Theory of computation[*]

October 1, 2014

Disclaimer

These notes have been prepared with the **only** purpose to help me pass the Computer Science qualifiying exam at the University of Illinois at Chicago. They are distributed as they are (including errors, typos, omissions, etc.) to help other students pass this exam (and possibly relieving them from part of the pain associated with such a process). I take **no responsibility** for the material contained in these notes (which means that you can't sue me if you don't pass the qual!) Moreover, this pdf version is distributed together with the original LaTeX (and LyX) sources hoping that someone else will improve and correct them. I mean in absolute no way to violate copyrights and/or take credit stealing the work of others. The ideas contained in these pages are **not mine** but I've just aggregated information scattered all over the internet.

Strongly suggested book

Before you even start reading this notes, do yourself a favor and go buy Michael Sipser's book. (http://www-math.mit.edu/∼sipser/book.html) It is crispy clear and beautifully written and prepared. All people that studied on this book passed the exam! :)

Contents

1 Regular Languages

1.1 DFA

Formal definition A *finite automaton* is a 5-uple $(Q, \Sigma, \delta, q_0, F)$, 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. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states.*

Formal definition of computation Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M accepts w if a sequence of states $r_0, r_1, \dots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \dots, n-1$ and
3. $r_n \in F$.

We say M recognizes language A if $A = \{w | M\, accepts\, w\}$.

Facts

- If A is the set of all strings that the machine M accepts, we say that A *is the language of machine* M and write $L(M) = A$. We say that M *recognizes* A or that M *accepts* A[1].
- A machine may accept several strings, but it always recognizes *only one language.*
- A language is *regular* if some finite automaton recognizes it.
- If the machine accepts no strings, it still recognizes one language, that is the empty language $\emptyset$.
- In order for a DFA to accept the empty string ε, the start state must also be an accept state.

1.2 NFA

Formal definition A *nondeterministic finite automaton* is a 5-uple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the *transition function*,[2]
4. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states.*

Formal definition of computation Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then N accepts w if we can write $w = y_1 y_2 \dots y_m$ where each y_i is a member of Σ_ε and a sequence of states $r_0, r_1, \dots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ for $i = 0, \dots, m-1$ and
3. $r_m \in F$.

[1] Recognize is preferred because accepting may also refer to strings

[2] Where $\Sigma_\varepsilon = \{\Sigma \cup \{\varepsilon\}\}$ and $\{\{\emptyset\}, \{Q\}\} \in \mathcal{P}(Q)$. Note that ε indicates the empty string of length 0. Note also how $L_A = \emptyset \neq L_B = \{\varepsilon\}$.

1.3 Equivalence of DFA and NFA

The following procedure converts a NFA $N = (Q, \Sigma, \delta, q_0, F)$ to the equivalent DFA $N' = (Q', \Sigma, \delta', q'_0, F')$.

1. $Q' = \mathcal{P}(Q)$; (which means that usually the DFA is much larger than the corresponding NFA!)
2. For $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$; (all states reachable from the states in R with input a)
3. $q'_0 = \{q_o\}$; (same start state)
4. $F' = \{R \in Q' | R \text{ contains an accept state of } N\}$

- If there are ε arrows we need to define $E(R) = \{q | q \text{ can be reched from } R \text{by traveling along 0 or more } \varepsilon \text{arrows}\}$ (called ε-closure) and modify points 3 and 4:

  3. $\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a)) \, \textit{for some} \, r \in R\}$ (same as before but all the reachable with an ε arrow)

  4. $q'_0 = E(\{q_o\})$; (same start state and all the ones reachable with an ε arrow)
- It is *very useful* to calculate E for all states before starting the conversion procedure
- We send a state q on the $\emptyset$ state on input a only if $\delta(q, a) = \emptyset$. This represents the stuck reject. Sink state that guarantees rejection.
- To go from a DFA to the equivalent NFA no further action is needed since DFAs are a particular case of NFAs.

In practice just start from the start state and compute δ only for all possible inputs and continue until no new states are added.

1.4 Closure automata construction[3]

- Union automata ($N_1 \cup N_2$): new start state with two ε arrows entering the start states of N_1 and N_2 respectively.
- Concatenation automata ($N_1 \cdot N_2$): ε arrows exiting the accept states of N_1 and entering the start state of N_2.
- Star automata (N_1^*): new start/end state with an ε arrow entering the old start state[4] and ε arrows connecting the accept states with the old start state.

