.2. If A is the set of all strings a machine M accepts, we say A is the **language** of M and write $L(M) = A$. We say M **recognizes** A or M **accepts** A. The **empty language** is the language of no strings, denoted $\emptyset$.

Definition 1.3. A language is called a **regular language** if some finite automaton recognizes it.

Definition 1.4. For languages A, B, the regular operations are:

Union: $A \cup B : \{x : x \in A \text{ or } x \in B\}$
Concatenation: $A \circ B : \{xy : x \in A \text{ and } y \in B\}$
Star: $A^* = \{x_1 x_2 \ldots x_k : k \geq 0 \text{ and each } x_i \in A\}$.

Definition 1.5. A **nondeterministic finite automaton (NFA)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where all are the same as in the deterministic case except

$$\delta : Q \times \Sigma_\varepsilon \to P(Q), \qquad \Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$$

and $P(Q)$ the power set of Q.

Definition 1.6. Two machines M_1, M_2 are **equivalent** if they recognize the same language.

Definition 1.7. R is a **regular expression** if R is

1. a for some $a \in \Sigma$,
2. ε,
3. $\emptyset$,
4. $R_1 \cup R_2, R_1 \circ R_2$, or R_1^* for R_1, R_2 regular expressions.

Definition 1.8. A **generalized nondeterministic finite automaton (GNFA)** is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$ where all else same as DFA, NFA, transition function given by

$$\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to \mathcal{R}.$$

1.2 Key Results

Theorem 1.9. *Class of regular languages closed under union operation.*

Proof. Consider machines that recognize A_1, A_2 and construct M recognizing $A_1 \cup A_2$ with $Q = Q_1 \times Q_2, \Sigma = \Sigma_1 \cup \Sigma_2, \delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)), q_0 = (q_1, q_2), F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$, keeping track of **pairs** of states.

Faster: Take two NFAs that recognize A_1, A_2, construct N recognizing $A_1 \cup A_2$ by creating new start state and sending ε-transitions to start states of N_1, N_2. □

Theorem 1.10. *Every NFA has an equivalent DFA.*

Proof. Massage states and transition function of an NFA N into the states and transition function of DFA M using sets. □

Corollary 1.11. *A language is regular* $\iff$ *some NFA recognizes it.*

Theorem 1.12. *Class of regular languages closed under concatenation.*

Proof. Use nondeterminism to guess where to make split by connecting accepting states of N_1 recognizing A_1 to start state of N_2 recognizing A_2 with ε-transitions. □

Theorem 1.13. *The class of regular languages is closed under the star operation.*

Proof. From N_1 recognizing A_1, create new start state q_0, connect to old start state via ε-transition, and connect all accepting states to old start state via ε-transitions. □

Theorem 1.14. *A language is regular* $\iff$ *some regular expression describes it.*

Proof. ($\impliedby$) Convert R into NFA N. ($\implies$) Convert DFA into GNFA into regular expression. The conversions are done by ripping out intermediate state and repairing all connections. □

Theorem 1.15 (Pumping lemma)**.** *If A a regular language, exists p (pumping length) where if $s \in A, |s| \geq p$, s can be divided into three pieces $s = xyz$:*

1. $\forall i \geq 0, xy^iz \in A$
2. $|y| > 0$,
3. $|xy| \leq p$.

1.3 Proof Concepts and Examples

Example 1.16. Creating DFAs, NFAs to show languages regular, as if you are machine.

Example 1.17. Use ε-transitions to prove closure properties and build NFAs.

Example 1.18. Use **pumping lemma** to prove langauge nonregular:

Let $B = \{0^n1^n : n \geq 0\}$. WTS B nonregular. Consider string $0^p1^p \in B$. Use pumping lemma, $s = xyz$. Three cases, y contains only 0s or 1s. After pumped, there will be unequal amount. If y has both $0, 1$, after pumping, will be out of order, so a contradiction $\implies$ B nonregular.

Example 1.19. Know how to convert from DFA to NFA to RegEx

Example 1.20. Two DFAs A with a states and B with b states differ on some input iff they differ on some input of length at most ab.

Example 1.21. Two NFAs A with a states and B with b states differ on some input iff they differ on some input of length at most 2^{a+b}

1.4 Problem Set Results

Problem 1.22. *Class of regular languages closed under complement.*

Proof. Swap accept and nonaccept states of a DFA M. □

Problem 1.23. *Class of regular languages closed under reversal. For any language A, $A^R = \{w^R : w \in A\}$. A regular $\implies$ A^R regular.*

Problem 1.24. *Class of* ***nonregular*** *languages is*

1. ***Not*** *closed under union*
2. ***Not*** *closed under concatenation*
3. ***Closed*** *under complementation.*

2 Context-Free Languages

2.1 Key Definitions

Definition 2.1. A **context-free grammar (CFG)** is a 4-tuple (V, Σ, R, S) where

1. V is a finite set called the **variables**,
2. Σ is a finite set, jisjoint from V called the **terminals**,
3. R is a finite set of **rules**, each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

Consider G_1 given by

$$
\begin{aligned}
S &\to 0S1 \mid B, \\
B &\to \#
\end{aligned}
$$

Here, S is the start variable, B is a variable, $0, 1, \#$ are terminals. A sequence of substitutions to obtain a string is a **derivation** and can be reprented pictorially with a **parse tree**.

Definition 2.2. Any language that can be generated by some context-free grammar is called a **context-free language (CFL)**.

Definition 2.3. A string w is derived ambiguously in a CFG G if it has two or more leftmost derivations. Grammar G is **ambiguous** if it generates some string ambiguously.

Definition 2.4. A context-free grammar is in **Chomsky normal form** if every rule is of the form

$$
\begin{aligned}
A &\to BC \\
A &\to a
\end{aligned}
$$

where a is any terminal and A, B, C are any variables, with B, C not the start variable. We permit the rule $S \to \varepsilon$ where S the start variable.

Definition 2.5. A **pushdown automaton (PDA)** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, where $Q, q_0, F \subseteq Q$ are the same as always with

1. Σ is the input alphabet,
2. Γ is the stack alphabet,
3. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function.

PDAs are like NFAs with an extra component called a **stack**, that provides additional memory beyond finite control. A PDA can write on and read symbols on the stack. Writing is called **pushing** and removing a symbol is called **popping**.

Definition 2.6. A **deterministic pushdown automaton (DPDA)** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, Γ, F all finite sets with

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to (Q \cup \Gamma_\varepsilon) \cup \{\emptyset\}$$

is the transition function satisfying: $\forall q \in Q, a \in \Sigma, x \in \Gamma$, exactly one of the values

$$\delta(q, a, x), \qquad \delta(q, a, \varepsilon), \qquad \delta(q, \varepsilon, x), \qquad \delta(q, \varepsilon, \varepsilon)$$

is **not** $\emptyset$. This conforms to the principle of determinism: at each step of computation, DPDA has at most one way to proceed according to transition function. The language of a DPDA is called a **deterministic context-free language (DCFL)**.

2.2 Key Results

Theorem 2.7. *Any context-free language is generated by a context-free grammar in Chomsky normal form.*

Proof. Convert any grammar G into Chomsky normal form. Add new start variable $S_0 \to S$, eliminate all ε-rules of form $A \to \varepsilon$ and eliminate all unit rules of the form $A \to B$ and patch up grammar to be sure that it generates the same language. □

Theorem 2.8. *A language is context-free $\iff$ some PDA recognizes it.*

Proof. ($\Rightarrow$) Convert CFG G into PDA P by nondeterministically selecting one of the rules for A and substituting A by the string on RHS of the rule. If matches input, pop the part of string that matches and continue. ($\Leftarrow$) Construct PDA P from CFG G. □

Theorem 2.9. *If a PDA recognizes some language, then it is context-free.*

Corollary 2.10. *Every regular language is context free.*

Theorem 2.11 (Pumping lemma for context-free languages). *If A a CFL $\Rightarrow$ $\exists p$ (pumping length) where if $s \in A : |s| \geq p$, s can be divided into five pieces $s = uvxyz$ satisfying conditions*

1. *$\forall i \geq 0, uv^i xy^i z \in A$,*
2. *$|vy| > 0$, and*
3. *$|vxy| \leq p$.*

Theorem 2.12. *Class of DCFLs is closed under complementation.*

2.3 Proof Concepts and Examples

Example 2.13. Use stack as additional memory and check for matches on input tape.

Example 2.14. Use pumping lemma to show language not context free.

Let $B = \{a^n b^n c^n : n \geq 0\}$. WTS B not context free.

2.4 Problem Set Results

Problem 2.15. *CFLs are **closed under** union, concatenation, and star.*

Problem 2.16. *CFL $\cap$ regular = CFL.*

Problem 2.17. *If G a CFG in Chomsky normal form, then for any string $w \in L(G)$ of length $n \geq 1$, exactly $2n - 1$ steps required for any derivation of w.*

3 The Church-Turing Thesis

3.1 Key Definitions

Definition 3.1. A **Turing machine (TM)** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are all finite sets and

1. Σ is the input alphabet not containing the **blank symbol** $\sqcup$,
2. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
3. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
4. $q_0 \in Q$ is the start state,
5. $q_{accept} \in Q$ is the accept state, and
6. $q_{reject} \in Q$ is the reject state, $q_{reject} \neq q_{accept}$.

The transition function has $\{L, R\}$, meaning after reading state symbol and writing a symbol, it moves either left or right. As a Turing machine, computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the Turing machine.

A Turing machine on an input may *accept*, *reject*, or *loop*. To **loop** means that the machine does not halt. Will use high-level descriptions to describe TMs.

Definition 3.2. The collection of strings that M accepts is the **language of** M, or the **language recognized by** M, denoted $L(M)$.

Definition 3.3. A language **Turing-recognizable** if some Turing machine recognizes it.

Definition 3.4. A language **Turing-decidable** or simply **decidable** if some Turing machine decides it. Deciders always make a decision to accept or reject, never halt.

Every decidable language is Turing-recognizable

Definition 3.5. A **multitape Turing machine** is an ordinary TM with several tapes. Each tape has its own head for reading and writing.

Definition 3.6. An **enumerator** is a Turing machine with an attached printer. The language enumerated by E is the collection of all the strings that it eventually prints out. E can generate the strings of the language in any order, possibly with repetitions.

3.2 Key Results

Theorem 3.7. *Every multitape TM has an equivalent single-tape TM.*

Proof. Convert multitape TM M into an equivalent single-tape TM S. $\forall a \in \Sigma$, add $\dot{a}$ to Σ to mark head positions of different tapes and separate different tape inputs by #. Simulate the M on S by writing all contents on tapes of M onto single-tape S and do what M does. □

Corollary 3.8. *A language is Turing-recognizable* $\iff$ *some multitape TM recognizes it.*

Theorem 3.9. *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

Corollary 3.10. *A language is Turing-recognizable* $\iff$ *some nondeterministic TM recognizes it.*

Corollary 3.11. *A language is decidable* $\iff$ *some nondeterminsitic TM decides it.*

Theorem 3.12. *A language is Turing-recognizable* $\iff$ *some enumerator enumerates it.*

Proof. Show if M enumerates A, a TM M recognizes A. Create M such that it accepts all strings E prints. Create E such that it prints all strings that M accepts. □

Theorem 3.13 (**Church-Turing thesis**). *Intuitive notion of algorithms = Turing machine algorithms.*

3.3 Proof Concepts and Examples

Example 3.14. Using high level descriptions for TM deciders and recognizers:

Let $A = \{\langle G \rangle : G$ is a connected undirected graph$\}$. Following high-level description of TM M that decides A.

