MTAT.07.004 — Complexity Theory September 8th, 2011

Lecture 2

Lecturer: Peeter Laud *Scribe(s): Alisa Pankova*

1 Introduction

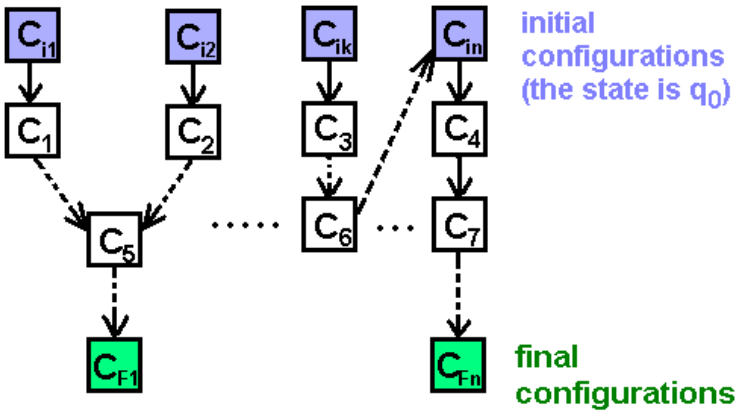
The second lecture introduces formal definitions of the famous classes P and NP, explains the relations between them, and shows consequences of what would happen if we proved that $P = NP$. We will overview polynomial reduction, or how one matemathical problem can be reduced to another. In the end, we will define *NP-hardness* and *NP-completeness*. There will be some examples of NP-complete languages and how one of them can be reduced to another.

On the previous lecture, we have defined two types of Turing machines: accepting a language and computing a function. Let all the Turing machine on this lecture be language-accepting. We do not lose anything in our proofs if we add this constraint.

2 Representing a Turing Machine as a Graph

Let M be a Turing machine with k tapes, an alphabet Γ, and a set of states Q. From the previous lecture, we know that a Turing Machine defines a relation $\xrightarrow{M}$ on the set of all the possible configurations $CONF^k_{\Gamma,Q}$. We may construct visual representation of a Turing machine as a directed graph $G = (V, E)$:

- The vertices of this graph are the configurations.
  $V := CONF^k_{\Gamma,Q}$
- An edge goes from configuration C to configuration C' iff $C \xrightarrow{M} C'$.
  $E := \left\{(C, C')|C \xrightarrow{M} C'\right\}$



Each vertex of a deterministic Turing machine may have only one outgoing edge. If there are no outgoing edges, it means that the configuration is final. Selecting the initial configuration determines the final configuration.

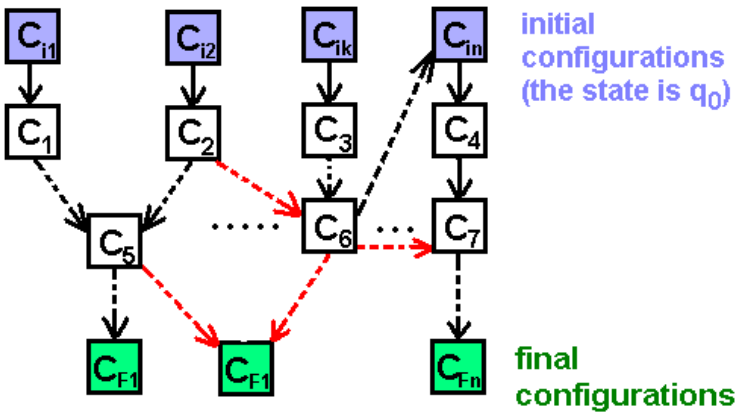
2.1 Non-deterministic Turing Machines

In the previous lecture, we have considered only deterministic Turing machines, where the work process of a TM is strictly determined by the input. As a useful computational model, we will now introduce *non-deterministic* Turing machines, where a TM may work in several different ways on the same input. In this case, we cannot claim that there is a (partial) function that maps a configuration C to the next configuration C'. The configuration C can actually have relations with several different configurations, and if we had at once $C \overset{M}{\rightarrow} C'$ and $C \overset{M}{\rightarrow} C''$, the relation $\overset{M}{\rightarrow}$ would no longer be a function, since a function applied to an argument must return exactly one value from its image. Instead of a function, we define a nondeterministic transition relation. A non-deterministic Turing machine is almost the same tuple $(\Gamma, Q, \delta, q_0, Q_F)$, where the transition *function* δ is replaced by a nondeterministic transition *relation*.