1.5 Regular Expressions

We say that R is a *regular expression* if R is

1. a for some $a \in \Sigma$,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \cdot R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

[3] See pictures p. 59, 61, 62 of Sipser's book

[4] This is done to avoid recognition of other undesired strings other than ε.

1.5.1 Facts:

1. We can write $R_1 \cup R_2$ also as $R_1|R_2$
2. $R^+ = RR^*$
3. $R \cup \emptyset = R$
4. $R \cdot \varepsilon = R$
5. $R \cdot \emptyset = \emptyset$
6. $\emptyset^* = \{\varepsilon\}$

A language is regular if and only if some regular expression describes it.

1.5.2 Conversion from R.E. to NFA (if)

We just consider the 6 points in the definition of regular expression:

1. Two states (start and accept) connected by an a arrow
2. One state: start and accept at the same time
3. One state: start with no accept
4. $R = R_1 \cup R_2$[5]
5. $R = R_1 \cdot R_2$
6. $R = R_1^*$

1.5.3 Conversion from DFA to R.E. (only if)

1. If we have a NFA we need first to convert it to a DFA.
2. Convert the DFA M into a GNFA G:
   (a) Add a new start state s with an ε arrow to the old start state and a new accept state a with ε arrows from the old accept states.
   (b) If any arrows have multiple labels (e.g. $\delta(q_i, a) = \delta(q_i, b) = q_j$) replace them with a single arrow whose label is the union of the previous labels.
3. We choose a state (q_{rip}) of G at the time and we eliminate it obtaining a new GNFA G':
   (a) $Q' = Q - \{q_{rip}\}$
   (b) For *every* pair q_i and q_j such that $q_i \in Q' - q_{acc}$ and $q_j \in Q' - q_{start}$, $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$ where $R_1 = (q_i, q_{rip})$, $R_2 = (q_{rip}, q_{rip})$, $R_3 = \delta(q_{rip}, q_j)$and $R_4 = \delta(q_i, q_j)$ and where $q_i \in Q' - q_{acc}$ and $q_j \in Q' - q_{start}$.
4. Once we have only s and a the label of $\delta(s, a)$ is the regular expression

[5] See pictures p.67 of Sipser's Book for this last three points.

2 Context Free Languages

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

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

If u, v and w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv *yields* uwv, written $A \Rightarrow uwv$.

Say u *derives* v, written $u \Rightarrow v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$.

The *language* of the grammar is $\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$.

2.1 Ambiguity

- A derivation of a string w in a grammar G is a *leftmost derivation* if at every step the leftmost remaining variable is the one being replaced. (Two derivations may differ merely in the order in which they replace variables yet not in their overall structure!)
- A string is derived *ambiguously* in a CFG G if it has two or more different leftmost derivations. Grammar G is *ambiguous* if it generates some string ambiguously.
- Some CFLs, however, can be generated only by ambiguous grammars: these languages are called inherently ambiguous.

2.2 Chomsky normal form

A CFG is in Chomsky normal for if every rule is of the form

1. $A \rightarrow BC$
2. $A \rightarrow a$

where a is any terminal and A, B and C are any variable - except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

- Any CFL is generated by a CFG in Chomsky normal form
- Any string w such that $|w| = n$ can be derived with $\frac{n}{2}$ derivations
- Every regular language is context free
- CNF is useful when working with grammar related algorithms

2.3 Push Down Automata (PDA)

- A language is context free if and only if some push down automaton recognizes it.

Not asked in the theory part but useful for compilers!!!

3 Turing machines

Formal definition A *Turing machine* is a 7-uple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

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

Computation We define a *configuration* of a TM as a setting of the state, the current tape content and the current head location. We usually represent them as a string uqv where q is a state, uv is the current tape content and the head location is the first symbol of v. The tape contains only blanks following the last character of v.

- The *start configuration* of M on input w is q_0w.
- In an *accepting configuration* the state of the configuration is q_{accept}.
- In a *rejecting configuration* the state of the configuration is q_{reject}.
- Accepting and rejecting configurations are *halting configurations* and do not yield further configurations.
- A Turing machine M accepts input w if a sequence of configurations $C_1C_2, \ldots, C_k$ exists, where
  - C_1 is the start configuration of M on input w,
  - each C_i yields C_{i+1}
  - C_k is an accepting configuration.