$M =$ "on input $\langle G \rangle$, the encoding of a graph G:

1. Select first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
   For each node in G, mark if it is attached by an edge to a node that is already marked.
3. Scan all nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*."

Example 3.15. Adding symbols to stack/tape alphabet to manipulate PDAs/to show equivalence.

Example 3.16. To show TM equivalence, need to show that operations can be simulated in both directions.

3.4 Problem Set Results

Problem 3.17. *A* ***deterministic queue automaton (DQA)*** *is like a push-down automaton with stack replaced by a queue. A* ***queue*** *is a tape allowing symbols to be written only on the left-hand side and read on the right-hand side. Each write operation (**push**) adds symbol to the left-hand end of the queue and each read operation (**pull**) reads and removes symbol on right-hand end. The input tape contains a cell with blank symbol to denote end of input.*

A language can be recognized by a DQA $\iff$ language is Turing-recognizable.

Problem 3.18. *A language is decidable $\iff$ some enumerator enumerates the language in* ***string order****. String order is the standard length-increasing, lexicographic order.*

Problem 3.19. $PUSHER = \{\langle P \rangle : P$ *is a PDA that pushes a symbol on its stack on some branch of computation at some point on input* $w \in \Sigma^*\}$ *is decidable.*

4 Decidability

4.1 Key Definitions

4.2 Key Results

Theorem 4.1. *$E_{DFA} = \{\langle B \rangle | B$ is a DFA and $L(B) = \emptyset\}$ is decidable.*

Proof. Basically do BFS marking starting from start state. If accept state marked *accept*, otherwise *reject*. □

Theorem 4.2. *EQ_{DFA}, equivalence problem for DFAs, is decidable.*

Proof. Construct language $(L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)})$. Regular languages closed under union and complement and intersection, so some DFA C decides this language. Now, emptiness problem for DFAs decidable, so run that TM on C. If empty, then *accept* (no problems in one and not other); otherwise *reject*. □

Theorem 4.3. *A_{CFG} (acceptance problem for CFGs) is decidable.*

Proof. Convert CFG to CNF. There are $2|w| - 1$ steps in any derivation ($|w| - 1$ steps to go from initial start non-terminal S of length 1 to non-terminals of length $|w|$; then $|w|$ conversions to terminal). Then try all strings of length $2|w| - 1$. □

Theorem 4.4. *$E_{CFG} = \{\langle G \rangle | G$ is a CFG and $L(G) = \emptyset\}$ is decidable.*

Proof. Work backwards. Mark all terminals, then mark all variables leading to symbols that are already marked. If start variable marked, *accept*; otherwise *reject*. □

Theorem 4.5. *EQ_{CFG}, equivalence problem for CFGs, is undecidable. Procedure for DFA equivalence does not work since CFLs not closed under union or intersection.*

Theorem 4.6. *Some set of languages are undecidable (and unrecognizable, even), set of all languages is uncountable, but set of all TM is countable.*

Theorem 4.7. *A_{TM} is undecidable. A_{TM} is recognizable. $\overline{A_{TM}}$ is unrecognizable; if it were then we could decide A_{TM}.*

Proof. By diagonalization. Suppose some TM H decides A_{TM}. Then can tell if M accepts w or not. Use H to create D on input $\langle M \rangle$ that simulates H on $\langle M, \langle M \rangle \rangle$, doing opposite of what M does on its own description. Then D on input $\langle D \rangle$ follows H's result on $\langle D, \langle D \rangle \rangle$, aka the opposite of how D would decide $\langle D \rangle$. Contradiction! □

Theorem 4.8. *Undecidable problems are closed under everything. This implies that undecidable languages are closed under **complement**. Recognizable problems are **not closed** under **complement**.*

Theorem 4.9. *$HALT_{TM}$ (halting problem for TMs) is undecidable.*

Proof. Suppose you could tell if a TM halted. Then you could solve A_{TM} by rejecting inputs that don't halt, and then running M only on halting w. □

Theorem 4.10. *E_{TM} (emptiness problem for TMs) is undecidable.*

Proof. Mapping reduce $A_{TM} \leq_m \overline{E_{TM}}$: $\langle M, w \rangle \mapsto M_w$ (ignore all inputs and run only on w). Clearly M accepts $w \iff M_w$ is not empty. Undecidable languages closed under complement. □

Theorem 4.11. *E_{TM} is also unrecognizable.*

Proof. Map $\overline{A_{TM}}$ E_{TM}: $\langle M, w \rangle \mapsto M_w$. M does not accept w ( ) $L(M_w)$ is empty. ☐

Theorem 4.12. *$EQUAL_{TM}$ and $\overline{EQUAL_{TM}}$ are both unrecognizable.*

Proof. 1) $\overline{A_{TM}}$ $EQUAL_{TM}$: $\langle M, w \rangle \langle M_w, M_{reject} \rangle$. If M does not accept w, then both languages empty.

2) $\overline{A_{TM}}$ $\overline{EQUAL_{TM}}$: $\langle M, w \rangle \langle M_w, M_{accept} \rangle$. If M does not accept w, then M_w empty but M_{accept} the entire language.

☐

4.3 Proof Concepts and Examples

4.4 Problem Set Results

Problem 4.13. *$AMBIG_{CFG}$ is undecidable.*

5 Reducibility

5.1 Key Definitions

Definition 5.1. A linearly bounded automaton (LBA) is a **1-tape TM** (not an automaton!) that cannot move its head off the input portion of the tape; i.e. it has work-space linear in the size of the input.

Definition 5.2. A configuration of a TM is a triple (q, p, t) encoding state, head position, and tape contents

Definition 5.3. Encode accepting computation history as a list of configurations $C_1 \# C_2 \# \ldots \# C_{accept}$

5.2 Key Results

Theorem 5.4. *Reductions and mapping reductions are equivalent notions.*

Theorem 5.5. *A_{LBA} is decidable.*

Proof. Keep a counter for steps. At most $|Q| \cdot n \cdot |\Gamma|^n$ unique configurations, so if more than that TM is looping. If accept/reject by then, okay; if not, *reject*. □

Theorem 5.6. *E_{LBA} is undecidable*

Proof. AFSOC E_{LBA} is decidable; then construct LBA $L_{M,w}$ that accepts valid computation histories of M on w. Run decider on $L_{M,w}$; if empty, *reject*. Otherwise, accept. Contraction. *Remark:* uses computation history method □

Theorem 5.7. *$PCP = \{\langle P \rangle | P$ has a match$\}$ is undecidable*

Recall that the PCP takes in as input P a collection of pairs of dominoes and tests for a match (finite sequence of dominoes in P where concatenation leads to equal values on top and bottom at every position).

Proof. Show A_{TM} reducible to PCP, using computation history method. Given M, w as input create PCP instance with dominoes as valid transitions in terms of configuration changes (top to bottom). This can yield a "partial match" (top of dominoes is precisely one configuration behind bottom dominoes). Now, key step: include a domino with a q_{accept} on top and q_{accept} on bottom. That way, if can get partial match (end with q_{accept} on bottom), can "finish off" by padding in blank space with these hooked dominoes, yielding match. Impossible to get match otherwise. Clearly if PCP has match then there is a valid computation history, hence M accepts w.

□

Theorem 5.8. *ALL_{CFG} is undecidable (CFL hits entire language).*

Proof. Make PDA that tests if input is accepting computation history for M on w; accept if *not*. Then test for all-ness; reject if yes, accept if no. □

5.3 Proof Concepts and Examples

5.4 Problem Set Results

6 Advanced Topics in Computability Theory

6.1 Key Definitions

6.2 Key Results

6.3 Proof Concepts and Examples

6.4 Problem Set Results

7 Time Complexity

7.1 Key Definitions

Definition 7.1. **NP** is the class of languages with polynomial time verifiers.

7.2 Key Results

Theorem 7.2. *Every $t(n) \geq n$ time multitape TM has an equivalent $O(t^2(n))$ time single-tape machine.*

Proof. Simulating a step requires a $t(n)$ scan for each of k branches. The multi-tape TM takes $t(n)$ time/steps, so simulating takes $O(t(n)t(n)) = O(t^2(n))$ time. □

Theorem 7.3. *Every $t(n) \geq n$ nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM.*

Proof. The single-tape essentially explores the NTM's computation tree via DFS (to simulate each branch of computation). There are at most b valid transitions at each NTM step, and the NTM runs in $t(n)$ time $\Longrightarrow$ there are $O(b^{t(n)})$ leaves. The number of leaves in a tree is basically half the number of all nodes $\Longrightarrow$ there are $O(b^{t(n)})$ nodes. Exploring each branch of computation is bounded by $t(n)$ so total time to simulate all branches is $O(t(n)b^{t(n)}) = O(2^{t(n)})$ □

Theorem 7.4. $PATH \in P$.

Proof. Doing BFS takes polynomial time. □

Theorem 7.5. *Let $RELPRIME = \{\langle x, y\rangle \mid x$ and y are relatively prime$\}$. $RELPRIME \in P$.*

Proof. Can't simply loop through all integers less than x, y since exponentially many (in length of representation). Instead, use Euclidean algorithm.

Define the algorithm $E =$ "On input $\langle x, y\rangle$:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \pmod y$
3. Exchange x and y
4. Output x."

Then just run E and check if it returns 1 or not. □

Theorem 7.6. *Every CFL is in P.*

Proof. Recall that CFGs can be converted to CNForm and all derivations of a CNForm grammar require only $2|w| - 1$ steps on input w (2.26 in the book). Naively, testing all derivations of length $|w|$ to see if they match could take exponential time, *so instead we use DP.*

The subproblems $DP(i, j)$ are whether $w_i \dots w_j$ can be generated by the CFG. The idea is if w is derivable, some sequence of substring splits must exist to get the string down to individual symbols.

There are n^2 such subproblems. Store the variable that generates string $w_i \dots w_j$ in a memo table at (i, j). So the base cases are $(i, i) = A$ for rules $A \to w_i$. For each subproblem, we need to loop through n split locations and then a constant r rules $A \to BC$ to check if some B and C form the desired split substrings (check memo table at left and right splits to see if they match B and C, store A at (i, j) if yes.

Check if S is in memo position $(1, n)$ (if yes, then following S will eventually yield w). Yes $\to$ accept, reject otherwise. There are n^2 subproblems; looping through splits is $n \Longrightarrow O(n^3)$ overall runtime. □

Theorem 7.7. *Recall $HAMPATH = \{\langle G, s, t\rangle | G$ is a* ***directed*** *graph with a Hamiltonian path from s to $t\}$. Hamiltonian means all nodes are visited and each node is visited exactly once. HAMPATH is in NP, but $\overline{HAMPATH}$ is not in NP.*

Proof. A certificate for HAMPATH is simply the path – verify that it visits all nodes once and that it goes from s to t.

