

🔗 Automata, Computability, & Complexity

Author: Guanzhou (Jose) Hu 胡冠洲 @ MIT 6.045

Teachers: [Ryan Williams](#)

Automata, Computability, & Complexity

Theory of Computation Overview

Automata

Formal Definition of Language

Deterministic Finite Automata (DFA) & Regularity

Non-deterministic Finite Automata (NFA)

Regular Expressions (REs) & Generalized NFA (GNFA)

Non-Regular Languages

DFA Minimization

The Myhill-Nerode Theorem

Streaming Algorithms

Computability

Turing Machines (TM), Recognizability, & Decidability

Universal TM & The Church-Turing Thesis

The Halting Problem & Mapping Reduction

Oracle TMs & Turing Reduction

Self-Reference & Recursion

Formal Systems

Complexity

Communication Complexity

Computation Time Complexity

Nondeterminism & P vs. NP

NP-Completeness & The Cook-Levin Theorem

CoNP & Oracle Complexity

Space Complexity

Randomized Complexity

Theory of Computation Overview

Main questions in theory of computation:

- What is computation? \Rightarrow Automata & Turing Machines as the mathematical model
- What can and cannot be computed? \Rightarrow Computability
- What can be efficiently computed? \Rightarrow Complexity

Mathematical proof should provide 3 levels:

1. Short phrase giving "hints"
2. One paragraph description of main ideas
3. Full proof (and nothing else)

Automata

See my "Computer Languages & Compilers" course note. The following are something more formal.

Formal Definition of Language

An **Alphabet** Σ is a finite set of characters:

- A **String** s over Σ is a finite sequence of characters $\in \Sigma^*$
 - $|x| =$ length of string x

- The unique string of length 0 (*empty string*) denoted by ε
- Σ^* is the set of all strings over Σ

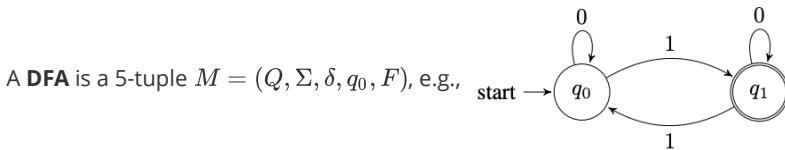
A **Language** over Σ is a set of strings over Σ , i.e., a subset of Σ^* :

- Sometimes, we think of a language L as a *function* $f : \Sigma^* \mapsto \{0, 1\}$ (accepting a string or not?)
- Sometimes, we use $L(A)$ to denote set of all strings that an automaton A accepts

Operations on languages:

Notation	Meaning	Notes
\emptyset	Empty language	$\neq \{\varepsilon\}$
$L_1 \cup L_2$	Union	$\{w \mid w \in L_1 \vee w \in L_2\}$
$L_1 \cap L_2$	Intersection	$\{w \mid w \in L_1 \wedge w \in L_2\}$
$\neg L$ or \bar{L}	Complement	$\{w \in \Sigma^* \mid w \notin L\}$
L^R	Reverse	$\{w_1 \dots w_k \mid w_k \dots w_1 \in L, w_i \in \Sigma\}$
$L_1 L_2$	Concatenation	$\{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
L^*	0 or more self-concatenation (<i>Kleen star/closure</i>)	$\{w_1 \dots w_k \mid \text{each } w_i \in \Sigma \wedge k \geq 0\}$
L^+	1 or more self-concatenation	$\{w_1 \dots w_k \mid \text{each } w_i \in \Sigma \wedge k > 0\}$

Deterministic Finite Automata (DFA) & Regularity



- Composed of:
 - Q : the finite set of *states*
 - Σ : the finite alphabet
 - $\delta : Q \times \Sigma \mapsto Q$ is the *transition function*
 - $q_0 \in Q$ is the *start state*
 - $F \subseteq Q$ is a finite set of *accept/final states*
- M accepts a string $w = w_1 \dots w_n$ if there is a sequence $r_0, r_1, \dots, r_n \in Q$, s.t.: (o.w. called *reject*)
 - $r_0 = q_0$, and
 - $\delta(r_{i-1}, w_i) = r_i$ for all $i = 1, \dots, n$, and
 - $r_n \in F$

A language L' is **regular** iff L' is recognized by a DFA, that is, there is a DFA M where $L' = L(M)$.

Regular Languages (RLs) are *closed* under the following cases:

1. *Union Theorem*: Union of two RLs $L_1 \cup L_2$ is still regular; rule to construct DFA:

- $Q = \{(q_1, q_2) \mid q_1 \in Q_1 \wedge q_2 \in Q_2\} = Q_1 \times Q_2$, i.e., composite pairs of states
- $q_0 = (q_0^{(1)}, q_0^{(2)})$
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \vee q_2 \in F_2\}$
- $\delta((q_1, q_2), c) = (\delta_1(q_1, c), \delta_2(q_2, c))$

2. *Intersection Theorem*: Intersection of two RLs $L_1 \cap L_2$ is still regular; rules:

- same as unioned DFA, except that
- $F = \{(q_1, q_2) \mid q_1 \in F_1 \wedge q_2 \in F_2\}$

3. *Complement Theorem*: Complement of an RL $\neg L$ is still regular; rules:

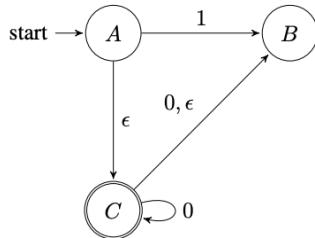
- everything the same as L , except that
- $F = \{q \in Q \mid q \notin F\}$, i.e., flip final & non-final states

4. *Reverse Theorem*: Reverse of an RL L^R is still regular; rules are not intuitive, we should first introduce NFAs

5. Similar for *Concatenation, Prefix, & Wrap*.

Non-deterministic Finite Automata (NFA)

An **NFA** is a 5-tuple $N = (Q, \Sigma, \delta, Q_0, F)$, e.g.,



- Composed of:
 - Q : the finite set of *states*
 - Σ : the finite alphabet
 - $\delta: Q \times \{\Sigma \cup \epsilon\} \mapsto 2^Q$ is the *transition function*, where 2^Q means the set of all possible subsets of Q
 - $Q_0 \subseteq Q$ is a set of *start states*
 - $F \subseteq Q$ is a finite set of *accept/final states*
- We allow multiple start states here, but normally the definition is only one; conversion is easy: add an extra state which stretches ϵ -transitions to all the start states
- M accepts a string $w = w_1 \dots w_n$ if there is a sequence $r_0, r_1, \dots, r_n \in Q$, s.t.: (o.w. called *reject*)
 - $r_0 \in Q_0$, and
 - $r_i \in \delta(r_{i-1}, w_i)$ for all $i = 1, \dots, n$, and
 - $r_n \in F$

For every NFA N , there is a DFA M s.t. $L(M) = L(N)$:

- Idea: *Subset Construction*, setting $Q' = 2^Q$; **see my PL note - "Convert NFA → DFA"**
- This means a language L' is regular iff L' is recognized by an NFA!
- This means using NFAs in place of DFAs can make proofs about RLs much easier!