Facts

- When we start a TM, three outcomes are possible: accept, reject or loop (the TM doesn't halt)
- We call a language *Turing-recognizable* (or recursively enumerable) if some Turing machine recognizes it. I can enumerate all the "yes" instances (strings that are accepted)
- Turing machines that halt on every input are called *deciders* because they always make a decision to accept or reject.
- We call a language *Turing-decidable* (or recursive) if some Turing machine decides it. Every decidable language is Turing-recognizable.

3.1 Variants of Turing machine

Every *multitape* Turing machine has an equivalent single-tape Turing machine.

Every *nondeterministic* Turing machine has an equivalent deterministic Turing machine. Which means that nondeterminism does not affect the power of the Turing machine model.

3.2 Church-Turing thesis

We choose to represent various computational problems by languages; which means we reduce algorithms (and problems) to languages that can or cannot be accepted by a Turing machine. If such a TM is a decider then the language (and therefore the problem) is decidable.

4 Decidability

4.1 Decidable languages

- $A_{DFA} = \{\langle B, w\rangle \,|B$ is a DFA that accepts input string $w\}$ is a decidable language. A_{DFA} represents the problem of testing whether a particular deterministic finite automaton B accepts a given string w: doing this is the same problem of testing whether $\langle B, w\rangle$ is a member of the language A_{DFA}. (To prove this just simulate B on M and if the simulation ends in an accept state, accept, otherwise, reject. The TM will terminate because w is finite)
- $A_{NFA} = \{\langle B, w\rangle \,|B$ is a NFA that accepts input string $w\}$ is a decidable language. (Proof: convert NFA to DFA)
- $A_{REX} = \{\langle R, w\rangle \,|R$ is a regular expression that generates string $w\}$ is a decidable language. (Proof: convert REX to DFA)
- $E_{DFA} = \{\langle A\rangle \,|A$ is a DFA and $L(A) = \emptyset\}$ is a decidable language. (Proof: Mark the states of A following the arrows and starting from the start state; if no accept state is marked, accept, otherwise, reject. The TM will terminate because $|Q_A|$ is finite.)
- $EQ_{DFA} = \{\langle A, B\rangle \,|A$ and B are DFAs and $L(A) = L(B)\}$ is a decidable language. (Proof: construct $L(C) = L(A) \otimes L(B)$ and test $E_{DFA}(C)$.)
- $A_{CFG} = \{\langle G, w\rangle \,|G$ is a CFG that generates string $w\}$ is a decidable language. (Proof: Convert grammar in Chomsky normal form, list all derivations with length $2|w| - 1$ steps, if any of these derivations generate w, accept, otherwise, reject. TM will terminate because the number of productions of length $2|w| - 1$ steps is finite.)
- $E_{CFG} = \{\langle G\rangle \,|G$ is a CFG and $L(G) = \emptyset\}$ is a decidable language. (Proof: Mark all terminals, mark any variable A where G has a rule $A \to U_1U_2 \ldots U_k$ and each symbol $U_1U_2 \ldots U_k$ has already been marked, if S is not marked accept, otherwise, reject. TM will terminate because number of variables in G is finite.)
- EQ_{CFG} is *undecidable*! (Because CFG are not closed w.r.t. intersection and complement)

Every context free language is decidable. This implies regular $\subseteq$ context free $\subseteq$ decidable $\subseteq$ Touring recognizable.

4.2 The halting problem

$A_{TM} = \{\langle M, w\rangle \,|M$ is a TM and M accepts input string $w\}$ is undecidable.

Proof is by contradiction using diagonalization. Suppose H is a decider for A_{TM} that, on input $\langle M, w\rangle$, accepts if M accepts w, and rejects if M does not accept w (including loops). We can define another TM D with$H$ as a subroutine that, on input $\langle M\rangle$, runs $H(M, \langle M\rangle)$ and outputs the opposite of what H outputs; that is, if H accepts, reject and if H rejects, accept. Run $D(\langle D\rangle)$ and observe that no matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus neither M nor H can exist.

- A_{TM} is Turing-recognizable (we can simulate $\langle M, w\rangle$ on a Turing machine U that accepts if M accepts, rejects if M rejects and loops if M loops).
- Some languages are not Turing recognizable. (We prove this by showing that the set of all Turing machines is countable while the set of all languages is uncountable) This means that there are some languages which not only can't be decided by a Turing machine but can't even be recognized by it.