It is difficult to provide a certificate to show that a graph *never* has a HAMPATH. □

Theorem 7.8. *COMPOSITES is in NP. It is also in P.*

Theorem 7.9. *$CLIQUE = \{\langle G, k\rangle | G$ is an* ***undirected*** *graph with a k-clique$\}$. CLIQUE is in NP. It is unclear if $\overline{CLIQUE}$ is in NP.*

Proof. The CLIQUE is the certificate. A poly-verifier can check if G contains all edges connecting nodes in the certificate clique. □

Theorem 7.10. *SUBSET-SUM = $\{\langle S, t\rangle | S = \{x_1, \ldots x_k\}$ and some subset sums to target $t\}$ is in NP. Also pseudo-polynomial in size of set by DP. It is unclear if $\overline{\text{SUBSET-SUM}}$ is in NP (how would you know for sure?)*

Theorem 7.11. *$SAT = \{\langle\phi\rangle | \phi$ is a satisfiable Boolean formula$\}$. It is not the language of assigments themselves. $SAT \in P \iff P = NP$ since SAT is NP-Complete.*

Theorem 7.12. *If $A \leq_p B$ and $B \in P$ then $A \in P$. Proof is obvious (chain of polynomial computations).*

Theorem 7.13. *CLIQUE is NP-complete. Recall $3SAT$ is an AND of OR clauses. $3SAT \leq_p CLIQUE$*

Proof. This is a crucial proof concept. We will convert formulas to graphs, where components of the graph mimic the function of the formula.

Given ϕ with k clauses, we poly-reduce with $f(\phi) = \langle G, k\rangle$ (so we are aiming to create a k-clique). The idea is to have triples of nodes encoding the behavior of each clause. All nodes are connected with edges barring two exceptions: 1) nodes that are contradictory (this helps w/ backward direction in particular) and 2) nodes from same triple cannot be connected (we need *exactly* k nodes in the clique).

($\Longrightarrow$): If $\phi \in 3SAT$, then to form a k-clique in G, include a node corresponding to a true literal in each clause of ϕ (if more than once than potentially larger than k-clique). The edge conditions from above automatically create a k-clique, since nodes are not contradictory and also not from same clause.

($\Longleftarrow$): If there is a k-clique in G, then make the literal corresponding to the included node of each triplet true. This satisfies ϕ. No problems arising from contradictory assignment since not allowed to be in clique by edge restrictions. □

Theorem 7.14. *If B is NP-Complete and $B \leq_p C$ for C in NP, then C is NP-Complete. Proof is fairly obvious; every problem must poly-reduce to C. Well all problems already poly-reduce to B, which poly reduce to C (chain of poly-reductions is poly).*

Theorem 7.15 (Cook-Levin)**.** *SAT is NP-Complete. Remark: serves as the basis for many other NP-Complete proofs. Review PCP for more precision. See problem 7.41 for practice.*

Proof. This is a pretty messy proof. Essentially, we need to convert input M, w to formula $\phi_{M,w}$ that tells us whether or not M accepts w. The idea is to use a $n^k \times n^k$ configuration *tableau* (one nondeterministic branch's configuration history; n^k rows for max time and n^k cells (width) since runtime n^k upper bounds cell usage). Note the tableau only contains *one* branch's history, but the ϕ we construct represents *all possible tableaus* for M on w.

The idea is to *create a formula ϕ that tells us if a tableau is satis able (i.e. accepts some input)*. The ϕ is constructed as an AND of 4 parts (omitting details for key ideas):

ϕ_{cell} : Checks if all tableau cells has one and only one assignment
ϕ_{start} : Checks if cells of first row are precisely a start configuration
ϕ_{accept} : Checks if cells of last row are an accept configuration (row gets carried down if accept early)
ϕ_{move} : Checks all 2×3 windows across all positions (i, j) for valid variable assignments (i.e. legal move)

It follows that M accepts $w \iff \phi \in SAT$ (if M accepts w some configuration accepts, hence tableau is accepting, so ϕ is true. Reverse is similar). ϕ is basically a bunch of groupings of literals for each cell position, so it is indeed $O(n^{2k})$, a poly-reduction. It is also easy to check that $SAT \in NP$ (just use satisfying assignment as certificate).

Note that this proof would not work with a 2×2 window in ϕ_{move}. □

Theorem 7.16. *$3SAT$ is NP-Complete.*

Proof. Again, assignment is certificate, so $3SAT \in NP$. For NP-hard, we slightly modify the previous proof. First, we convert each sub-ϕ to CNF form (really just ϕ_{move}, since others are already in CNF). To do so, note thata an OR of ANDs can be written as an AND of ORs. Then the outer-AND is just now a regular AND over ORs, hence we have a CNF.

Now, to get each clause to have exactly 3 literals: for all clauses with less than 3, just duplicate literals until you get to 3. If more than 3 literals, then note you can split a clause into an AND of clauses with dummy variables (assigned at will), like so:

$$(a_1 \vee a_2 \vee \cdots \vee a_n) = (a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{n-3}} \vee a_{n-1} \vee a_n)$$

□

Theorem 7.17. *P is closed under union, intersection, complement, concatenation, and Kleene star.*

Proof. https://www.cs.umd.edu/~gasarch/COURSES/452/F14/poly.pdf □

Theorem 7.18. *NP is closed under union, intersection, and concatenation; but is not known to be closed under complement.*

Proof. http://people.cs.aau.dk/~srba/courses/tutorials-CC-10/t13-sol.pdf □

7.3 Proof Concepts and Examples

Example 7.19. Simulating multi-tape machine on single-tape machine.

A single tape TM S stores each of the multi-tape TM's tapes horizontally. There are special dotted tape symbols to represent a head position.

To simulate M, S scans across its tape to find where the dotted symbols are (representing M's tape heads). Makes another pass to update tape contents according to M. If one of the multi-tapes needs more space, S shifts all its tape contents right by one.

Each multi-tape TM step is simulated in is $O(t(n)) \cdot k$ time. Multi-tape runs in $O(t(n))$ time, i.e. steps. So $O(t^2(n))$ time to simulate.

Example 7.20. Use DP to bring exponential time problems down to poly-time.

Example 7.21. Converting into $3CNF$ form:

Duplicate literals (does not change satisfiability) to reach 3. To reduce to 3:

$$(a_1 \vee a_2 \vee \cdots \vee a_n) = (a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{n-3}} \vee a_{n-1} \vee a_n)$$

7.4 Recitation Results

Example 7.22. $P \subseteq NP$, but unknown if $P = NP$ or $P \neq NP$.

Example 7.23. $PATH \in P$. (Run BFS and mark edges, taking polynomial time. Accept if t is reached, reject otherwise.)

Example 7.24. SUBSET-SUM is NP-Complete (reduction from 3SAT)

Easy to show SUBSET-SUM in NP (verify element subset sums to target).

NP-Hard:

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

Each row is an element in the set S. The target t is the last row. Each variable $x_i \Rightarrow y_i, z_i$ for $i = 1, \dots l$. The left half of the table represents an assignment ($y_1 = 1$ means $x_1 = True$ and $y_1 = 0 \implies z_1 = 1$ means $\overline{x_1} = True$. The right half specifies variable *location* within a clause. You pick each row to be a valid subset.

Then if $\phi \in 3SAT$, must have valid assignment. Choose corresponding rows y_i or z_i. Left half must sum to 1 in t per location. Upper right half can sum to anywhere between 1 − 3, but lower half can force overall sum to 3 per location. Conversely if exists subset summing to t, then take each row and assign x_i according to whether y_i or z_i chosen. Satisfies ϕ since each clause has one True literal, and no variable-value contradictions.

Time complexity: $O(l + k)^2$. Easily polynomial.

Example 7.25. $UHAMPATH$ is NP-Complete.

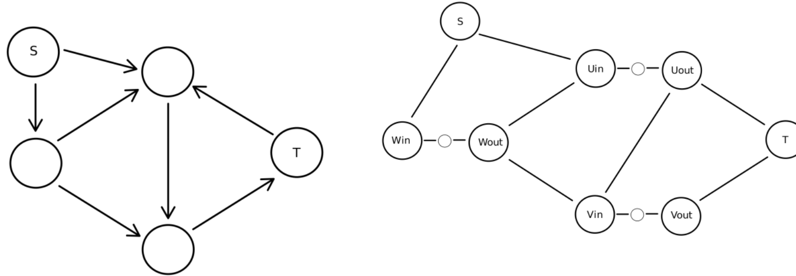
$UHAMPATH = \{\langle G, s, t\rangle | G$ is an undirected graph with path from s to t through all nodes with no repeats$\}$

Can't just map directed G to undirected G'. G having HAMPATH $\implies$ G' having UHAMPATH, but not other way around. Easy to show in NP. Verify a valid path.

NP-Hard: Map $s, u_1, u_2, \ldots, t$ to $s_{out}, u_1^{in}, u_1^{mid}, u_1^{out}, \ldots, t_{in}$. The in/out/mid represents if the node has incoming or outgoing edge.

If G has hampath then certainly G' has hampath (we already had this direction previously). But now if G' has hampath then G is also guaranteed to have hampath (the in/out encodes the direction of each edge). Roughly linear time to construct this.

Construction by example:



Example 7.26. *NAE-SAT* is NP-complete.

Recall that:

$$NAE\text{-}SAT = \{\phi \mid \phi \text{ a 3cnf satisfiable with each clause having } \geq 1 \text{ TRUE or FALSE (i.e. not all equal) literal}\}$$

Proof. First, obviously $NAE\text{-}SAT \in NP$.

For NP-hard, we reduce: $3\text{-}SAT \leq_p NAE\text{-}SAT$. For every clause $(x_1 \vee x_2 \vee x_3)$, map to:

$$(x_1 \vee x_2 \vee z_i) \wedge (\overline{z_i} \vee x_3 \vee b)$$

z_i is a dummy variable for each clause, b is dummy for entire ϕ'. Note that if $\phi \in 3SAT$ then following satisfies ϕ': if $x_1 \vee x_2$ TRUE, let z_i = FALSE; else let z_i = TRUE. Always set b FALSE. So first clause always has at least one TRUE and FALSE, second always has at least one FALSE (b) and TRUE (either x_3 TRUE to begin with or $\overline{z_i}$ forced TRUE by $x_1 \vee x_2$ being TRUE).

If $\phi' \in NAE\text{-}SAT$, then assume WLOG that b = FALSE (note that negation of any valid NAE assignment is still NAE). Then if z_i is TRUE, x_3 must be TRUE (satisfies clause). Else if z_i FALSE, either x_1 or x_2 TRUE, satisfying clause. Reduction takes $O(n)$ time.

□

7.5 Problem Set Results

Problem 7.27. *Let $MODEXP = \{\langle a, b, c, p\rangle | a, b, c, p$ are positive binary integers such that $a^b = c (\bmod p)\}$. Assume that basic arithmetical operations, such as $+$, $\times$, and mod, are computable in polynomial time. Then $MODEXP \in P$.*

Proof. Mistakes: multiplying a directly by b times; or using repeated squaring without modding at each step (would require squaring an exponentially-large number).

Main technique: repeated squaring *with* modding. Start with $r = 1$ and use bit representation of $b = b_1 \dots b_k$. If $b_i = 0$, $r = r^2 \bmod p$; else if $b_i = 1$, $r = ar^2 \bmod p$. Then after loop check if $r = c$. This runs in poly-time since only looping for len(b).

```
result = 1
for each bit of n from left to right:
    result = result * result
    if the current bit is 1:
        result = result * a
```

Figure 1: Repeated Squaring (Imagine Modding as Well)

□

Problem 7.28. *Let UNARY-SSUM be the subset sum problem in which all numbers are represented in unary, i.e., 1^k represents the number k. Why does the NP-completeness proof for SUBSET-SUM (see textbook) fail to show UNARY-SSUM is NP-complete? Show that UNARY-SSUM $\in P$.*

Proof. Proof in book relies on creating large decimal numbers to represent each variable. Doing this in unary would be 10^{k+l} for k clauses l variables, so exponential time.