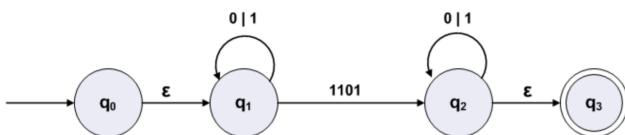
Constructing NFA over language operations: *Thompson's construction*; **see my PL note - "From RE → NFA"**.

Regular Expressions (REs) & Generalized NFA (GNFA)

First, **check my PL note - "Regular Expressions"**.

Expressiveness: DFA \equiv NFA \equiv RE. They all express regular languages. Proof is two fold, where RE \Rightarrow regular is easy. Regular \Rightarrow RE introduces the concept of GNFA.

A Generalized NFA (**GNFA**) can read in entire substrings on one edge (labeled by an RE), instead of only a single character, e.g.,



- Every NFA is also a GNFA, and
- Every RE represents a GNFA (with only two states)

Converting from an NFA N into a GNFA G with only two states, then we can prove that $L(N)$ (which is regular) is equivalent to the language represented by the RE on that only edge of G . Conversion goes as the following:

```

1 // Converting NFA to a two-states GNFA which is essentially an RE.
2 G convert(N) {
3     if (#states == 2) return N;
4     else {
5         pick state q_rip which is not q_start or q_acc;
6         for (every pair of states (q_i, q_j) that originally has path through q_rip) // q_i can ==
7             q_j.
8             R(q_i, q_j) += R(q_i, q_rip) R(q_rip, q_rip)^* R(q_rip, q_j);
9         results in a 1-state fewer GNFA G';
10        return convert(G');
11    }

```

Non-Regular Languages

So far, our topics cover:

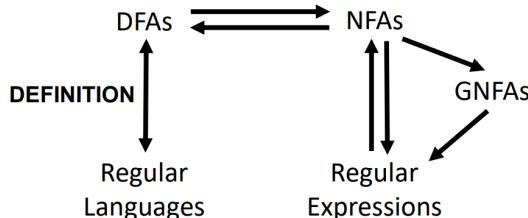


Figure from [Ryan's slides](#).

There are many languages that are **non-regular** (Cannot find a DFA for it). A typical example is $L = \{0^n 1^n \mid n \geq 0\}$. The big idea here is that: A DFA cannot remember how many zeros it has read in, if there are more zeros than its # states. Its # states is finite!

More examples: $\{0^n 1^n \mid n \geq 0\}$, $\{w \mid w \text{ has equal number of 0s and 1s}\}$, $\{w \mid w = w^R\}$ (*palindromes*),

Proof method - The *confusion* technique: Assume there is a DFA M , and # states = Q . Carefully choose string x , where $|x| > Q$. Use *Pigeonhole Principle* to show that some state q must be visited at least twice. Then, M is in state q after reading x' or x'' , where x' and x'' are two different prefixes of x . Then pick a string y s.t. exactly one $x'y$ or $x''y$ is in $L \Rightarrow$ *Contradiction!*

DFA Minimization

For every regular language L , there is a *unique & minimal* DFA M (minimal here means fewest number of states; unique here means up to re-labeling states) s.t. $L(M) = L$.

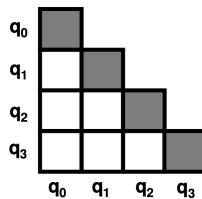
We introduce the following terms:

- For DFA $M = (Q, \Sigma, \delta, q_0, F)$, denote $M_q = (Q, \Sigma, \delta, q, F)$, $q \in Q$ which is the sub-DFA starting from q
- String $w \in \Sigma^*$ **distinguishes** states p and q iff: M_p accepts $w \Leftrightarrow M_q$ rejects w (i.e., leading to one accepting \wedge the other rejecting)
 - If \exists such w (including ϵ !), then $p \not\sim q$ (*distinguishable* from)
 - If $\forall w \in \Sigma^*$, w cannot distinguish p from q , then $p \sim q$ (*indistinguishable* from)

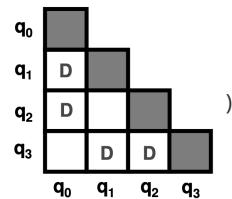
Easy to prove that *indistinguishability* relation is an *equivalence relation*. This means we can partition Q into several *equivalence classes*, where each of them $[q] = \{p \mid p \sim q\}$. Big idea of minimizing DFA is to find such partition into equivalence classes, and shrinking each of them down to one state. Output M_{min} has the following three properties:

1. $L(M) = L(M_{min})$
2. M_{min} has no *inaccessible* states (i.e., unreachable from initial state)
3. M_{min} is *irreducible* (i.e., every pair of states is distinguishable)

Minimization algorithm: first prepare a table



; (after the algorithm it looks like



```

1 // Minimizing a DFA M.
2 M_min minimize(M) {
3     Run a graph search from initial state  $q_0$  to remove all inaccessible states from M;
4     // Running the Table Filling algorithm...
5     for (all (p, q) s.t. exactly one of them is an accepting state)
6         Mark 'D' at entry (p, q);    // Base case: distinguishable by  $\epsilon$ .
7     while (there is update in this iteration) {      // Iter: infer pairs distinguishable with
8         longer strings.
9         if (exists (p, q) and any symbol a in alphabet s.t.  $\delta(p, a) = p_2$ ,  $\delta(q, a) = q_2$ , and
10            we already marked 'D' at entry ( $p_2, q_2$ ))
11            Mark 'D' at entry (p, q);
12     }
13     Get EQUIV set = {[q] | q in Q} by those unmarked pairs;
14     return M_min = (EQUIV,  $\Sigma$ ,  $\delta_{\min}([q], a) \rightarrow [\delta(q, a)]$ , [ $q_0$ ], {[q] | q in F});
15 }
```

Detailed proof about correctness & uniqueness can be found in Lec5's slides.