- **Deterministic:** $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times Move^k$
- **Non-deterministic:** $\delta \subseteq (Q \times \Gamma^k) \times (Q \times \Gamma^{k-1} \times Move^k)$

Note that in the second case δ is a subset of all the possible relations, what means that the Turing machine cannot go from each configuration to any other configuration, but there are transitions only between some of them.

A non-deterministic Turing machine is not a realistic model, but it can be very useful in describing some computational problems. In a graph representation, a vertex may have several outgoing edges.



2.2 About Time Function in Graphs

From a graph representation, we can easily derive the number of steps that the Turing machine makes to make its accept/reject decision on the given argument. In our graph, each step is represented by an edge. We just need to find the initial configuration U, the

final configuration V, and measure the length of the directed path from U to V. If there is no such path, it means the Turing machine cannot start with configuration U and finish with configuration V.

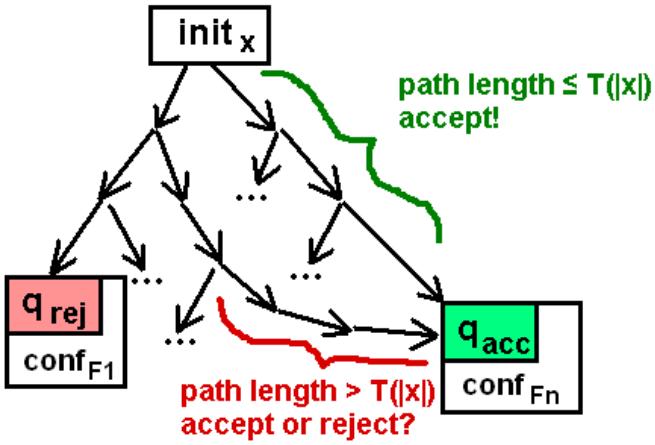
In a deterministic Turing machine, each vertex in a graph may have at most one outgoing edge. It implies that there is exactly one path between the initial configuration and the final configuration.

Consider now a non-deterministic Turing Machime M. Let $init_x$ be the initial configuration of M corresponding to the input $x \in \{0,1\}^*$. Define P as the path from the state $init_x$ to one of the accepting final configurations (if there are any).

$$P : init_x \to \langle q_{acc}, tapes, position \rangle.$$

We say that M accepts L in time T iff $[x \in L \Leftrightarrow \exists P : |P| \leq T(|x|)]$.

Let M be initialized in the state $init_x$. It is given a word x as the input. Now suppose that we have discovered that there exists a path from the corresponding initial configuration to the final configuration that accepts x. Even if this path does exist, but the length of this path is greater than $T(|x|)$, it would actually mean that M has not accepted the word x in time $T(|x|)$ as we wanted.



In the given lecture, we will not add any constraints to the path length. If we want to limit computation time to $T(|x|)$ for all $x \in \{0,1\}^*$ and for all paths the computation may proceed along, we may try to limit it somehow from the outside.

We may construct a NTM M', which accepts the same languague L as an input, and where all path from starting configurations have length at most $O(T)$. If we consider Turing machines as programs, we may say that M' contains a subprogram that runs in parallel with M and stops it if M has been running for too long time.

This new NTM M' would do the following:

1. Compute $L := T(|x|)$ and put into a separate tape. We may introduce several additional tapes for this purpose.

2. The main process:
   do
   do a step of M

L := L - 1
until (M stops) or ($L == 0$)

We need to show that, in any case, this construction stops in $O(T)$ steps. We know that making the steps of M do not add additional complexity, this time is still in $O(T)$. We have done this with Universal Turing Machine, and the computational time of $\mathcal{U}$ increases $C(\alpha)$ times, which is the constant that depends on M, not on the input.

There are more problems with decreasing the value of L. Although it may seem that subtracting one is trivial, it is actually not so simple!

Subtracting one affects the least significant bit of the value. If the least significant bit is 1, we change it to 0, and there are no problems. If this bit is 0, we cannot just change it to 1. We additionally need to move the head towards more significant bits and change all the 0-s on our way to 1-s, until we reach a 1 (there must be one since if all the bits were zeroes, the machine would stop earlier).

In the following example, the least significant bit is the leftmost.