Mistakes to show NP-hard: converting unary numbers to binary/hex/decimal/etc. representation (numbers require less storage, but concerned chiefly about *set size*).

Main technique: using DP. Let subproblem be L_i storing list of all possible sums from subsets of $\{x_1, \dots, x_i\}$. $L_0 = \{0\}$. Then $L_i = L_{i-1} \cup \{a + x_i | a \in L_{i-1}\}$. Accept if $t \in L_i$, else reject. n subproblems, each taking n sums. Representing L_i requires unary representation of sum; maximum sum is n; maximum n sums up until $n \implies n^2$ storage space per L_i. So overall $O(n^4)$, indeed poly-time. □

Problem 7.29. *Show that if $P = NP$, then every language $A \in P$, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete.*

Proof. Main technique: leverage $P = NP$ to solve *NP* problems as part of poly-time reduction. First, note $A \in NP$ (immediate from $A \in P$). Then WTS $B \le_p A$ $\forall B \in NP$. Some poly-NTM N decides B. Since $P = NP$ some poly-DTM M decides B. Now use M during mapping. Consider reduction D: simulate M on w in poly-time. If accept, $w \mapsto a_{in} \in A$ (guaranteed since $A \neq \emptyset$). If reject, $w \mapsto a_{out} \notin A$ (guaranteed since $A \neq \Sigma^*$). Clearly valid. Also poly-time using M.

Remark: in most situations we do not have the power to "wishfully" map input to strings *known* to be inside or outside target language (as we did with a_{in} and a_{out}). In this instance, however, we have the power of poly-time DTM granted by P=NP, hence can know "for sure" where to map input to during reduction. □

Problem 7.30. *Show that if P = NP, we can factor integers in polynomial time. (Note: The algorithm you are asked to provide computes a function, and NP contains languages, not functions. Therefore, you cannot solve this problem simply by saying "factoring is in NP and P = NP so factoring is in P". The assumption P = NP implies that all languages in NP are in P, so you need to find an NP language that relates to the factoring function.)*

Proof. Consider the language $F = \{\langle a, b, c\rangle | a, b, c$ are binary integers and $a = pq$ for $b \leq p \leq c\}$. It's easy to see $F \in NP$ (verify $a = pq$ in polytime) $\implies F \in P$ since $P = NP$ by assumption. Then, perform essentially binary search to find a factor (run poly-DTM for F on smaller intervals starting from $[2, a-1]$). Final factor call gives a factor if accept (interval $[b, b]$, else *reject*. Poly-DTM for $\log n$ # of calls $\implies$ overall poly. □

Problem 7.31. *Let $CNF_k = \{\langle\phi\rangle | \phi$ is a satisfiable cnf-formula where each variable appears at most k times$\}$. Show that $CNF_2 \in P$.*

Proof. Main technique: self-strip-down ϕ by removing clauses according to first variable (linear pass for $O(n)$ clauses). First just check to see if each variable appears more than twice $O(n^2)$. Then while ϕ is non-empty:

1. Take the first literal x. Loop through rest of clauses to see if x appears again.
2. If x appears once or twice in a *single* clause, this clause is satisfiable (assign x without affecting other clauses' satisfiability). So can safely remove clause.
3. If x appears only positively or only negatively $\bar{x}$ in two clauses, can safely remove those two clauses.
4. If x appears positive in one clause $(x \vee y)$ but negative in other $(\bar{x} \vee z)$, need to be careful. Both satisfiable iff $(y \vee z)$ satisfiable (can then assign $x, \bar{x}$ accordingly). Simpler cases: $(x) \ldots \wedge (\bar{x} \vee z)$ means $x = True \implies \bar{x} = False$ to satisfy, so replace second clause with (z). Similarly $(\bar{x}) \ldots \wedge (x \vee y) \implies (\bar{x}) \ldots (y)$ But *not possible to satisfy* if have clauses $(x), (\bar{x})$, in which case *reject*.
5. If all clauses removed, all satsifiable, hence ϕ satisfiable, so *accept*.

Remark: self-strip reduction only works with $k = 2$ (could introduce excessive clauses if considering 3 literal/clause instances). □

Problem 7.32. *CNF_3 is NP-complete (note contrast with previous problem).*

Proof. Main technique: chain together implication clauses to force all variables true or all variables false. First, recall rules of boolean logic:

$$(\bar{a} \vee b) \iff (a \to b) \quad \text{(if } a \text{ true } b \text{ must be true, otherwise no restrictions on } b\text{)}$$

First, clearly $CNF_3 \in NP$ (verify satisfying assignment and has ≤ 3 of each variable in $O(n^3)$). For NP-Hard, reduce from SAT. Consider ϕ. ϕ could have > 3 of each var x. **Trick**: make each instance of x new variable x_i. Add implication chain of x_i's, so logically equivalent to just having x. ϕ' looks something like:

$$\phi' = \phi \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \cdots \wedge (\overline{x_{k-1}} \vee x_k) \wedge (\overline{x_k} \vee x_1) \wedge (\overline{y_1} \vee y_2) \ldots$$

There's *exactly* 3 of each variable (one in original ϕ, exactly 2 in implication clauses). Crucially, $\phi \in SAT \iff \phi'$ satisfiable. Why? Well all x_i must all be the same, so equivalent to just setting single $x = True/False$.

□

Problem 7.33. *For a cnf-formula ϕ with m variables and c clauses, show that you can construct in polynomial time an NFA with $O(cm)$ states that accepts all nonsatisfying assignments, represented as Boolean strings of length m. Conclude that P = NP implies that NFAs cannot be minimized in polynomial time. Here,* ***minimizing an NFA*** *means finding an NFA with the fewest possible number of states that recognizes the same language as a given NFA.*

Proof. First, consider a NFA with c copies of $m+1$ nodes labeled $c_{i,j}$ for clause i and variable j. The final node in each "row" is an accept state. Have ϵ-transitions into the first node of each set. We want "bad" assignments to lead to an accept, so: if x_j appears positive in clause i, create a 0-transition to x_{j+1}; do the opposite for $\overline{x_j}$. If no literal appears for the current node, its assignment does not affect the current clause's truth value; so create 0 and 1-transitions out of the node. If both x_j and $\overline{x_j}$ appear in a clause, impossible to make clause false, so don't create any transitions out of it. If a clause is possible to make false, there will be a path to the accept state. Can create a chain of $m+2$ nodes to accept all strings of wrong length as well: m accept states with both 0 and 1-transitions, $m+1$th state is reject (since correct length), $m+2$nd state is accept (wrong length again).

Now, we show contrapositive: poly-time NFA minimization implies P = NP. Approach is to show $SAT \in P$. On input ϕ, create corresponding NFA that accepts all nonsatisfying assignments. Reduce in polynomial time. **Key Idea:** if reduced NFA is a single reject state, no assignment can make ϕ true. Then we know $\phi \notin SAT$. If any other form for NFA, $\phi \in SAT$. So poly-time procedure, hence $SAT \in P$ and $P = NP$.

□

8 Space Complexity

8.1 Key Definitions

Definition 8.1. A TM M runs in $f(n)$ space if M uses at most $f(n)$ tape cells on all inputs of length n. NTM N runs in $f(n)$ space if all branches halt and each branch uses at most $f(n)$ tape cells on all inputs of length n.

Definition 8.2. B is **PSPACE-complete** if:

1. $B\ 2\ PSPACE$
2. $8A\ 2\ PSPACE$, $A \quad _p B$, where p a polynomial-*time* reduction

Remark: we consider poly-*time* reductions instead of poly-*space* reductions. If we use the latter, then all problems in PSPACE reducible to each other (proof similar to PSET 4.3: reduction as strong as problem's class has the power to "know" answer to problem and thus whether to map input string to output string in/out of second language), and hence all complete.

Definition 8.3. For sublinear (log) space computation, use 2-tape TM model with read-only input. Only count cells used on read/write work tape (log max).

Remark: Log space can represent a constant number of pointers into the input.

Definition 8.4. A log-space transducer is a TM with 3 tapes; one for reading input of size n; one for working of size $O(\log n)$; output tape of unlimited size.

8.2 Key Results

Theorem 8.5. *For* $t(n) \quad n$*,* $TIME(t(n)) \quad SPACE(t(n))$

Proof. If use at most $t(n)$ steps, clearly use at most $t(n)$ cells (at most one new cell per step). *Remark*: this shows $P \quad PSPACE$.

□

Theorem 8.6. $SPACE(t(n)) \quad \bigcup_c TIME(c^{t(n)})$.

Proof. Say TM uses at most $t(n)$ tape cells. Remember, DTM must halt on all inputs. Takes at most $c^{t(n)}$ time to go through literally all possible configurations without repeats (would imply looping) before halting, where c is the number of symbols per cell.

□

Theorem 8.7. $NP \quad PSPACE$

Proof. First, $SAT\ 2\ PSPACE$: try all possible assignments on $O(n)$ tape, reusing that tape for each assignment. If one accepts, *accept*; else *reject*. Now, another theorem (consistent with what we've seen): if $A \quad _p B$ and $B\ 2\ PSPACE$ then $A\ 2\ PSPACE$. Then since all problems in NP poly-time reduce to $SAT\ 2\ PSPACE$, then all $NP \quad PSPACE$.

□

Theorem 8.8. $LADDER_{DFA} =$ *fh*B, u, v*ij*B *is a DFA and* $L(B)$ *contains a ladder* $y_1, y_2 \ldots y_k$ *where* $y_1 = u$ *and* $y_k = v$g *(can get from* u *to* v *by changing one letter at a time).* $LADDER_{DFA}\ 2\ NPSPACE$

Proof. At first glance, would want to store the ladder. But careful, could have exponential-length ladder. Also need to be careful of looping (all branches of NTM need to halt). Instead, nondeterministically guess letter change:

Let $N =$ "On input $\langle B, u, v\rangle$:

1. Let $y = u$. Let $m = |u|$.
2. For count $t \le |\Sigma|^m$ (prevent looping; not an issue to restrict length, as if exists ladder greater than t, must exist ladder less than t, since only so many unique combinations for a word):
    3. Nondeterministically change one symbol in y.
    4. Reject if $y \notin B$.
    5. Accept if $y = v$.
6. Reject (have exceeded t steps).

Remark: polynomial space necessary to store changing y (linear space reused) and t (log-representation-of-exponential-number = poly space).

□

Theorem 8.9. *$LADDER_{DFA} \in PSPACE$, somewhat more surprisingly.*

Proof. Main Technique: use recursion (also shows up for Savitch, and TQBF PSPACE-completeness).

First, consider language *BOUNDED-LADDER*$_{DFA} = \{\langle B, u, v, b\rangle | B$ a DFA and $u \xrightarrow{b} v$ by a ladder in $L(B)$ $\}$. Solved by B-L = "On input $\langle B, u, v, b\rangle$. Let $m = |u| = |v|$:

1. For $b = 1$ (base case), *accept* if $u, v \in L(B)$ and differ in ≤ 1 place, else *reject*.
2. For $b > 1$, binary recursion across all w of length $|u|$:
    3. Recursively test $u \xrightarrow{b/2} w$ and $w \xrightarrow{b/2} v$.
    4. *Accept* if both accept.
5. *Reject* (all w at this depth have failed to be a valid mid-ladder point).

Then solve $LADDER_{DFA}$ by running B-L on $\langle B, u, v, t\rangle$, where $t = |\Sigma|^m$ (again to prevent looping). Depth is $\log(t) = m = O(n)$. $O(n)$ cells at each layer $\to O(n^2)$ overall.