The Myhill-Nerode Theorem

Let us define an equivalence relation which is purely on the language itself and not assuming any DFAs: Let $L \subseteq \Sigma^*$ and $x, y \in \Sigma^*$, $x \equiv_L y$ means that $\forall z \in \Sigma^*, xz \in L \Leftrightarrow yz \in L$. We call x and y are *indistinguishable to L* (or L -equivalent).

The **Myhill-Nerode Theorem** states that, a language L is regular \Leftrightarrow the number of equivalence classes of \equiv_L is finite.

- Proof is two-fold, by proving both directions; See Lec6's slides.
- Application: a new way of proving a given language is not regular - give an infinite set of strings S in L (called a *distinguishing set*) s.t. $\forall w_i \neq w_j \in S, w_i$ and w_j are *distinguishable to L* .

Streaming Algorithms

Streaming algorithms are an extension to finite automata. They have 3 components:

1. *Initialize*: a bunch of *variables* and their initial assignments
2. *Next symbol action*: pseudocode to operate on variables when seeing the next incoming symbol σ
3. *Stop*: accept/reject condition when the stream stops

They are very similar to DFAs except that their *memory usage* can increase with the string length. This gives them the ability to recognize non-regular languages!

Example - streaming algorithm that recognizes $L = \{x \mid x \text{ has more 1s than 0s}\}$ over alphabet $\{0, 1\}$:

```

1 // Initialize:
2 C = 0, B = 0;
3
4 // When next symbol is 'a':
5 if (C == 0) {
6     B = a;
7     C = 1;
8 } else if (C != 0 && B == a)
9     C++;
10 else if (C != 0 && B != a)
11     C--;
12
13 // When stream stops:
14 if (B == 1 && C > 0)
```

```

15     ACCEPT;
16 else
17     REJECT;

```

When using binary representation to store variables B and C , space usage of this algorithm is $\log_2 n + O(1)$.

Proving lower bounds of memory usage of a streaming algorithm:

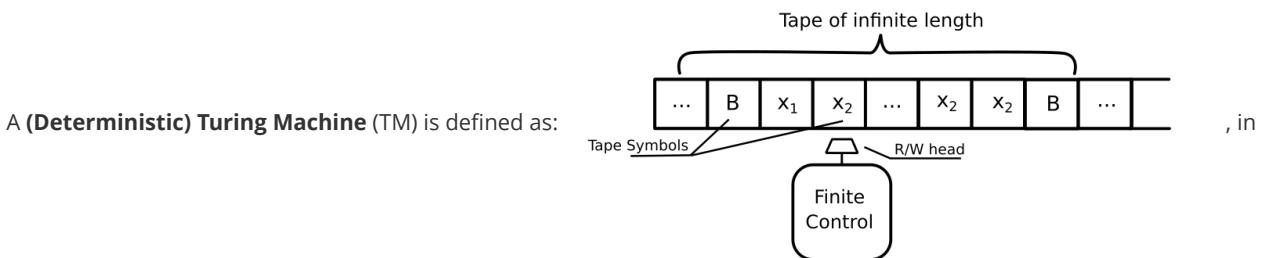
- If language L is computable by a streaming algorithm with space usage $\leq S(n)$, then $\forall n \in \mathbb{N}$, there is a DFA M with $< 2^{S(n)+1}$ states s.t. $L_n = L(M)_n$, i.e., $\{w \in L \mid |w| \leq n\} = \{w \in L(M) \mid |w| \leq n\}$ (proof - see Lec7's slide);
- A *streaming distinguisher* D_n for L_n is a subset of Σ^* s.t. $\forall x, y \in D_n$, there is a string $z \in \Sigma^*$ s.t. $|xz|, |yz| \leq n$ and z distinguishes x, y ;
- *Streaming Theorem*: Suppose $\forall n \in \mathbb{N}$, there is a streaming distinguisher D_n with $|D_n| \geq 2^{S(n)}$, then all streaming algorithms that recognize L must use at least $S(n)$ space.

For 2-pass streaming algorithm solution to FREQUENT ITEMS problem, see Lec7 slides.

Computability

Turing Machines (TM), Recognizability, & Decidability

Introduced by Alan Turing in 1936.



each step:

1. Reads a symbol
2. Writes a symbol
3. Finite control changes state
4. Moves left / right a block

A formal definition would be a 7-tuple $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$,

- Composed of:
 - Q : the finite set of states
 - Σ : the finite input alphabet
 - Γ : the finite tape alphabet, where \square ("blank" symbol) $\in \Gamma$ and $\Sigma \subset \Gamma$
 - $\delta: Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$, where L, R means moving left / right
 - $q_0 \in Q$ is the start state
 - $q_{acc} \in Q$ is the accepting state
 - $q_{rej} \in Q$ is the rejecting state, and $q_{acc} \neq q_{rej}$
- Initially, input string is written on tape and the control *head* points to the first symbol
- A TM configuration $\in (Q \cup \Gamma)^*$ is the stuff on tape with current state inserted right before control head
 - Configuration C_1 yields C_2 if T goes to C_2 after running for one step in C_1
 - T accepts a string w if there is a sequence of configurations s.t.: (reject is similar)
 - $C_0 = q_0 w$, and
 - C_{i-1} yields C_i for all $i = 1, \dots, n$, and
 - C_n contains the accept state q_{acc}
- A TM can have 3 kinds of behaviors on an input string:
 1. Accept
 2. Reject

3. Running forever w/o halting

Recognizability is defined as the following:

- A TM T recognizes L iff T accepts exactly the strings in L (but for other strings it might run forever)
- A language L is *recognizable* (i.e., *recursively enumerable*) if some T recognizes L

Decidability is defined as the following: (this is stronger than recognizability)

- A TM T decides L iff T accepts all strings in L , and rejects all strings not in L
- A language L is *decidable* (i.e., *recursive*) if some T decides L