$\triangleright$ $\overline{\mathbf{0}}011010101$

$\triangleright$ $1\overline{\mathbf{0}}11010101$

$\triangleright$ $11\overline{\mathbf{1}}1010101$

$\triangleright$ $11\overline{\mathbf{0}}1010101$

$\triangleright$ $1\overline{\mathbf{1}}01010101$

$\triangleright$ $\overline{\mathbf{1}}101010101$

If we encode the L as a bit string, its length would be $log(L)$. In the worst case, the value of L is 00....001, and the head has to move along the whole string. It seems that the complexity is already $O(T * log(T|x|))$, and that is definitely not what we wanted.

But it is the worst case when the head has to move along all the bits. It turns out that in average the number of steps in constant!

Suppose that we have a set of all bit strings with length $log(T(|x|))$. It is obvious that exactly half of them end with 1. It means that in 50% of the cases we make just one step.

What about the other possibilities? Consider how many bit strings end with 01. There are $1/(2^2) = 1/4$ of them, and each such bit string requires 3 steps (move forth, replace, move back). Analogically, there are 1/8 of the strings that end with 001, and they in turn require 5 steps (move 2 times forth, replace, move 2 times back). In general, we get that the average number of steps is the following:

$$\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 3 + \frac{1}{8} \cdot 5 + \frac{1}{16} \cdot 7 \ldots$$

This sum is finite for a finite number L. We can show that this sum is convergent even in the infinite case. First, rewrite this sum:

$$(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) + 2(\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots) + 2(\frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \ldots) + \ldots$$

Although the contents of the parentheses are infinite, they are actually geometric series, which converge no constant numbers.

$$1 + 2 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 2 \cdot \frac{1}{8} + \ldots = 1 + 2(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) = 1 + 2 = 3$$

The infinite sum is equal to the constant 3, and for finite input it is even smaller. Since the total time is now multiplied by some constant, it is still $O(T)$.

3 Non-Deterministic Random Access Machines

A random access machine consists of a program and a set of registers. In a non-deterministic RAM (NRAM), the program may split into branches in some places. A RAM acccepts a word if it accepts it in at least one of the branches.

More formally, we can say that a NRAM has nondeterministic choice operation $T \leftarrow \{0, 1\}$. The value of the register T will be nondeterministically chosen as 0 or 1. The configuration of RAM where this instruction is executed will have two successors in the RAM's computation graph.

The NRAM-s and NTM-s can simulate each other without much loss in efficiency, like the deterministic RAM-s and TM-s.

4 The Class NP

In the previous lecture, we introduced the classes $DTIME$ and P.

Let f be any function defined on natural numbers ($f : \mathbb{N} \to \mathbb{N}$).

The class $DTIME(f) \subseteq 2^{\{0,1\}^*}$ is defined as the set of all languages that can be accepted by a TM in time $O(f)$. The class P is defined as $P = \bigcup_{c \in \mathbb{N}} DTIME(\lambda n.n^c)$.

In the same way, we may define classes $NTIME$ and NP. The class $NTIME(f) \subseteq 2^{\{0,1\}^*}$ is defined as the set of all languages that can be accepted by a NTM in time $O(f)$. The class NP may be then defined as $NP = \bigcup_{c \in \mathbb{N}} NTIME(\lambda n.n^c)$.

If we replace "NTM" with "NRAM", the class NP will stay the same, because NTM and NRAM can simulate each other. There is definitely polynomial overhead, but we do not consider it since we still stay in NP.

Theorem 1 *$L \in P$ iff there exists a DTM M and a polynomial p, such that:*

- *$x \in L$ iff*
- *$\exists y \in \{0, 1\}^*$ with $|y| < p(|x|)$ such that $M(x, y)$ accepts x in at most $p(|x|)$ steps.*

In this case, we consider L as a set of pairs of bit strings: $L \subseteq \{0, 1\}^* \times \{0, 1\}^*$. If we want to check if x belongs to L, we first need to generate a certificate y (the definition of NP does not specify how this cerificate is being obtained). If we run afterwards $M(x, y)$, it either accepts or rejects x by help of y in polynomial time. $L \in P \overset{almost}{\Leftrightarrow} first(L) \in NP$.

The proof of this theorem is not shown on this lecture.

4.1 Examples of problems in NP

One of the most common problems in NP is search. First, the algorithm searches for a a certificate, and afterwards checks it in polynomial time.