Remark: $|v| = O(n)$ to write and attempt to recurse across *all* w. Reuse space at *each* layer across *all* recursion histories (think of recursion tree, just $O(n)$ space at each layer).

□

Theorem 8.10. ***(Savitch)**: For $f(n) \ge n$, $NSPACE(f(n)) \subseteq SPACE(f^2(n)) \implies NPSPACE \subseteq PSPACE$. I.e. convert NTM N to equivalent TM M.*

Proof. Similar recursive algorithm to *BOUNDED-LADDER*$_{DFA}$, this time with configurations c_i, c_j instead of strings. Define CANYIELD(c_i, c_j, b) = "On input c_i, c_j, b :

1. If $b = 1$, check N's transition function to see if c_i has valid move to c_j.
2. If $b > 1$, binary recursion across all c_{mid} of length b:
    3. Recursively test $c_i \xrightarrow{b/2} c_{mid}$ and $c_{mid} \xrightarrow{b/2} c_j$.
    4. *Accept* if both accept.
5. *Reject* (all c_{mid} at this depth have failed to be a valid mid-ladder configuration).

Need to try x (change within constant space) at each layer. $|\phi|$ layers so $O(n)$ space overall.

□

Theorem 8.11. *TQBF is PSPACE-complete. Main PSPACE-complete language.*

Proof. First, TQBF $2\ PSPACE$ (almost-linear recursion). Let $D =$ "On input $h\phi i$:

1. If ϕ has no quantifiers $!$ all variables assigned $!$ ϕ either true (*accept*) or false.
2. If $\phi = 9x[\psi]$, evaluate ψ with $x = True/False$ recursively. *Accept* if either accepts; *reject* if else.
3. If $\phi = 8x[\psi]$, evaluate ψ with $x = True/False$ recursively. *Accept* if both accept; *reject* if else.

Now, to show PSPACE-hard, want to design $\phi_{M;w}$ to simulate M on w. Mix idea of recursion with computation tableau from Cook-Levin. Picture this tableau detailing a computation history for (deterministic) M on w. Could try Cook-Levin idea directly, but would lead to exponentially-large non-QBF ϕ.

We want to create ϕ that both encodes a valid tableau but also does not blow up in size. First, define $\phi_{c_1;c_2;1}$ as in Cook-Levin (2 3 valid transition window idea). Now, it seems we want some recursive formula like:

$$\phi_{c_i;c_j;b} = 9c_{mid}[\phi_{c_i;c_{mid};b=2} \wedge \phi_{c_{mid};c_j;b=2}] = 9c_{mid}(9c'_{mid}[\text{same size}] \wedge 9c''_{mid}[\text{same size}]) = \dots$$

Unfortunately, this doubles ϕ at each recursive layer; recall max steps is exponential d^{n^k} so log $!$ polynomial layers, so exponential-sized ϕ. Instead, main technique here is to "wrap" the exists and double AND by using 8 to replace two 9's as follows:

$$\phi_{c_i;c_j;b} = 9c_{mid}8(c_1, c_2)\ 2\ f(c_i, c_{mid}), (c_{mid}, c_j)g[\phi_{c_1;c_2;b=2}] \quad (1)$$

So now instead of doubling, only add roughly $O(n)$ to ϕ at each layer $!$ $O(n^{2k})$ overall reduction: $\phi_{M;w} = \phi_{c_{start};c_{accept};t}$, where $t = d^{n^k}$.

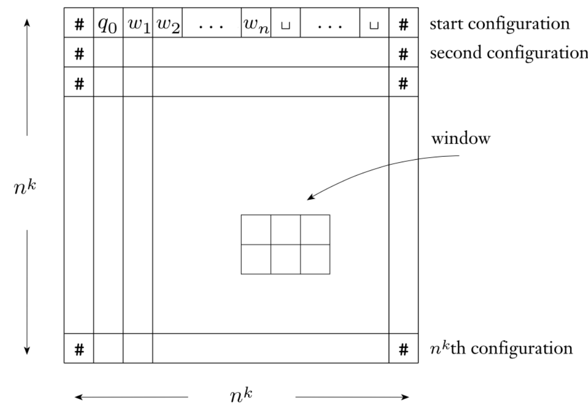


Figure 2: Computation history tableau (with d^{n^k} rows; can go through poly-space symbol combinations)

□

Theorem 8.12. *Let FORMULA-GAME = $fh\phi ij$ player 9 has winning strategy for formula game associated with ϕg Then FORMULA-GAME is PSPACE-complete.*

Proof. Recall that 9 has a winning strategy if it is always able to force a win on $\phi = 988998\dots[\psi]$. I.e., it is always able to make ψ true by choosing the right assignment, regardless of how player 8 plays/chooses assignments. Well, if 9 is able to do this, then ϕ has a satisfying assignment! Clearly then *FORMULA-GAME* $= TQBF$, hence PSPACE-complete.

□

Theorem 8.13. *GG is PSPACE-complete.*

Recall that the Generalized Geography Game is played on a *directed* graph, with two players taking turns picking nodes to form a *simple* path (no repeats); first player with no valid node to move towards loses.

Define $GG = \{\langle G, a\rangle \mid$ Player I has a **forced win** on graph G starting at node $a\}$ (guaranteed win if both players play optimally).

Proof. First, we need to show that $GG \in PSPACE$. This is actually a non-trivial proof: consider DFS/recursion-ish approach. First, remove all edges causing loops (since only want simple path). Then, let $D =$ "On input $\langle G, a\rangle$:

1. If a has no out-edges, current player loses. *Reject.*
2. Otherwise, recurse. Remove node a and associated edges to form G'
3. For each neighbor b of a:
    4. Run D on $\langle G', a\rangle$
5. If all return accept, then this current player has no valid move to win, so *reject.* Otherwise, it can make the "right" move leading to a loss for next player, so *accept.*

Space analysis: reuse constant space at each layer to write down node and check for out-egdes. $O(n)$ layers since n nodes. So $O(n)$ space.

Now, for PSPACE-hard, we reduce from TQBF. Construct G to mimic formula game on ϕ. This is a really messy proof, but the idea is to construct "diamonds" to as variable gadgets, and corresponding clauses:

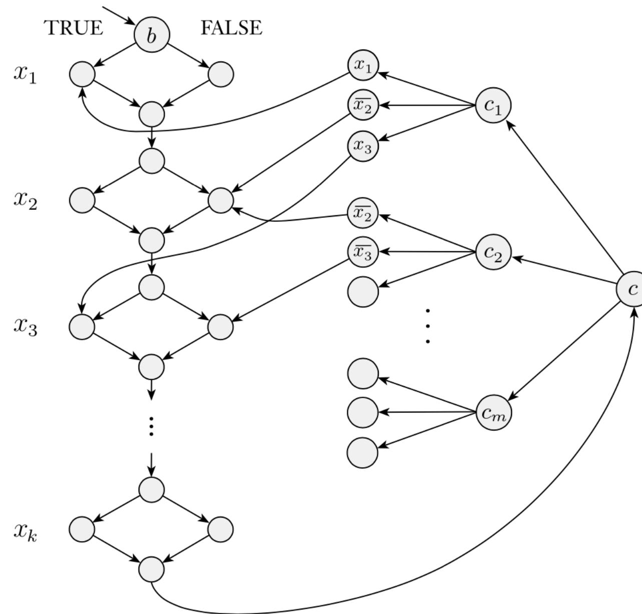


FIGURE 8.16
Full structure of the geography game simulating the formula game, where
$\phi = \exists x_1 \forall x_2 \cdots \exists x_k \, [(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \cdots) \wedge \cdots \wedge (\quad)]$

The left column basically simulates variable assignment for $x_1, \ldots, x_k$ by players $\exists$ (Player 1) and $\forall$ (Player 2). Note that each step gives freedom of choice to one player (wedge in diamond), but forces the other

player into a subsequent move. Also note that TRUE and FALSE aren't literal labels; they are being "simulated" by choosing left/right node. Finally we assume $\exists$ plays last (always can convert into this form); so Player 2 always ends up with choice at node c.

The right columns, especially the loops back to the original left column nodes (left is positive, right if negated), are critical for this proof to work. Note ϕ satisfiable, then regardless of which clause Player 2 picks at c, Player 1 can choose a node x corresponding to TRUE literal. Then the only out-edge connects back to a node corresponding to a boolean value that *forced* x to be true. This means the **node has already been visited**. So Player 2 is stuck, and loses. Now, if ϕ unsatisfiable, some clause unsatisfied, so Player 2 picks that clause. Then Player 1 has no TRUE node to pick, so picks whatever. Then Player 2 has can visit the left-column node, since the node not being TRUE (right column) means it wasn't *assigned* TRUE (left column) in the first place. But now Player 1 forced to make a move to a node that has certainly been visited before (straight line down), so Player 1 loses.

Reduction time: polynomial ($O(n)$ for left column, $O(n)$ again for right columns).

□

Theorem 8.14. $\{ww^R \mid w \in \Sigma^*\} \in L$

Proof. Store pointers for start and end of input, traverse inwards. If ever don't match, reject; otherwise accept. Only use two pointers throughout entire computation, so constant space. □

Theorem 8.15. $L \subseteq P$ *(relates* ***sublinear space*** *with* ***polynomial time*** *– not immediate at all!)*

Proof. Suppose M decides A in log space (only uses that much space on input w). A configuration for M on w takes form (q, p_1, p_2, t) – state, read tape head, write tape head, tape contents $\to |Q| \cdot n \cdot O(\log n) \cdot d^{O(\log n)} \le O(n^k)$ max time (since M must halt). So M runs in poly-time, hence $A \in P \implies L \subseteq P$. □

Theorem 8.16. $NL \subseteq P$*, somewhat more surprisingly (since unsure if L = NL).*

Proof. Main technique: **configuration graph** $G_{M,w}$ to describe computation of NTM M on w. Nodes are configurations of M on w; edges given by transition function (can yield $c_i \to c_j$ in one step). Following is a poly-time algorithm T for A:

1. Construct $G_{M,w}$. $O(n^k)$ nodes, as seen in previous problem – but not deterministic (configuration order not predetermined), so we include valid edge transitions to simulate non-determinisim.
2. Check if $G_{M,w}$ has a path from c_{start} to c_{accept} (recall PATH $\in$ P). Having a path means M on w has accepting computation history, so *accept*; otherwise *reject*.

□

Theorem 8.17. $NL \subseteq SPACE(\log^2(n))$*, by Savitch (even though* $f(n) < n$*).*

Theorem 8.18. *If* $A \le_L B$ *and* $B \in L$ *then* $A \in L$*.*

Proof. This seems obvious (we have seen versions of this for P, NP, and PSPACE), but is actually not so straightforward. Consider TM for $A =$ "On input w:

1. Compute $f(w)$.
2. Run decider for B on $f(w)$. Issue is that $f(w)$ itself may not be sublinear. A needs to simulate B on $f(w)$ *internally*, it cannot call B outside of the TM and magically gain access to the result. Instead, don't store $f(w)$ in its entirety. Basically, re-compute symbol-by-symbol of $f(w)$ on-the-fly and then feed into B as needed. This takes an enormous amount of time, but is indeed log-space.

□

Theorem 8.19. *$PATH$ is NL-complete. This is the main NL-complete language we have.*

Proof. Easy to show in NL: nondeterministically select nodes that path should follow, then use pointer to traverse and confirm that path.