- A language (only one, not the class!) is *co-Turing-recognizable* if it is the complement of a Turing-recognizable language. Co-RE languages are the ones for which I can enumerate all the NO instances (strings not accepted).

- A language (only one, not the class!) is decidable if and only if it is both Turing-recognizable and co-Turing-recognizable. R = RE ∩ co-RE.

- $\overline{A_{TM}}$ is not Turing-recognizable.

5 Reducibility

A *reduction* is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

As a rule of thumb, every time an unknown A can be reduced to a more specific B we can say A gets the same more specific properties of B. Vice-versa, whenever A can be reduced to an unknown B the only thing we can assert is that B has the same general properties of A.

5.1 Mapping reducibility

A function $f : \Sigma^* \to \Sigma^*$ is a *computable function* if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Language A is *mapping reducible* to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w, $w \in A \iff f(w) \in B$. The function f is called *reduction* of A to B.

5.1.1 Facts

- Note how the previous definition also implies $w \notin A \iff f(w) \notin B$ that is $\bar{A} \leq_m \bar{B}$.

- If $A \leq_m B$ and B is decidable, then A is decidable. (Proof: we have a decider M for B. Therefore we can specify a decider for A that first computes $f(w)$ and then runs M on input $f(w)$ and outputs whatever M outputs)

- If $A \leq_m B$ and A is undecidable, then B is undecidable. (Proof: corollary of the previous one)

- If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable. (Proof: same as before)

- If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable. (Proof: corollary of the previous one) Since we know that $\overline{A_{TM}}$is not Turing recognizable we often let A be $\overline{A_{TM}}$ so that, to show B is not Turing-recognizable, we just need to prove $A_{TM} \leq_m \bar{B}$ (which is equivalent to $\overline{A_{TM}} \leq_m B$.

5.2 Undecidable problems

- *$HALT_{TM}$* is undecidable. Proof of this theorem illustrates the general technique to prove undecidability! By contradiction we assume $HALT_{TM}$ is decidable and we show that A_{TM} is reducible to $HALT_{TM}$ which will imply that A_{TM} is decidable too, which is clearly a contradiction.

- $E_{TM} = \{\langle M\rangle | M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable. (Proof: we assume that E_{TM} is decidable and then we show that this yields to the decidability of A_{TM})

- $EQ_{TM} = \{\langle M_1, M_2\rangle | M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$ is undecidable. (Proof: show that E_{TM} is reducible to EQ_{TM} and, therefore, if EQ_{TM} were decidable, E_{TM} also would be decidable.)

- $REGULAR_{TM} = \{\langle M\rangle | M \text{ is a TM and } L(M) \text{ is a regular language}\}$ is undecidable.

- *Rice's theorem.* Let P be any nontrivial property of the language of a Turing Machine. The problem of determining whether a given Turing Machine's language has property P is undecidable.

- $A_{LBA} = \{\langle M, w\rangle | M$ is a LBA that accepts string $w\}$ is decidable. A *linear bounded automaton* (LBA) is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. This automata recognizes languages that are Type-1 in the Chomsky classification. (Proof: the reason why the problem is decidable is that such and automata can have only a limited number of configurations when string of length n is the input, which means such an automata can be simulated on a TM which will be able to recognize when a loop occurs, thanks to the finite amount of configurations)

6 Complexity

Let M be a deterministic TM that halts on all inputs. The running time or time complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that a M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time TM.

Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exists such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$. Using the asymptotic notation allows us to omit details of the machine description that affect the running time by at most a constant factor.

6.1 Complexity relationships among models

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the *time complexity class*, $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ Turing Machine.

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Let N be a nondeterministic Turing machine that is a decider. The running time of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ it the maximum number of steps that Nuses on any branch of its computation on any input of length n.

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine. (Which means nondeterministic machines are exponentially quicker than deterministic TMs!)

6.2 The class P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$P = \bigcup_k TIME(n^k)$$

P roughly corresponds to the class of problems realistically solvable on a computer. All reasonable deterministic models are polynomially equivalent

6.2.1 Examples of problems in P

When we analyze an algorithm to show that it runs in polynomial time, we need to do two things. First, we need to give a polynomial upper bound on the number of stages that the algorithm uses when it runs on an input of length n. Then, we have to examine the individual stages in the description of the algorithm to be sure that each stage can be implemented in polynomial time on a reasonable deterministic model. Also, we need to make sure that a reasonable encoding method for the problem is used where by "reasonable method" we mean one that allows for polynomial time encoding and decoding of objects into natural internal representation.