1. Does a graph G have a clique of size at least k? (A clique is a subset of graph's vertices where all the vertices are connected with each other.)
   - **Certificate:** a set of vertices that are claimed to be a clique of size k.
   - **Check:** there must be at least k vertices in this set, and all of them must be connected to each other.
2. Does a boolean formula with variables have a satisfying assignment to those variables?
   - **Certificate:** an assignment of the variables of the given formula.
   - **Check:** evaluate the formula with the given assignment and compute its value.
3. Does a weighted graph have a traveling salesman tour of length at most k?
   - **Certificate:** a traveling salesman tour.
   - **Check:** the given tour must pass all vertices exactly once and its length must be at most k.
4. Is a number n composite?
   - **Certificate:** any factor k of n.
   - **Check:** the n should be divisible by k, and $k \neq 1$, $k \neq n$.
5. Can the vertices of a graph be colored with three colors? (The correct coloring means that if any two vertices are adjacent, they cannot have the same color.)
   - **Certificate:** the coloring of the vertices.
   - **Check:** there should be only three colors, and each pair of adjacent vertices must have different colors.
6. Are two graphs (given e.g. by their adjacency lists) isomorphic?
   - **Certificate:** the isomorphism mapping $f : G(V, E) \to G'(V', E')$.
   - **Check:** the given mapping must be really an isomophism. Check one-to-one correspondence between the vertices, and check if for each u, v having an edge (u, v) in G is equivalent to having an edge (u', v') in G', where $f(G) = G'$.
7. Do the vertices u and v of some graph belong to the same connected component?
   It seems that the certificate may be the path $u \to v$. We check if it is really a path, and if its endpoints are u and v. There is however a more interesting solution:
   - **Certificate:** null (no certificate).
   - **Check:** use breadth-first search to find the way from u to v.

There are no contradictions with the definition of NP since breadth-first search is bounded by polynomial time (it is even linear). We may simulate P by NP using empty certificates. The relation between P and NP is more precisely described in the next section.

5 Relation Between P and NP

Theorem 2 $P \subseteq NP \subseteq \bigcup_{c\in\mathbb{N}} DTIME(2^{n^c})$.

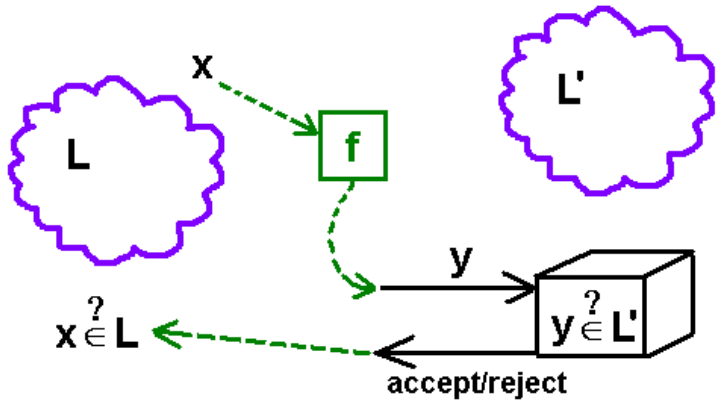
Here $DTIME(2^{n^c})$ is the exponential time class. It is a very powerful class which makes a complete search in the problem domain and check all the certificates one by one, until any of them matches or the domain ends.

Proof:

- $P \subseteq NP$: every DTM is a NTM. It is obvious since the function $\xrightarrow{M}$ is also a relation. We may also think that each tree is a graph, and there are probably more ways to describe why a DTM is a NTM.
- $NP \subseteq DTIME(2^{n^c})$: using time $2^{O(p(n))}$ we can check every certificate of length $p(n)$. In more general form, $NTIME(f) \subseteq \bigcup_{c\in\mathbb{N}} DTIME(\lambda n.c^{f(n)})$.

5.1 Polynomial Reducibility

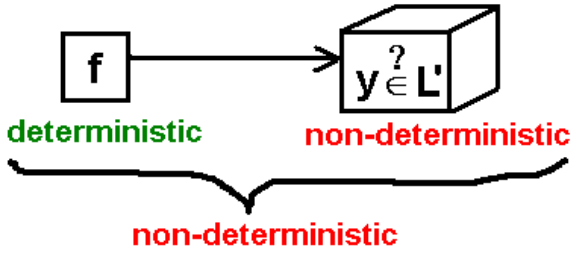
Suppose that we are given a word x and a language L. We want to check if x belongs to L, but we don't have an idea how to construct a TM that would decide the language L. Now suppose that we have some other TM M' that decides language L'. Given a bit string y, we may check if $y \in L'$. We would like to construct a function f, such that $f(x) = y$, and $x \in L \Leftrightarrow y \in L'$. In this case we can use M' and f and construct a new Turing machine M that would decide L. It would be a composition of f and M'. Of course, it is not always possible to construct such a function f. We would also want f to be easily computable.