Now, to show NL-hard, must log-transduce all $A \in NL$ to PATH. Consider $A \in NL$. Some TM decides if $w \in A$ or not. This TM has a computation history. Have the log-space transducer list out all possible configurations as nodes (go through all strings of length $c\log(n)$ representing max configuration length and test if legal config for M on w). Then go through all pairs (c_1, c_2) and check if valid from M's transition function. Listing out all nodes gives you a graph G, then list c_{start} and c_{accept}. So if $w \in A$ then obviously G has a path from $c_{start} \rightarrow c_{accept}$; otherwise no path.

□

Theorem 8.20. *$\overline{2SAT}$ is NL-complete.*

Proof. First, $\overline{2SAT} \in NL$. This is actually quite tricky itself (and not proven in class). First, we map ϕ to a graph of implications: construct nodes x_i and $\bar{x}_i$ for each $x_i \in \phi$. Then for each clause, list edges of form $\bar{x}_i \rightarrow x_j$ and $\bar{x}_j \rightarrow x_i$, both equivalent to $(x_i \vee x_j)$. Then if $\phi \in \overline{2SAT}$, then there must exist some boolean contradiction, so G has path from some $x_i \rightarrow \bar{x}_i$ or $\bar{x}_i \rightarrow x_i$; can check for all i using just log-space via pointer (would take an enormous amount of time, but irrelevant here),

To show NL-hard, log-reduce from PATH. Consider $\langle G, s, t\rangle$. Make ϕ with clauses $(u \rightarrow v) = (\bar{u} \vee v)$ for each edge $(u, v) \in G$. Then add clauses $(t \rightarrow \bar{s})$ and $(s \vee s)$. This construction is quite slick: if G has path from $s \rightarrow t$, then these implications of type $(u \rightarrow v)$ will force all variables simulating nodes on this path to be of same type, leading to contradiction from final $\bar{s}$ and initial s. If no path exists, then ϕ satisfiable by setting all nodes reachable from s to be TRUE and others FALSE; will never cause contradiction and every clause will have at least one true value (if reach u, must reach v; if cannot reach u, cannot reach v).

□

Theorem 8.21. *L = P and L = NL are unsolved questions. If L = P then L, NL, P all collapse into same class since $L \subseteq NL \subseteq P$*

Theorem 8.22. *$NL = coNL$ (Immerman-Szelepcsenyi). This is a* **long** *proof! We omit much of the details to capture the main ideas and techniques.*

Proof. First, we show that if some NL-machine computes $c_d = |R_d|$ (# nodes reachable from s), then some NL-machine computes $path_{d+1}$, which tells us whether or not u is reachable from s within $d+1$ steps. It's easy to compute c_{d+1} using $path_{d+1}$. Doing this iteratively will yield $c_{m-1} \rightarrow path_m$. Now, consider the following algorithm: "On input $\langle G, s, t\rangle$:

1. Compute c_d
2. Let $k = 0$
3. For each node u:
   4. Nondeterministically go to one of following two branches:
      5. Nondeterministically pick a path from s to u of length $\leq d$. If fail (check using log-pointer), *reject*. If not fail and u has edge to t, output YES on branch; else set $k = k + 1$
      6. Skip u on this branch (ideal, since u may not be reachable at all; want a good nondeterministic branch to make this judgement call
7. If $k \neq c_d$, *reject* (branch did not test all possible nodes to see if they were t).
8. Output NO (tried all nodes and did not output YES yet).

As mentioned, we can compute $path_m$ iteratively using the above procedure, which allows us to decide $\overline{PATH}$ via the following algorithm. "On input $\langle G, s, t\rangle$:

1. $c_0 = 1$
2. Compute c_{d+1} from c_d, all the way up to c_{m-1}, which will give you $path_m$
3. Then if $path_m(\langle G, s, t\rangle)$ outputs YES, *reject*.
4. Otherwise, *accept*.

Remark: PATH itself is obviously also decidable via the above. Too, note that basically an exponential number of functions are being passed around as subroutines $path_m$ called on each u to decide c_m, etc. but we only ever store a handful of pointers and counters (c_i, c_{i+1}, u, k, m, t, etc.) suffices throughout the entire computation! Hence in log-space.

□

8.3 Recitation Results

Theorem 8.23. *$A_{LBA} = \{\langle M, w\rangle | M$ is an LBA that accepts string $w\}$. A_{LBA} is PSPACE-complete.*

Proof. First, need to show $A_{LBA} \in PSPACE$. Well, since M decides w in $O(n)$ space, just have a TM simulate M on w using $O(n)$ space. Also need to **ensure halting by storing a counter**. $O(d^n)$ possible configurations before looping; bin representation is $O(n) \rightarrow$ linear.

Now, to show PSPACE-hard, reduce from TQBF. Recall that we can decide TQBF in $O(2n)$ space (n cells to change test modified ϕ). But LBAs can only use n space. Easy fix: make input longer: $f(\phi) = \phi\#^{n}$. Then LBA M simply checks if input in proper form, and uses blank space to store variable assignments. Clearly $\phi \in TQBF \iff f(\phi) \in A_{LBA}$. Polynomial time to include n more cells. □

Theorem 8.24. *BIPARTITE $\in$ coNL. Recall undirected graph is bipartite if can be split in two groupings, with edges only spanning groupings.*

Proof. First, we prove a relevant fact from graph theory:

Lemma 8.25. *G is bipartite $\iff$ all cycles are even.*

Proof. ($\Longrightarrow$): G bipartite $\Longrightarrow$ two disjoint groupings as defined above $\Longrightarrow$ getting from one node to itself requires crossing over, then crossing back $\Longrightarrow$ even steps.

($\Longleftarrow$): Suppose G has all even cycles. For each connected component, choose node u and color v reachable in odd steps (edges) red, even steps blue. Observe that having nodes of same color only one edge apart implies odd cycle. So all even $\Longrightarrow$ no one-edge-apart-same-color nodes $\Longrightarrow$ existing bipartition (colors are sets).

□

Now, we equivalently show $\overline{BIPARTITE} \in NL$ by showing G has an odd cycle in NL-space. Nondeterministically guess u, storing it in log-space. Then keeping a counter i of nodes (starting at 1), guess next node and set $i := i + 1$. If next no edge from *prev* to next node, *reject*. If next node is u and i odd, *accept*. If loop through all $|V|$ nodes and no accept yet, *reject*. Store only *prev*, u, and counter, so indeed log-space.

□

Theorem 8.26. *STRONGLY-CONNECTED is NL-Complete*

Proof. First, it's easier to show $\overline{\text{STRONGLY-CONNECTED}}$ is in NL, and by NL=coNL $STRONGLY$ $CONNECTED \in$ NL. Also recall that PATH $\in NL$, so $\overline{PATH} \in NL$ as well. Invoke NL = coNL!. Then to test if G not strongly connected, guess the right node pairing (u, v) and simulate decider for

$\overline{PATH}$ on input $\langle G, u, v\rangle$, accepting if return yes. Decider runs in NL space, so simulator in NL space as well.

Showing NL-hard is actually quite easy. Make all node pairings connected by an edge (loop through nodes using two pointers, double for loop), except for s and t. Clearly path from $s \to t$ means all node pairings have path; otherwise no. □

. . . .

8.4 Proof Concepts and Examples

Example 8.27. $O(f(n))$ space $\implies$ $2^{O(f(n))}$ time before machine loops (think about all possible different configurations machine could take on before repeating one).

Example 8.28. (Problem 8.8): We can test the equivalence of two regular expressions in polynomial space

Problem 8.29. *$A_{DFA} \in L$. Solution: store pointer for state and head position on input.*

Problem 8.30. *Show NL closed under complementation, union, concatenation, and kleene star.*

Problem 8.31. *Show that $GM = \{\langle B\rangle | B$ is a position in generalized go-moku, where "X" has a winning strategy$\}$ is in PSPACE.*

Proof. Use recursion (sort of similar to PSET 6.3). Consider next-move cases for player X and O; consider how you would know if X could win at any given step (hint: $\exists$ win for X and $\forall$ win for O). □

Problem 8.32. *Show that CYCLE is NL-complete*

Proof. Use level-sets. □

. . . .

8.5 Problem Set Results

Problem 8.33. *Let SET-SPLITTING $= \{\langle S, C\rangle | S$ is a finite set and $C = \{C_1, \ldots, C_k\}$ is a collection of subsets of S, where the elements of S can be colored red or blue so every C_i has at least one red element and at least one blue element$\}$. Show that SET-SPLITTING is NP-complete.*

Proof. Poly-reduce from NAE-SAT. First, make add an element to S for each literal of ϕ. Map clauses to subsets. Then create some additional subsets of the form $(x_i \vee \overline{x_i})$. So if $\phi \in SAT$, then color true literals blue and false literals red (never have all in a single clause). Now if set splittable, make blue elements true literals and vice versa for red (this is why we need extra sets, to prevent a coloring where two contradicting nodes are both blue/red). □

Problem 8.34. *Say that two Boolean formulas are equivalent if they have the same set of variables and are true on the same set of assignments to those variables (i.e., they describe the same Boolean function). A Boolean formula is minimal if no shorter Boolean formula is equivalent to it. (For definiteness, say that the length of a Boolean formula is the number of symbols it has.) Let MIN-FORMULA be the collection of minimal Boolean formulas. Show that MIN-FORMULA $\in$ PSPACE.*

Proof. We can show this directly. Consider ϕ – how can we be sure it's minimal? In poly-space, write down all possible ϕ' where $|\phi'| < |\phi|$, as well as all possible assignments to each ϕ'. Test *every* set of assignments on ϕ and ϕ'. If ever disagree, move on to next ϕ'. If not ϕ', move onto next assignment. If

make it to end of loop of assignments, ϕ and ϕ' must be equivalent, so *reject* (ϕ not minimum). If make it to end of outer loop of ϕ' without reject, *accept* (ϕ' indeed minimal).

□

Problem 8.35. *Two parts:*

1. *Explain why the following argument fails to show that MIN-FORMULA $\in coNP$:*
   (a) *If $\phi \notin$ MIN-FORMULA, then ϕ has a smaller equivalent formula.*
   (b) *A NTM can verify that $\phi \in \overline{\text{MIN-FORMULA}}$ by guessing that formula.*
2. *Show (despite part a) that if $P = NP$, then MIN-FORMULA $\in P$.*

Proof. For 1), it doesn't suffice to *only* guess a shorter formula; you need to also check ϕ and ϕ' equivalent on all inputs (which is not doable in polynomial time – in fact, exponential).

For 2), we show $\overline{\text{MIN-FORMULA}} \in NP$ (equivalent problem formulation since P=NP). First, nondeterministically guess shorter ϕ'. How do we check equivalence with original ϕ? Well, easier to check *in*-equivalence, which is in NP (guess the disagreeing assignment) $\to$ can decide *in*-equivalence in P since P=NP, which means equivalence checkable in P (closed under complement). Then check if ϕ' and ϕ equivalent in poly-time on guess ϕ'. This is a valid NTM procedure!