- $PATH = \{\langle G, s, t\rangle | G$ is a directed graph that has a directed path from s to $t\}$ is P. (Proof: run BFS/DFS starting from s)

- $RELPRIME = \{\langle x, y\rangle | x$ and y are relatively prime$\}$ is P. (i.e. 1 is the largest number that divides both)

- Every CFL (and therefore every CFG) is a member of P. (Proof: we already proved that A_{CFG} is decidable, we just need to show that the algorithm used in that proof runs in polynomial time, unfortunately it doesn't so we need to devise a new one using dynamic programming)

6.3 The class NP

A *verifier* for a language A is an algorithm V, where $A = \{w|V\, accepts\, \langle w, c\rangle\, for\, some\, string\, c\}$. We measure the running time of a verifier only in terms of the length of w so a *polynomial time verifier* runs polynomial time in the length of w.

A language A is *polynomial time verifiable* if it has a polynomial time verifier.

A verifier uses additional information, represented by the symbol c in the previous definition, to verify that string w is a member of A. This information is called a *certificate* or *proof.*

NP is the class of languages that have polynomial time verifiers.

An alternative characterization, which also gives the name to the NP class, can be given by using nondeterministic Turing machines.

NP is the class of languages that are decidable in polynomial time on a nondeterministic Turing machine. In other words,

$$NP = \bigcup_k NTIME(n^k)$$

where $NTIME(t(n)) = \{L|L$ is a language decided by a $O(t(n))$ nondeterministic Touring machine$\}$.

Note that we make a separate complexity class, called $CoNP$, which contains the languages that are complements of languages in NP (like $\overline{CLIQUE}$).

6.3.1 Examples of problems in NP

- $HAMPATH = \{\langle G, s, t\rangle|G$ is a directed graph with an Hamiltonian path from s to $t\}$ is NP. (Proof: the Hamiltonian path is the certificate)
- $CLIQUE = \{\langle G, k\rangle|G$ is an undirected graph with a k-clique$\}$ is NP. (Proof: the clique is the certificate)

6.4 Polynomial time reducibility

A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial computable function* if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

Language A is *polynomial time mapping reducible*, or simply *polynomial time reducible*, to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists , where for every w, $w \in A \iff f(w) \in B$. The function f is called *polynomial time reduction* of A to B.

Basically, polynomial time reducibility is the efficient analog to mapping reducibility.

6.4.1 Facts:

- If $A \leq_P B$ and $B \in P$, then $A \in P$.
- $3SAT = \{\langle\phi\rangle|\phi$ is a satisfiable 3cnf-formula$\}$ is polynomial time reducible to $CLIQUE$. (where $3SAT$ is a special case of $SAT = \{\langle\phi\rangle|\phi$ is a satisfiable boolean formula$\}$ [6])

[6] A Boolean formula is *satisfiable* if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. Note we don't have to find such an assignment, of course if we find it the formula is satisfiable, but we need to be able to say if there is at least one such an assignment.

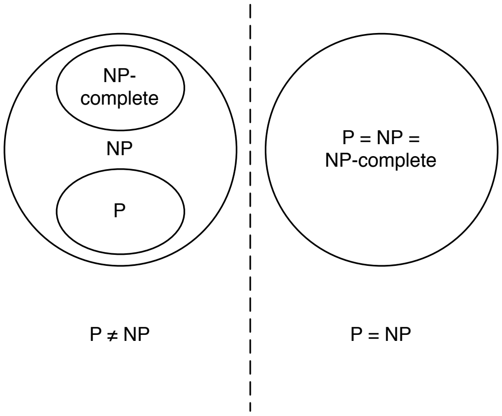
6.5 The class NP-complete

A language B is *NP-complete* if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

Which means that, if B is NP-complete and $B \in P$ then $P = NP$. In other words, if a polynomial time algorithm exists to solve B, then all problems in NP are solvable in polynomial time.

The possible containments among the classes P, NP and NP-complete are showed in the following diagram.