Universal TM & The Church-Turing Thesis

We can encode a TM into a *bit string* $b = w_T w_{\delta_0} w_{\delta_1} \dots$, where (DFA / NFA are similar)

- $w_T = 0^n 10^m 10^k 10^s 10^t 10^r 10^u 1$ encodes the machine metadata
 - $n = \# \text{ states}$
 - $m = \# \text{ tape symbols}$ (we assume first k are input symbols)
 - $k = \# \text{ input symbols}$
 - $s = \text{index of the start state}$
 - $t = \text{index of the accepting state}$
 - $r = \text{index of the rejecting state}$
 - $u = \text{index of the blank symbol}$
- Each $w_\delta = 0^p 10^i 10^q 10^j 1(0 + 00)1$ encodes a function map in $\delta : p, i \mapsto q, j, L/R$

We then encode a pair of bit strings $\langle b_1, b_2 \rangle$ as $0^{|b_1|} 1 b_1 b_2$. Then, any DFA / NFA / TM is just a language over $\Sigma = \{0, 1\}^*$!

- $A_{\text{DFA}} = \{\langle M, w \rangle \mid M \text{ encodes a DFA and } M \text{ accepts } w \in \Sigma^*\}$
- $A_{\text{NFA}} = \{\langle N, w \rangle \mid N \text{ encodes a NFA and } N \text{ accepts } w \in \Sigma^*\}$
- $A_{\text{TM}} = \{\langle T, w \rangle \mid T \text{ encodes a TM and } T \text{ accepts } w \in \Sigma^*\}$

A **Universal TM** U can simulate any other TMs, i.e., U accepts $\langle T, w \rangle \Leftrightarrow T$ accepts w . In other words, there is a TM that can run any other arbitrary TM code! Proof is in Lec9 slides.

- Since A_{DFA} is not regular, there isn't a DFA which can simulate any other DFA
- A_{DFA} and A_{NFA} are both decidable; A_{TM} is recognizable but undecidable

The **Church-Turing Thesis** states that "Any function on natural numbers can be calculated by an *effective method* (i.e., has an *algorithm*) \Leftrightarrow it is computable by a TM". The Church-Turing Thesis implies that there are *unrecognizable* languages! We can prove by showing that there is no *onto* function from the set of all TMs ($\subseteq \{0, 1\}^*$) to the set of all languages over $\{0, 1\}$ (powerset of $\{0, 1\}^*$, i.e., $2^{\{0,1\}^*}$). In other words, there are more problems to solve than there are programs to solve them.

The Halting Problem & Mapping Reduction

Example of a *recognizable* but *undecidable* problem:

- Consider the language $A_{\text{TM}} = \{\langle T, w \rangle \mid T \text{ encodes a TM and } T \text{ accepts } w \in \Sigma^*\}$. This is equivalent to the problem: given code of a TM T and an input w , does T accept w ? This problem is recognizable but undecidable.
- Proof idea: *diagnolization* - suppose there is a machine U that decides A_{TM} , define a TM $D_U(T)$ that runs U on $\langle T, T \rangle$ and outputs the opposite answer. Now consider the input $\langle D_U \rangle$ for D_U : $D_U(\langle D_U \rangle)$ accepts $\Leftrightarrow U(\langle D_U, D_U \rangle)$ rejects $\Leftrightarrow D_U(\langle D_U \rangle)$ rejects. Contradiction!

Example of an *unrecognizable* problem:

- Consider the language $\neg A_{\text{TM}} = \{\langle T, w \rangle \mid T \text{ encodes a TM and } T \text{ does not accept } w \in \Sigma^*\}$ (w can be empty). This is equivalent to the "non-acceptance" problem of TMs.
- Lemma: L is decidable $\Leftrightarrow \neg L$ are both recognizable.
- Then, $\neg A_{\text{TM}}$ is not recognizable, otherwise both A_{TM} and $\neg A_{\text{TM}}$ would be decidable.

The **Halting Problem** states that:

- Consider the language $\text{HALT}_{\text{TM}} = \{\langle T, w \rangle \mid T \text{ encodes a TM and } T \text{ halts on } w \in \Sigma^*\}$. This language is undecidable. In other words, there is no TM that can decide whether a TM halts on an input.
- Proof idea: suppose there is a TM U that decides HALT_{TM} , then we can use U to construct a TM T' that decides A_{TM} :

- $T'(\langle M, w \rangle)$ runs $U(\langle M, w \rangle)$
- If U rejects, then reject
- If U accepts, run $M(w)$ and produce the output
- In general, this technique is: to prove language L is undecidable, we prove that "If L is decidable, then so is A_{TM} ".

A function $f : \Sigma^* \mapsto \Sigma^*$ is a *computable* function if there is a TM T that halts with exactly $f(w)$ written on tape, for every input w . A language L_1 is **mapping reducible** to language L_2 , formally $L_1 \leq_m L_2$, if there is a computable function f s.t.

$\forall w \in \Sigma^*, w \in L_1 \Leftrightarrow f(w) \in L_2$. Here, f is called a *mapping reduction* (or *many-one reduction*) from L_1 to L_2 .

- Mapping reducible relation \leq_m is *transitive*
- If $L_1 \leq_m L_2$, then $\neg L_1 \leq_m \neg L_2$
- If $L_1 \leq_m L_2$ and L_2 is recognizable, then L_1 is recognizable; If $L_1 \leq_m L_2$ and L_1 is unrecognizable, then L_2 is unrecognizable
- If $L_1 \leq_m L_2$ and L_2 is decidable, then L_1 is decidable; If $L_1 \leq_m L_2$ and L_1 is undecidable, then L_2 is undecidable

The language A_{TM} is "*complete*" in recognizability!

Interesting conclusions & corollaries, see Lec11 slides for details:

- $A_{TM} \leq_m \text{HALT}_{TM}, \neg A_{TM} \leq_m \neg \text{HALT}_{TM}, \text{HALT}_{TM} \leq_m A_{TM}, \text{HALT}_{TM} \equiv_m A_{TM}$
- $\text{EMPTY}_{TM} = \{\langle M \rangle \mid M \text{ is a TM s.t. } L(M) = \emptyset\}, \neg A_{TM} \leq_m \text{EMPTY}_{TM}$, thus unrecognizable
- $\text{EQ}_{TM} = \{\langle M, N \rangle \mid M, N \text{ are TMs and } L(M) = L(N)\}, \text{EMPTY}_{TM} \leq_m \text{EQ}$, thus unrecognizable

Oracle TMs & Turing Reduction

An **Oracle TM** T is one equipped with a set $B \subseteq \Gamma^*$ and an oracle tape, and T may enter a special state q_o to ask queries about whether string on oracle tape $\in B$ or not (goes to q_{yes} if yes, o.w., q_{no}). Assume this "oracle" check finishes in one step.