□

Problem 8.36. *For any positive integer x, let x^R be the integer whose binary representation is the reverse of the binary representation of x. (Assume no leading 0's in the binary representation of x.) Define the function $R^+ : \mathcal{N} \to \mathcal{N}$ where $R^+(x) = x + x^R$.*

1. *Let $A_2 = \{\langle x, y\rangle | R^+(x) = y\}$. Show $A_2 \in L$.*
2. *Let $A_3 = \{\langle x, y\rangle | R^+(R^+(x)) = y\}$. Show $A_3 \in L$.*

Proof. Showing a) is straightforward, but tedious. Idea is to compute bit-by-bit (hence in log space) and print out each bit of the sum. Keep pointer for value of kth-to-last and k^{th} bit. Another pointer to store sum. Another pointer to mod this sum. This is bit value of $x + x^R$ at k^{th} digit. Compare with y (so another pointer necessary). But, need to account for carrying a 1 (sometimes), so have a pointer for that as well. So indeed a valid log-space procedure.

Now for b), we can reduce $A_3 \leq_L A_2 \in L$. Indeed, consider $\langle x, y\rangle \mapsto \langle R^+(x), y\rangle$. Computing $R^+(x)$ takes log-space by part a). Clearly $\langle x, y\rangle \in A_3 \implies \langle R^+(x), y\rangle \in A_2$. *Remark:* implicitly uses the compute-on-the-fly technique, since can't store output $\langle R^+(x), y\rangle$ internally in the decider for A_3. □

Problem 8.37. *Show that A_{NFA} is NL-complete.*

Proof. First, it's obvious that $A_{NFA} \in NL$. On input w, guess which transitions in NFA to follow to get to accept. Store single pointer to keep track of state. Important detail I keep on forgetting: need a counter to keep track of steps to kill computation and ensure halting. In this case, at most $|N|$ unique states to visit; so if not halt by then must be looping.

To show NL-hard, reduce from PATH. Pretty simple mapping, but one important detail: send nodes of G to states in NFA. Directed edges are **ϵ-transitions**. $s \mapsto q_{start}$, $t \mapsto q_{accept}$. So $\langle G, s, t\rangle \mapsto \langle N, \epsilon\rangle$. We need an ϵ, since we can't really predetermine what input the length should be (how long a path is). □

9 Intractability

9.1 Key Definitions

Definition 9.1. An **intractable** problem is one that can't be solved practically due to excessive time or space requirements.

Definition 9.2. A **space constructible** function is a function/TM $f(n) \quad O(\log n)$ that takes in 1^n as input and outputs the binary representation of $f(n)$ using only $O(f(n))$ space. Ex) $f(n) = k$ for constant k is not space constructible, as the computation requires deleting 1^n in $O(n)$ time before writing the binary representation $\log k$. To account for $f(n) < O(\log n)$ we use the same "read-only" tape idea as the log-space transducer.

Definition 9.3. Function $t(n) \quad O(n \log n)$ is time constructible if t maps 1^n to the binary representation of $t(n)$ in $O(t(n))$ time.

Definition 9.4. An **oracle** for language A reports whether w 2 A in one step.

Definition 9.5. An **oracle TM** M^A is a TM that can query an oracle for A via an oracle tape. P^A is class of languages decidable in poly-time with a TM with oracle A. NP^A is class of languages decidable in non-deterministic poly-time with a TM with oracle A. *Remark*: NP^A means non-determnistically pick, then check with P^A.

9.2 Key Results

Theorem 9.6. $L \quad NL \quad P \quad NP \quad PSPACE \quad EXPTIME \quad EXPSPACE$

Theorem 9.7 (Space hierarchy)**.** *For any space constructible f, there exists language A that is decidable in $O(f(n))$ space but not $o(f(n))$ space. I.e. some language* ***requires*** *at least $f(n)$ space to be decided.*

Proof. Proceed by diagonalization. Basically, we want to describe a language by constructing an associated TM that does the exact "opposite" of all "smaller"-space TMs. This algorithm runs TMs on descriptions of TMs, doing the opposite of individual TMs that run in $o(f(n))$ space (no requirement to be different from TMs that run in more than $f(n)$ space), and rejecting otherwise.

Consider the following algorithm that decides A: Let $D =$ "On input w:

1. Let n be the length of w.
2. Compute $f(n)$ in $O(f(n))$ space (constructability). Mark off $f(n)$ cells – this is the maximum space that any simulated TM can use. *Reject* if more space is ever used.
3. Check if w is in the form $hM i10$ for some M. The trailing 0's are to allow asymptotic behavior to "kick in" for large enough n (D might run in more than $f(n)$ space for small n and miss an opportunity to contradict M running in $o(f(n))$ space). If not in this form *reject*.
4. Simulate M on w and count the number of steps used in simulation. D might loop, so we cap the steps at $2^{f(n)}$ since there are max $f(n)$ cells to use. Exceed cap =) loop =) *reject*.
5. If M accepts, *reject*. If M rejects, *accept*.

D is obviously a decider. It runs in $f(n) = O(f(n))$ space, so A decidable in $O(f(n))$ space. ASOC some M decides A in $o(f(n))$ space. Then on sufficiently long input $hM i10^{n_0}$, D runs in $f(n)$ space and does the opposite of M, so A cannot be decided by M.

□

Theorem 9.8. *$f_1(n)$ is $o(f_2(n))$ and f_2 is space constructible =) $SPACE(f_1(n)) \quad SPACE(f_2(n))$*

Proof. Immediate corollary from Space Hierarchy – if some languages absolutely require $O(f(n))$ space, then the languages decidable in this space is larger than a smaller space. □

Theorem 9.9. $0 \leq \epsilon_1 < \epsilon_2 \implies SPACE(n^{\epsilon_1}) \subsetneq SPACE(n^{\epsilon_2})$

Theorem 9.10. $NL \subsetneq PSPACE$. *Remark: this implies $TQBF \notin NL$ (since $TQBF$ is PSPACE-complete w.r.t. log-space reduction, then $TQBF \in NL \implies$ all problems in PSPACE log-space reducible to problem in $NL \implies PSPACE \subseteq NL \implies NL = PSPACE$.*

Proof. $NL = NSPACE(\log n) \subseteq SPACE(\log^2 n)$ by Savitch. Then by Space Hierarchy, $SPACE(\log^2 n) \subsetneq SPACE(n^k) \subseteq PSPACE$. □

Theorem 9.11. $PSPACE \subsetneq EXPSPACE$

Theorem 9.12. ***Time Hierarchy****: For any time constructible $t(n)$, there exists language A requiring $O(t(n))$ time to be decided (not decidable in $o(t(n)/\log t(n))$ time).*

Proof. Remark: note the weaker bound. This is because simulating M on $\langle M \rangle$ requires a logarithmic increase in time, rather than a constant increase in space (just use multiple D tape symbols to represent a larger alphabet of M).

For the actual proof, consider the following $O(t(n))$ time D deciding A:

1. Let n be the length of w.
2. Compute $t(n)$ (constructible in $O(t(n))$) and store binary representation of the counter $t(n)/\log t(n)$. Decrement counter for each simulation step of M on w. If counter hits 0, M has used $t(n)/\log t(n)$ time, meaning D has taken $t(n)$ time, so *reject*.
3. If w is not in form $\langle M \rangle 10^*$, *reject*.
4. Simulate M on w.
5. Do opposite of M.

The details of the log increase in simulation time for step 4 are omitted for the sake of sanity. □

Theorem 9.13. *$t_1(n)$ is $o(t_2(n)/\log t_2(n))$ and t_2 is time constructible $\implies TIME(t_1(n)) \subsetneq TIME(t_2(n))$.*

Theorem 9.14. $1 \leq \epsilon_1 < \epsilon_2 \implies TIME(n^{\epsilon_1}) \subsetneq TIME(n^{\epsilon_2})$.

Theorem 9.15. $P \subsetneq EXPTIME$

Theorem 9.16. *Let $EQ_{REX\uparrow} = \{\langle Q, R \rangle | Q$ and R are equivalent regular expressions with exponentiation$\}$. Then we have that $EQ_{REX\uparrow}$ is EXPSPACE-Complete.*

Proof. Reductions by computation histories (see page 220 in Section 5.1 for review).

First we show $EQ_{REX\uparrow} \in EXPSPACE$. Writing out the regular expressions as concatenations instead of exponentiation gives us exponential-length inputs. Converting regex to NFAs increases size linearly. Then test *in*-equivalence of NFA using NTM. To do so, non-deterministically pick one-by-one an input symbol to read for $2^{q_1} \cdot 2^{q_2} = 2^{q_1+q_2}$ steps (all possible subsets of states, more steps would guarantee a repeat of "state of states"). This takes linear time in length of input (recall encoding NFA encodes exponential transition function possibilities). A deterministic version takes n^2 time (Savitch), where n is exponential.

Now, we need to show $EQ_{REX\uparrow}$ is *EXPSPACE-Hard* (all languages poly-reduce to it). We basically map $w \in A$ to regex R_1 and R_2. $R_1 = \Delta^*$ where $\Delta = \Gamma \cup Q \cup \#$. We construct R_2 to be all the computation histories that do not lead to a reject on input w. Clearly $w \in A \iff R_1 = R_2 \iff \langle R_1, R_2 \rangle \in EQ_{REX\uparrow}$.

Let $R_2 = R_{bad\ start} \cup R_{bad\ reject} \cup R_{bad\ window}$. We describe each regex qualitatively:

$R_{bad\ start} = S_0 \cup \cdots \cup S_n \cup S_b \cup S_\#$. Each regex S_i generates strings *not* including the ith appropriate symbol of the starting config at the ith location. Special case for S_b, since encompasses all missed trailing blank locations (location $n+2$ to $2^{(n^k)}$, could be expo). Instead: $S_b = \Delta^{n+1}(\Delta \cup \epsilon)^{2^{(n^k)}-n-2}\Delta_{-\sqcup}\Delta^*$.

Then $R_{bad \quad reject} = \Delta \quad q_{reject}$ (straightforward).

Similar to Cook-Levin proof, let $R_{bad \quad window} = \bigcup_{\text{invalid}(abc, def)} \Delta \; abc\Delta^{(2^{n^k}) \quad 2} def \Delta$. This is the same 2 3 invalid window approach we've seen before. Note the $(2^{n^k}) \quad 2$ difference: this is the distance from c to d one configuration away (c to f is exactly 2^{n^k}, so subtract 2). ☐

Theorem 9.17. *There exists oracle A such that $P^A \neq NP^A$. Remark: this suggests we cannot solve $P = NP$ via, as that would imply $P^A = NP^A$ for all A.*

Proof. Consider language $L_A = \{w \mid \exists x \in A[|x| = |w|]\}$ for any oracle A. $L_A \in NP^A$ (to check if $w \in L_A$, guess the right x and check if $x \in A$ using oracle for A). We construct a *particular* A.

Consider $M_1, M_2 \ldots$ running in n^i time. At each i, we construct A so that M_i^A cannot decide L_A. At stage i, pick n that is greater than length of any string currently in A, *and* such that $2^n > n^i$.

Run M_i

☐

Theorem 9.18. *There exists oracle B such that $P^B = NP^B$. Remark: this suggests we cannot solve $P \neq NP$ via, as that would imply $P^B \neq NP^B$ for all B.*

Proof. Consider any *PSPACE-Complete* problem, like $TQBF$. Then $NP^B \quad NPSPACE \quad PSPACE$ P^B. ☐

9.3 Proof Concepts and Examples

Example 9.19. $NP \quad P^{SAT}$ since all problems in NP reduce to SAT with some poly-time reduction, then we can check in one step if in SAT. It follows that $NP \quad coP^{SAT} \implies coNP \quad P^{SAT}$.