If there does exist a polynomial-time computable function f such that for all $x \in \{0,1\}^*$ $x \in L$ iff $f(x) \in L'$, we say that the language L is **polynomially [many-one]** reducible

to the language L'. We denote it as $L \leq_m^p L'$, which can be in other words explained as "membership problem for L' is at least as hard to decide as membership problem for L". If we know how to test membership in L', we also know how to test membership in L.

If the machine that tests membership in L' is non-deterministic, the machine that tests membership in L will be also non-deterministic.

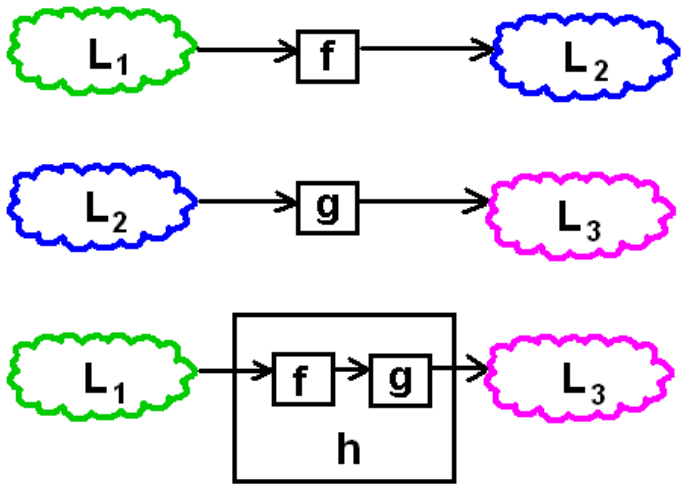


The reducibility is not just theoretical definition, but it can be very efficiently applied in practice. For example, suppose that we have designed a new cryptographic primitive A and want to prove that it is secure. We have another primitive B that is already proved to be secure. If we show that if the attacker could break A, then he could also break B, it would mean that A is secure since otherwise the attacker would break B, and we have proved before that it is impossible.

Reducibility also allows us to solve non-standard problems. If we are given a mathematical (or even non-mathematical) problem that we cannot solve, but we know that we are good at graphs, we can try to reduce the given problem to a graph problem that we know how to solve.

Reducibility has the following properties:

- If $L_1 \leq_m^p L_2$ and $L_2 \leq_m^p L_3$, then $L_1 \leq_m^p L_3$.
  Let f be the function that reduces L_1 to L_2, and g the function that reduces L_2 to L_3. We may define a function h that reduces L_1 to L_3 as $h = f \circ g$ (the composition function).



This property implies the following properties:

- If $L_1 \leq_m^p L_2$ and $L_2 \in P$, then $L_1 \in P$.
- If $L_1 \leq_m^p L_2$ and $L_2 \in NP$, then $L_1 \in NP$.

Let us define L^c as the complementary language of L, which contains exactly those words that do not belong to L ($L^c = \{0,1\}^* \setminus L$).

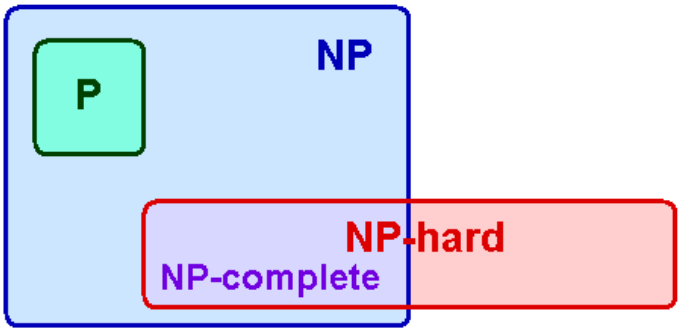
- If $L_1 \leq_m^p L_2$, then $L_2^c \leq_m^p L_1^c$.
  Suppose that we have a function f such that $x \in L_1 \Leftrightarrow f(x) \in L_2$. We need to construct g such that $x \in L_1^c \Leftrightarrow g(x) \in L_2^c$. We can use the property $x \notin L_1 \Leftrightarrow f(x) \notin L_2$ and still use the function f as g.