A language L_1 is *decidable with* L_2 if there is an oracle TM with oracle L_2 that accepts strings in L_1 and rejects others. We say L_1 **Turing reduces** to L_2 , formally $L_1 \leq_T L_2$.

- Turing reducible relation \leq_T is *transitive*
- If $L_1 \leq_T L_2$ and L_2 is decidable, then L_1 is decidable; If $L_1 \leq_T L_2$ and L_1 is undecidable, then L_2 is undecidable

We claim that $A_{TM} \leq_T \text{HALT}_{TM}$:

- In other words, we can decide the acceptance problem, given an oracle for the halting problem
- Proof idea: write pseudocode for an oracle TM for A_{TM} on input $\langle M, w \rangle$:
 - If $\langle M, w \rangle \in \text{HALT}_{TM}$, then run $M(w)$ and outputs its answer
 - Else, reject

Interesting conclusions & corollaries, see Lec11 slides for details:

- $A_{TM} \leq_T \text{HALT}_{TM}, \text{HALT}_{TM} \leq_T A_{TM}, \text{HALT}_{TM} \equiv_T A_{TM}$
- If $L_1 \leq_m L_2$, then $L_1 \leq_T L_2$
- $\neg A_{TM} \leq_T A_{TM}, \neg A_{TM} \not\leq_m A_{TM}$

Oracle TMs cannot solve all problems. In fact, there is an infinite hierarchy of unsolvable problems:

1. $\text{SUPERHALT}^0 = \text{HALT} = \{\langle M, x \rangle \mid M \text{ halts on } x\}$
2. $\text{SUPERHALT}^1 = \{\langle M, x \rangle \mid M, \text{ with oracle for HALT, halts on } x\}$
3. $\text{SUPERHALT}^2 = \{\langle M, x \rangle \mid M, \text{ with oracle for SUPERHALT}^1, \text{ halts on } x\}$
4. ...

Self-Reference & Recursion

There is a computable function $q : \Sigma^* \mapsto \Sigma^*$ s.t., for every string w , $q(w) = \langle P_w \rangle$ where P_w always prints out w and then accepts. Based on this, we can define a self-printing TM by:

- $B(\langle M \rangle) = \langle \text{"takes input } w, \text{ outputs } M(\langle M \rangle)" \rangle$
- Now consider the TM which takes input w and runs $B(\langle B \rangle)$: it just prints out a machine description $\langle \text{"takes input } w, \text{ outputs } B(\langle B \rangle)" \rangle$

This is the essential idea behind *quine programs*.

More formally, the *Recursion Theorem* states that for every computable function $T : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$, there $\exists \text{TM } R : \Sigma^* \mapsto \Sigma^*$, such that for every string w , $R(w) = T(\langle R \rangle, w)$. Construction of such TM:

- $B(\langle M \rangle) = \langle \text{"takes input } w, \text{ outputs } M(\langle M \rangle, w)" \rangle$
- $Q(\langle M \rangle, w) = T(B(\langle M \rangle), w)$
- $R(w) = T(B(\langle Q \rangle), w)$, we can show that $B(\langle Q \rangle) \equiv \langle R \rangle$

This theorem implies that we can use the operation "obtain your own description" in TM pseudocode!

A novel approach to prove that A_{TM} is undecidable: Assume $H(\langle M \rangle, w)$ decides A_{TM} . Define $T(\langle M \rangle, w)$ to run H and outputs the opposite answer. According to the Recursion Theorem, there is a TM R s.t. $R(w) = T(\langle R \rangle, w)$ and it would say R rejects w when R accepts w . Contradiction.

Also see proof of $\text{MIN} = \{\langle M \rangle \mid M \text{ is minimal-state}\}$ is undecidable in Lec12 slides.

Formal Systems

See Lec12 slides page 17~25. I don't really catch them.

Complexity

Communication Complexity

Consider a theoretical model of *communication*: we have a function $f(x, y) : \{0, 1\}^* \times \{0, 1\}^* \mapsto \{0, 1\}$, where

- Alice only knows x , a binary string input; Bob only knows y , another binary string input
- Goal: get result of $f(x, y) = 0/1$? by communicating as few bits as possible between Alice & Bob
- Assume they alternate in communicating, and in every step send 1 bit / a STOP signal, and the last bit sent is the result

Define a **protocol** computing f to be a pair of functions $A, B : \{0, 1\}^* \times \{0, 1\}^* \mapsto \{0, 1, \text{STOP}\}$, with the semantics:

```

1 // Communication protocol semantics
2 r = 0, b_r = eps;
3 while (b_r != STOP) {
4     r++;
5     if (r is odd)
6         Alice sends b_r = A(x, b_1...b_r-1);
7     else
8         Bob   sends b_r = B(y, b_1...b_r-1);
9 }
10 output f(x, y) = b_r-1;

```

The *cost* of a communication protocol (A, B) on n -bit strings is the *maximum* number of *rounds* (i.e. $r - 1$) taken by the algorithm over all possible input $x, y \in \{0, 1\}^*$. The *communication complexity* of f on n -bit strings, $cc(f)$, is the *minimum* cost over all protocols computing f on n -bit strings.

- We are only interested in the communication cost, not including computation on either side
- There is always a trivial protocol for any f :
 - Alice sends the bits of x in odd rounds and Bob sends the bits of y in even rounds
 - \Rightarrow For every function f , $cc(f) \leq 2n$
- Connection to streaming algorithms & DFAs: For x, y with $|x| = |y|$, define $f_L(x, y) = 1 \Leftrightarrow xy \in L$
 - If L has a streaming algorithm using $\leq S(n)$ space on string length $\leq 2n$, then $cc(f_L) \leq O(S(n))$. Idea: Alice runs streaming algorithm on x and reaches a memory configuration, she sends that configuration to Bob in $O(S(n))$ rounds, and Bob continues the algorithm on y from that configuration, then sends result bit to Alice
 - \Rightarrow For every regular L , it has a DFA to compute, so $cc(f_L) = O(1)$

Proving lower bounds of communication complexity:

- Define *communication pattern* of a protocol on input x, y to be the sequence of bits Alice & Bob send, i.e. $b_1 b_2 \dots b_{r-1}$. If x, y and x', y' have the same pattern P , then x, y' and x', y also have pattern P
- Suppose $cc(f) \leq n - 1$, then there are $\leq 2^n - 1$ possible patterns. By the Pigeonhole Principle, ...