6.5.1 The P versus NP question

There are two possibilities: either $P = NP$ or $P \neq NP$. So far we have been unable to prove the existence of a single language in NP that is not in P. (which would prove they are different). At the same time we still have not found polynomial time algorithms for all problems in NP. (which will prove they are the same).

The best method known for solving languages in NP deterministically uses exponential time. In other words, we can prove that $NP \subseteq EXPTIME$ where $EXPTIME = \bigcup_k TIME\left(2^{n^k}\right)$, that is the class of languages that are decidable in exponential time on a single-tape deterministic Turing machine. Again, this is the practical method but NP could be contained in a smaller deterministic time complexity class as we will show in section 6.7.1.

6.5.2 Facts:

- If B is NP-complete and $B \leq_P C$ and $C \in NP$ then C is NP-complete. (Which means that if we want to demonstrate that a problem is NP-complete we need to show that a known NP-complete problem reduces to out problem in polynomial time!)
- SAT is NP-complete (Cook-Levin theorem: very important because identifies the “first” NP-complete problem)
- $SAT \in P \iff P = NP$ (Cook-Levin theorem rephrased)
- A problem B is NP-hard if and only if there is an NP-complete problem A that is polynomial time Turing-reducible to B. This differs from NP-complete problems because for those problems there is also the requirement that $B \in NP$ which is not present here.

6.5.3 Examples of problems in NP-complete

- $HAMPATH$ is NP-complete
- $CLIQUE$ is NP-complete

6.6 Space complexity

Analogously to what we did for time we estimate space complexity of Turing machines by using asymptotic notation.

Let $f : \mathcal{N} \to \mathbb{R}^+$be a function. The space complexity classes $SPACE(f(n))$ and $NSPACE(f(n))$ are defined as follows.

$$SPACE(f(n)) = \{L|L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Touring machine}\}$$

$$NSPACE(f(n)) = \{L|L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic Touring machine}\}$$

Savitch's theorem For any function $f : \mathcal{N} \to \mathbb{R}^+$, where $f(n) \geq n$, $NSPACE(f(n)) \subseteq SPACE(f^2(n))$. This means that deterministic machines can simulate nondeterministic machines using a surprisingly small amount of space. Hence the idea that space appears to be more powerful than time because space can be reused, whereas time cannot.

6.7 The class PSPACE and NPSPACE

By analogy with the class P, we define the class PSPACE as the class of languages that are decidable in polynomial space time on a deterministic Turing machine. In other words,

$$PSPACE = \bigcup_k SPACE(n^k)$$

We define NPSPACE, the nondeterministic counterpart to PSPACE, in terms of the NSPACE classes. However, $PSPACE = NPSPACE$, by virtue of Savitch's theorem, because the square root of any polynomial is still a polynomial.

6.7.1 Relationship to other complexity classes

$$P \subseteq NP \subseteq PSPACE = NSPACE \subseteq EXPTIME$$

The conjecture is that all of these containments are proper. We know for sure that $P \neq EXPTIME$ which means that at least one containment is proper, but we don't know which one.

A Closures

The following table summarizes the closure properties of language families with respect to some common operations.

Operation		Regular	DCFL	CFL	r.e.
Union	$L_1 \cup L_2 = \{w \vert w \in L_1 \vee w \in L_2\}$	Yes	NO	Yes	Yes
Concatenation	$L_1 \cdot L_2 = \{w \cdot z \vert w \in L_1 \wedge z \in L_2\}$	Yes	NO	Yes	Yes
Star	$L_1^* = \{\varepsilon\} \cup \{w \in L_1 \wedge z \in L_1^*\}$	Yes	NO	Yes	Yes
Intersection	$L_1 \cap L_2 = \{w \vert w \in L_1 \wedge w \in L_2\}$	Yes	NO	NO	Yes
Complement	$\overline{L_1} = \{w \vert w \notin L_1\}$	Yes	Yes	NO	NO
Reverse	$L_1^R = \{w^R \vert w \in L_1\}$	Yes	NO	Yes	Yes
Intersec. with reg. lang.	$L_1 \cap R = \{w \vert w \in L_1 \wedge w \in R, R\,\text{regular}\}$	Yes	Yes	Yes	Yes

- Regular = Regular languages
- DCFL = Determininistic Context Free Languages
- CFL = Context Free Languages
- r.e. = Recursively Enumerable languages