Example 9.20. It is unclear if $\overline{\textit{MIN-FORMULA}} \in NP$ (guess smaller formula, but would need to verify truthiness across potentially exponential inputs). However, we know $\overline{\textit{MIN-FORMULA}} \in NP^{SAT}$. First, we can decide in-equivalence of ϕ in NP (guess the right assignment), so equivalence is decidable in $coNP$. To decide *MIN-FORMULA*, guess the right smaller ϕ' then easily verify if $\phi' = \phi$ using P^{SAT} since $coNP \quad P^{SAT}$.

9.4 Problem Set Results

10 Advanced Topics in Complexity Theory

10.1 Key Definitions

Definition 10.1. A **probabilistic TM** M is a nondeterministic TM where each nondeterministic step is a **coin flip step** with two equally legal moves. The probability of following any branch of computation b is $\Pr[b] = 2^{-k}$, with k being the number of coin-flip steps on the branch. We define the probability of PTM M accepting w as $\Pr[M \text{ accepts } w] = \Pr_{b \text{ accepting}}[b]$

Definition 10.2. A PTM decider M need not be correct *all the time*. Indeed, we say M decides A with error probability ϵ if the decider is wrong with probability ϵ, i.e.:

1. $w \in A \implies \Pr[M \text{ accepts } w] \geq 1 - \epsilon$
2. $w \notin A \implies \Pr[M \text{ rejects } w] \geq 1 - \epsilon$

Definition 10.3. BPP is the class of languages decidable by a probabilistic poly-time TM with $\epsilon = 1/3$ (sufficient by **amplification lemma**).

Definition 10.4. A **branching program** is a directed acylic graph that has **query nodes** labeled x_i having two outgoing edges labeled 0 or 1, two **output nodes** labeled 0 *and* 1 without any outgoing edges. One node is designated the start node. *Remark:* a BP describes a boolean function $f : \{0,1\}^m \to \{0,1\}$ (following the BP path using assignment will lead you to a 0 or 1).

. . . .

10.2 Key Results

Theorem 10.5. ***Polynomial Lemma:*** *If $p(x) \neq 0$ is polynomial of degree $\leq d$ then p has $\leq d$ roots.*

Theorem 10.6. *If $p_1(x)$ and $p_2(x)$ both degree $\leq d$ and $p_1 \neq p_2$, then $p_1(z) = p_2(z)$ for $\leq d$ values z.*

Proof. Set $p = p_1 - p_2$; then $p_1(z) - p_2(z) = 0$ for $\leq d$ values by Polynomial Lemma.

Remark: the above holds for any*any* field; in particular, we choose finite field $\mathbb{F}_q$ with q elements. □

Theorem 10.7. *If $p(x) \neq 0$ and has degree $\leq d$, then $\Pr[p(r) = 0] \leq d/q$; treat 0 as polynomial and invoke above corollary; d matches out of at most q elements.*

Theorem 10.8. ***Schwartz-Zippel:*** *If $p(x_1 \ldots x_m) \neq 0$ has degree $\leq d$ in **each** x_i, and we pick random $r_1, \ldots, r_m \in \mathbb{F}_q$, then $\Pr[p(r_1, \ldots, r_m) = 0] \leq md/q$. This is a generalization of the previous corollary.*

Theorem 10.9. $EQ_{ROBP} \in BPP$

Recall that $EQ_{ROBP} = \{\langle B_1, B_2 \rangle \mid B_1$ and B_2 are equiavlent ROBP$\}$.

Proof. On input $\langle B_1, B_2 \rangle$ [on input/nodes $x_1 \ldots x_m$]:

1. Pick a prime $q \geq 3m$.
2. Choose assignment $r_1 \ldots r_m$.
3. Evaluate B_1, B_2 on assignment. *Accept* if both equal, *reject* if else.

If $B_1 = B_2$, then they are TRUE on the same assignments, hence their truth table is the same. Thus the polynomials encoding their boolean behavior are the same, and thus are equal even for non-boolean inputs. So accept with 100% correctness.

If $B_1 \neq B_2$, then $p_1 \neq p_2$ and the polynomials *should ideally* be unequal on given non-boolean input. They *may* get lucky and be equal, but by Schwartz-Zippel this only happens with probability $\leq md/q \leq 1/3$.

□

Theorem 10.10. $coNP \quad IP$

Proof. Show $\#SAT\ 2\ IP$. Note $\overline{\#SAT}$ NP-complete since $\overline{SAT}\quad_p \overline{\#SAT}$ (ϕ 7 $h\phi, 0i$). □

10.3 Problem Set Results

Example 10.11. Review probelm 6

11 Class Relations, Closure Properties, and Useful Problems

Theorem 11.1. $L \quad NL \quad P \quad NP \quad PSPACE \quad EXPTIME \quad EXPSPACE$

Theorem 11.2. $NL \subsetneq PSPACE$ *by Space Hierarchy (strict containment)*

Theorem 11.3. $PSPACE \subsetneq EXPSPACE$ *by Time Hierarchy (strict containment)*

Theorem 11.4. $IP = PSPACE$.

Theorem 11.5. $coNP \quad IP$

Theorem 11.6. $P \quad BPP$

Theorem 11.7. $BPP \quad PSPACE$

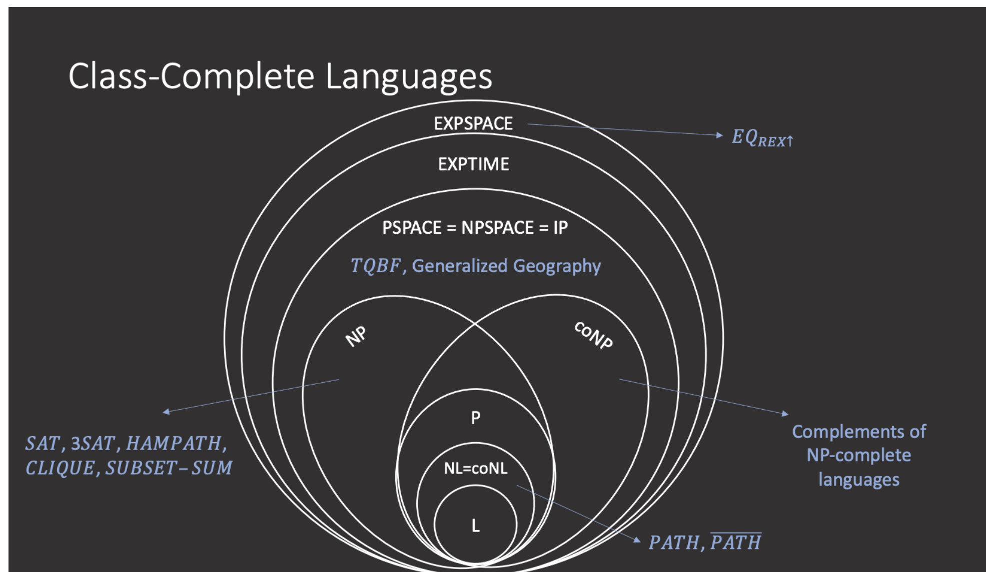
Theorem 11.8. *BPP = coBPP*

Theorem 11.9. *Useful closure properties:*

Class	∪	∘	*	∩	$\overline{\cdot}$	$\cdot^R$ (reversal)	\
L	Yes	Yes	?	Yes	Yes	Yes	Yes
NL=coNL	Yes	Yes	Yes	Yes	Yes	Yes	Yes
P	Yes	Yes	Yes	Yes	Yes	Yes	Yes
NP	Yes	Yes	Yes	Yes	?	Yes	?
coNP	Yes	Yes	Yes	Yes	?	Yes	?
BPP	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IP=PSPACE=NPSPACE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
EXP/EXPTIME	Yes	Yes	Yes	Yes	Yes	Yes	Yes
EXPSPACE	Yes	Yes	Yes	Yes	Yes	Yes	Yes

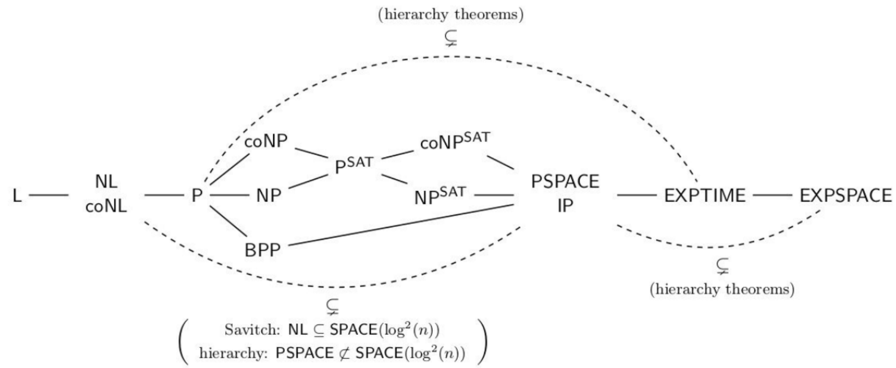
P closed under concatenation since it can guess $O(n)$ splits and run $O(n^k)$ TM on each substring, so overall polynomial. For closure under Kleene star, use DP for an $O(n^3)$ algorithm (check all substrings of $1 \ldots n$) length. P^{SAT} has identical closure properties (follow same algorithms as P, but can query SAT in $O(1)$ time).

Theorem 11.10. *Class Complete Languages:*



Theorem 11.11. *Known and Unknown Class Containments:*

Complexity Class Containments



Fill lines (A—B) represent A ⊆ B, but B ⊆ A is unknown
Dotted lines (A---B) represent A ⊊ B (A is a strict subset of B)

Theorem 11.12. *Common Decidable/Undecidable Languages:*

Review: Common Languages

T-Decidable	T-Recognizable (undecidable)	T-coRecognizable (undecidable)	T-Unrecognizable (neither T-recog nor T-coRecog)
• A_{DFA} • A_{NFA} • E_{DFA} • EQ_{DFA} • A_{CFG} • E_{CFG}	• A_{TM} • negate(EQ_{CFG}) • $HALT_{TM}$	• E_{TM} • negate(A_{TM}) • EQ_{CFG}	• EQ_{TM} • negate(EQ_{TM})

Index

accepts, 3
alphabet, 3
ambiguous, 5
amplification lemma, 34

BPP, 34
branching program, 34

Chomsky normal form, 5
Church-Turing thesis, 8
coin flip step, 34
Concatenation, 3
configuration, 7
configuration graph, 26
context-free grammar (CFG), 5
context-free language (CFL), 5

decidable, 7
derivation, 5
deterministic context-free language (DCFL), 5
deterministic pushdown automaton (DPDA), 5
deterministic queue automaton (DQA), 8

empty language, 3
ensure halting by storing a counter, 28
enumerator, 7
equivalent, 3

finite automaton, 3

generalized nondeterministic finite automaton (GNFA), 3

intractable, 31

language, 3
loop, 7

multitape Turing machine, 7

nondeterministic finite automaton (NFA), 3
NP, 13

oracle, 31
oracle TM, 31
output nodes, 34

parse tree, 5
Polynomial Lemma:, 34
popping, 5
probabilistic TM, 34
PSPACE-complete, 22
pull, 8
pumping lemma, 4
push, 8
pushdown automaton (PDA), 5
pushing, 5

query nodes, 34
queue, 8

recognizes, 3
regular expression, 3
regular language, 3
rules, 5

Schwartz-Zippel:, 34
space constructible, 31
space hierarchy, 31
stack, 5
Star, 3
states, 3
string order, 8

terminals, 5
transition function, 3
Turing machine (TM), 7
Turing-decidable, 7
Turing-recognizable, 7

Union, 3

variables, 5