Computation Time Complexity

The very basic notations of *asymptotic complexity* (*Bachman-Landau notations*):

Notation	Definition	Abbr.	Meaning
$f(n) = o(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$	<	f is dominated by g (strictly "smaller" than g)
$f(n) = O(g(n))$	$\lim_{n \rightarrow \infty} \left \frac{f(n)}{g(n)} \right < \infty$	\leq	f is bounded above by g
$f(n) = \Theta(g(n))$	$f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$	=	f is "equal" to g
$f(n) = \Omega(g(n))$	$\lim_{n \rightarrow \infty} \left \frac{f(n)}{g(n)} \right > 0$	\geq	f is bounded below by g
$f(n) = \omega(g(n))$	$\lim_{n \rightarrow \infty} \left \frac{f(n)}{g(n)} \right = \infty$	>	f dominates g (strictly "larger" than g)

Measuring worst-case **time complexity** of a TM can be done on counting the steps taken for a TM to halt on input of length n . Formally, $T : \mathbb{N} \mapsto \mathbb{N}$ where $T(n)$ = the maximum number of steps taken by M over all inputs of length n .

Example of a TM for deciding $L = \{0^k 1^k \mid k \geq 0\}$ using $O(n \log n)$ time (we say $L \in \text{TIME}(n \log n)$):

1. If w is not of the form $0^* 1^*$, reject
2. Repeat until all bits crossed out:
 1. If parity of 0s \neq parity of 1s, reject
 2. Cross out every other 0; Cross out every other 1
 3. If all bits are crossed out, accept

To prove that there is no algorithm using less time, we can prove that any $\text{TIME}\left(\frac{n \log n}{\alpha(n)}\right), \alpha(n)$ is an unbounded function, it contains only regular languages, thus not including L .

Different computation models can yield different time complexity! For example, on a two-tapes TM, there is an $O(n)$ algorithm for L :

1. Sweep over all 0s, copy them onto the second tape
2. Sweep over all 1s, each time crossing out a 0 from the second tape

Every multi-tape TM using $t(n)$ time on a language L has an equivalent $O(t^2(n))$ time one-tape TM \Rightarrow Language decidable in polynomial time on any multi-tape TM can be decided in polynomial time on a traditional one-tape TM. See an intuitive simulation on Lec13 slides, page 17-20.

An *efficient* universal TM U is one that takes in an extra input 1^t and simply rejects when the simulated TM exceeds t steps. Obviously, U accepts $\langle M, w, 1^t \rangle \Leftrightarrow M$ accepts w in t steps. We can also guarantee that such U runs in $O(|M|^2 t^2)$ time.

The **Time Hierarchy Theorem** states that for all reasonable $f, g : \mathbb{N} \mapsto \mathbb{N}$ where $g(n) > n^2 f^2(n)$ for all n , $\text{TIME}(f(n)) \subsetneq \text{TIME}(g(n))$. "We can solve *strictly more* problems if given quadratically more time to compute."

- Proof idea is to use *diagonalization* with a clock, and the contradiction implies that the input cannot be decided in $f(|M|)$ time. But, by picking $g(|M|) > |M|^2 t^2 = |M|^2 f^2(|M|)$, we can construct a universal TM running in $O(g(|M|))$ time to simulate a TM over this problem that runs in $f(|M|)$ time
- Claim actually still holds for $g(n) > f(n) \log^2 f(n) \Rightarrow \text{TIME}(n) \subsetneq \text{TIME}(n^2) \subsetneq \text{TIME}(n^3) \subsetneq \dots$

Nondeterminism & P vs. NP

Define:

$$P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k).$$

- PRIMES = $\{n \mid n \text{ is a prime number}\} \in P$

These are the *effectively decidable* problems in the world of complexity theory (can be efficiently solved on real-world machines). The **Extended Church-Turing Thesis** states that our notion of *efficient* algorithms \Leftrightarrow polynomial time TMs. This thesis is under doubt nowadays because of the existence of quantum algorithms.

Define a *decidable predicate* $R(x, y)$ as a proposition about inputs x, y s.t. some TM M implements R (i.e., $R(x, y) = \text{true} \Rightarrow M(x, y)$ accepts, $R(x, y) = \text{false} \Rightarrow M(x, y)$ rejects; i.e., R is a computable function: $\Sigma^* \times \Sigma^* \mapsto \{\text{true}, \text{false}\}$). Theorems states that a language A is recognizable \Leftrightarrow there is a decidable predicate $R(x, y)$ s.t. $A = \{x \mid \exists y \in \Sigma^*, R(x, y) = \text{true}\}$.

- This bridges recognizability via decidability. The proof is trivial for \Rightarrow , and for \Leftarrow , let $R(x, y)$ be true iff M accepts x in $|y|$ steps.
- Examples of using this theorem to show some language is recognizable - $L = \{\langle M \rangle \mid M \text{ accepts at least one string}\}$: let $R(\langle M \rangle, \langle x, y \rangle)$ be true iff M accepts string x in $|y|$ steps. Keep guessing some x and *verify* it in finite $|y|$ time!
- This implies a very important corollary: "**Determinism vs. Nondeterminism**, are they equally powerful?"
 - Yes for finite automata
 - No for Turing machines
 - ??? for polynomial time

A **Nondeterministic Turing Machine** (NTM) is one whose state diagram is an NFA. A formal definition is a 7-tuple $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$,

- Composed of:
 - Normal components of a TM
 - $\delta : Q \times \Gamma \mapsto 2^{Q \times \Gamma \times \{L,R\}}$
- An *accepting computation history* for N on w is a sequence of configurations C_0, C_1, \dots, C_t following the definition of acceptance of a TM
 - $N(w)$ accepts in t time \Leftrightarrow such a history exists
 - N has *time complexity* $T(n)$ if on all inputs w of length at most n , N halts in $T(n)$ time

Now we can define $\text{NTIME}(t(n)) = \{L \mid L \text{ is decided by an } O(t(n)) \text{ time NTM}\}$. Define:

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k).$$

Can think of it as "recognizability" in complexity field. It means that tracing down a specific computation history in the "nondeterministic parallel tree" takes polynomial time. Equivalently, it means there is an algorithm which can *verify* (i.e., *prove*) whether a given solution (*certificate*) is correct, in polynomial time. [Existential Analogy]

Formally, $L \in \text{NP} \Leftrightarrow$ there is a constant k and a polynomial-time NTM V (verifier) s.t.

$$L = \{x \mid \exists y \in \Sigma^* \text{ where } |y| \leq k|x|^k, V(x, y) = \text{true}\}.$$

Examples of NP problems:

- Boolean SAT problems: $\text{SAT} = \{\phi \mid \phi \text{ is a satisfiable boolean formula}\}$
 - Special case: *conjunctive negation formula* (CNF) - conjunction of clauses who are disjunctions of literals
 - 3-CNF takes the form: $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee \neg x_5) \wedge \dots$; each clause has 3 literals
 - 3-SAT = $\{\phi \mid \phi \text{ is a satisfiable 3-CNF}\}$
- Hamiltonian path problem
- EQUIV / NEQUIV formula
- K -Clique problem; Independent set; Vertex cover
- Knapsack problems (0-1)
- ... (many more)

NP-Completeness & The Cook-Levin Theorem

A **polynomial time reduction** is a mapping reduction where the function f is computable in polynomial time. We denote that as $A \leq_P B$.

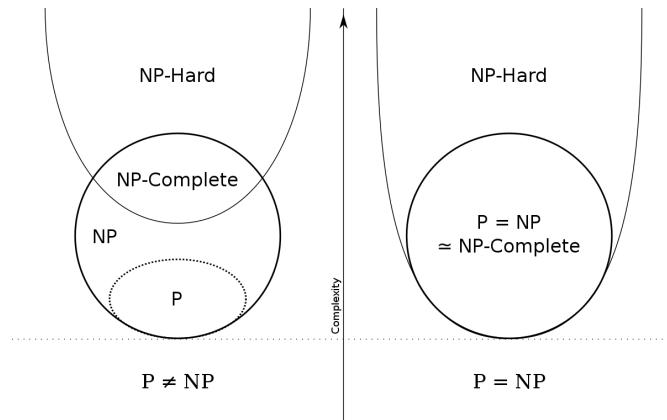
- A useful property of such f is that for any input w , $|f(s)| \leq k|w|^k$
- \leq_P is also transitive
- If $L_1 \leq_P L_2$ and $L_2 \in \text{P}$, then $L_1 \in \text{P}$; If $L_1 \leq_P L_2$ and $L_1 \notin \text{P}$, then $L_2 \notin \text{P}$
- If $L_1 \leq_P L_2$ and $L_2 \in \text{NP}$, then $L_1 \in \text{NP}$; If $L_1 \leq_P L_2$ and $L_1 \notin \text{NP}$, then $L_2 \notin \text{NP}$

So we can ask: what are the "hardest" NP problems under such reduction? We define a language L to be **NP-Complete** (NPC) if $L \in \text{NP}$ and L is **NP-Hard**: for every $L' \in \text{NP}$, $L' \leq_P L$.

- We can easily show that $A_{\text{NTM}} = \{\langle N, w, 1^t \rangle \mid \text{Nondeterministic TM } N \text{ accepts } w \text{ in } \leq t \text{ steps}\}$ is NPC
- The **Cook-Levin Theorem** states that the 3-SAT language is NPC! See Lec15 slides for a high-level proof.

The entire P=NP? question can be answered if we can prove whether a logic problem 3-SAT \in or \notin P.

3-SAT can be polynomial-time reduced to problems including (see Lec16 slides): Clique, Vertex cover, Independent set, Subset sum, Knapsack, Fair partition, Bin packing, Hamiltonian path, Longest path.... So these are all NP-Complete, thus equivalent in some deep sense.



CoNP & Oracle Complexity

The class **coNP** is the set of languages whose complement is in NP: $\text{coNP} = \{L \mid \neg L \in \text{NP}\}$. This is called *conondeterministic computation*. For a coNP language L , there is a polynomial-time algorithm that can verify that a certificate is not in L (i.e., can efficiently verify a counter-example). In other words, a conondeterministic machine "tries all" polynomial-time paths and accepts only if all these paths lead to accept. [Universal Analogy]

- $P \subseteq \text{coNP}$; generally, deterministic complexity is closed under complement, so in fact $L \in P \Leftrightarrow \neg L \in P$
- Is $NP = \text{coNP}$? It is also an open question!

Examples of coNP problems:

- TAUTOLOGY = $\{\phi \mid \text{every variable assignment satisfies } \phi\} = \{\phi \mid \neg \phi \in \text{UNSAT}\}$
- UNSAT = $\{\phi \mid \text{no variable assignment can satisfy } \phi\} = \neg \text{SAT}$
- ...

A language is **coNP-Complete** if $L \in \text{coNP}$ and L is **coNP-Hard**: for every $L' \in \text{coNP}$, $L' \leq_P L$.

- Key trick: $A \leq_P B \Leftrightarrow \neg A \leq_P \neg B$, so we can easily prove coNP-completeness by using NP-completeness
- Easy to see that UNSAT is coNP-complete, and the same for others

A very important class of languages is the $\text{NP} \cap \text{coNP} = \{L \mid L \in \text{NP} \wedge \neg L \in \text{NP}\}$.

- FACTORING = $\{(n, k) \mid n > k > 1 \text{ and there is a prime factor } p \text{ of } n \text{ where } k \leq p < n\} \in \text{NP} \cap \text{coNP}$
- Is $P = \text{NP} \cap \text{coNP}$? is also an open question!

Next we move on to *complexity classes with oracles*. Denote P^B be the set of languages decidable in polynomial-time with an oracle for B . Similarly:

- P^P be the class of languages decidable in polynomial-time with an oracle for some language in P
 - $P^P = P$ because running polynomial steps of polynomials still gives a polynomial
- P^{NP} be the class of languages decidable in polynomial-time with an oracle for some language in NP
 - For any language B which is NP-complete, $P^B = P^{NP}$; For example, $P^{NP} = P^{\text{SAT}} = \dots$
 - $\text{NP} \subseteq P^{NP}$, $\text{coNP} \subseteq P^{NP}$, $P^{NP} = P^{\text{coNP}}$
 - FIRST-SAT = $\{(\phi, i) \mid \phi \text{ is satisfiable and } i\text{-th variable of the lexicographically first satisfying assignment is true}\} \in P^{NP}$
- NP^{NP} be the class of languages decidable by a polynomial-time NTM with an oracle for some language in NP
 - Is $NP = NP^{NP}$? is an open question!

- coNP^{NP} be the class of languages decidable by a polynomial-time conondeterministic TM with an oracle for some language in NP
 - Is $\text{coNP}^{\text{NP}} = \text{NP}^{\text{NP}}$? is an open question!
 - $\text{MIN-FORMULA} = \{\phi \mid \phi \text{ is minimal among all its equivalences}\} \in \text{coNP}^{\text{NP}}$

Space Complexity

We measure **space complexity** by finding the largest tape index reached during computation of TM. The worst-case space complexity $S(n)$ is the largest tape index reached by TM on any input of length n .

Use $\text{SPACE}(s(n))$ as the set of languages decided by a TM with $O(s(n))$ space complexity. We have $\text{NTIME}(t(n)) \subseteq \text{SPACE}(t(n))$. Every multi-tape TM using $s(n)$ space on a language L has an equivalent $O(s(n))$ space one-tape TM.

Like the Time Hierarchy, we have the **Space Hierarchy Theorem** that for all reasonable $f, g : \mathbb{N} \mapsto \mathbb{N}$ where $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, $\text{SPACE}(f(n)) \subsetneq \text{SPACE}(g(n))$. "We can solve *strictly more* problems if given more space to compute."

Define:

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k).$$

- $\text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$
- Intuition: You can always re-use space, but you cannot re-use time! So same order of space is intuitively stronger.

For every *halting* TM using $s(n)$ space, the upper bound of running time is the total number of possible configurations $2^{O(s(n))}$, otherwise the TM loops. This implies that there is a TM running in $2^{O(s(n))}$ time that decides the same language.

This means that if we define:

$$\text{EXPTIME} = \bigcup_{k \in \mathbb{N}} \text{TIME}(2^{n^k}).$$

- Then $\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$
- And NP^{NP} and $\text{coNP}^{\text{coNP}} \subseteq \text{PSPACE}$, thus $\subseteq \text{EXPTIME}$

We can also have nondeterminism here. Define $\text{NSPACE}(s(n))$ as the set of languages decided by an NTM with $O(s(n))$ space complexity. Define:

$$\text{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k).$$

- NPSPACE is also $\subseteq \text{EXPTIME}$
- The **Savitch's Theorem** states that for functions $s(n) \geq n$, $\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s^2(n))$, this means $\text{PSPACE} = \text{NPSPACE}$!

One example of PSPACE-complete problems is: [True-Fully-Quantified-Boolean-Formula \(TQBF\)](#).

P , NP , & PSPACE complexity can model the complexity of some games:

- P captures short "zero-player" games, such as *Conway's Game of Life*
- NP captures short "one-player" games with a goal, including many single-perspective video games
- PSPACE captures short "two-player" games with a winning strategy

Randomized Complexity

A **probabilistic TM** M is an NTM where each nondeterministic step is a *coin flip*. Suppose each step has only two legal next moves. The probability that M runs on a path p is $\Pr[p] = 2^{-k}$, where k is the number of coin flips that occur on p .

The probability that M accepts input w is $\Pr[M \text{ accepts } w] = \sum_{p \text{ accepts on } w} \Pr[p]$.

- Language L in NP means $w \in L \Leftrightarrow \Pr[M \text{ accepts } w] > 0$
- Language L in coNP means $w \notin L \Leftrightarrow \Pr[M \text{ accepts } w] > 0$

A probabilistic TM M decides a language L with error ε iff for all input w :

- $w \in L \Rightarrow \Pr[M \text{ accepts } w] \geq 1 - \varepsilon$, and

- $w \notin L \Rightarrow \Pr[M \text{ doesn't accept } w] \geq 1 - \varepsilon$.

We can define *Bounded Probabilistic P* (BPP) to be the set of languages decided by some probabilistic polynomial-time TM with error at most $\varepsilon \leq \frac{1}{2} - \frac{1}{n^k}$ for some k . Using the *Error Reduction Lemma* as stated in Lec21 slides, we can show that for any language L in BPP, there is an equivalent machine M' which decides L in the same time complexity with error $< \frac{1}{2^{n^{2k}}}$. A one-sided version is *Randomized P* (RP) where negative inputs are always not accepted but positive inputs have bounded error.

One example of BPP problems is: Zero-Identical-Polynomial (ZERO-POLY). It is not known how to solve ZERO-POLY efficiently without randomness.

A widely-conjectured complexity space venn diagram looks like the following:

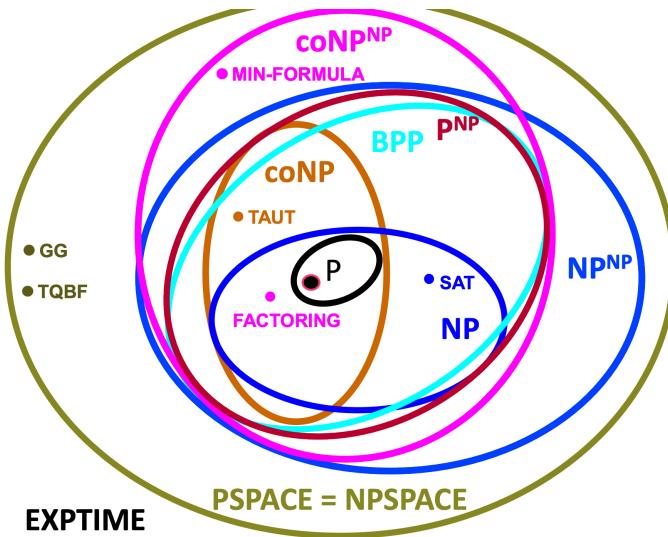


Figure taken from Ryan's slides.