6 NP-hardness and NP-completeness

There are formal definitions of NP-hardness and NP-completeness:

- A language L is **NP-hard** if for all $L' \in NP$ we have $L' \leq_m^p L$. It means that each language in the class NP can be reduced to L.
- A language L is **NP-complete** if L is NP-hard and $L \in NP$.

It means that there are languages that are NP-hard, but they themselves do not belong to NP. The relations between classes can be represented graphically as sets.



There is a famous problem that has not yet been solved. The question is if $P = NP$. Nobody has proven that it is true, but nobody could also prove that it is false. There are however some properties that relate to this equation:

- If a language L is NP-hard and $L \in P$ then $P = NP$.
  This results from the definition of NP-hardness. If each problem in NP is reducible to a problem in P, it means that each problem in NP can be solved in polynomial time. Therefore, $NP \subseteq P$. On the other hand, $P \subseteq NP$. We get that $P = NP$. It means that if anyone constructs a language that belongs to P and is NP-hard (or proves that such a language does exist), he proves that $P = NP$.
- If a language L is NP-complete then $L \in P$ if and only if $P = NP$.
  Suppose that L is NP-complete. We need to show that $L \in P$ if and only if $P = NP$.

$\Rightarrow$ NP-completeness implies NP-hardness. We use the previous property and get $P = NP$.

$\Leftarrow$ NP-completeness implies that $L \in NP$. If $P = NP$ and $L \in NP$, it is obvious that $L \in P$.

7 Existence of NP-Complete Languages

The definitions of NP-completeness and NP-hardness would not be very useful if the corresponding languages would not exist in reality. But it has been proven that these languages do exsist.

Theorem 3 *There exist NP-complete languages.*

Proof: Consider the following language L. We show that it is NP-complete.

$$L = \{\langle M, x, 1^n\rangle | M \text{ accepts } x \text{ in at most } n \text{ steps } \}$$

where:

- M is a non-deterministic Turing machine.
- x is the input
- 1^n is the time bound. It is expressed in unary, and therefore its length is n, not $log(n)$.

In order to show that L is NP-complete, we need to show that $L \in NP$ and that L is NP-hard.

- L **is in NP:** the certificate consists of the choices M must make to accept x. The certificate is therefore a path in the configuration graph. We need to check the validity of this path, check if M can really make these steps.
- L **is NP-hard:** let $L' \in NP$, and let M' be the NTM that accepts L' in time T. We want to reduce L' to L. Take $f(x) = \langle M', x, 1^{T(|x|)}\rangle$. If M' finishes its work on input x in time $T(|x|)$ and accepts x, the function returns true (accepted). On the other hand, if M' rejects x or makes more than $T(|x|)$ steps, the function returns false (rejected). In this way we may reduce any language in NP to L. $\square$

8 The SAT Language

We have showed that there exists at least one NP-complete language, but are there any other languages? If we find a language $L' \in NP$ that we want to demonstrate as NP-complete, we do not need to prove that each language from NP is reducible to L'. We only need to prove that the previously defined NP-complete language L is reducible to L'.

Now we want to introduce a new language. But first it requires several definitions:

- A **boolean formula** over variables $u_1, \ldots, u_n$ consists of those variables and the logical operators $\vee$(or), $\wedge$(and), $\neg$(not), ... connecting them. We may define more logical operators, like $\rightarrow$(implication) or $\leftrightarrow$(equivalence), but these three are sufficient.

  - Let BF be the set of all boolean formulas.

- A **valuation** of $u_1, \ldots, u_n$ is a mapping from $\{u_1, \ldots, u_n\}$ to { true, false }. It means that after evaluation each variable in the formula gets assigned a value either $true$ or $false$. We may afterwards compute the value of the whole formula.

- Let $u_1, \ldots, u_n$ be the variables of a boolean formula. The formula can be:
  - **unsatisfiable:** is false for any valuation $(u_1 \wedge \neg u_1)$;
  - **satisfiable:** is true for at least one valuation $((u_1 \wedge u_2) \vee (u_2 \wedge \neg u_3))$;
  - **tautology:** is true for any valuation $(((u_1 \rightarrow u_2) \rightarrow u_1) \rightarrow u_1)$.

We may now define a language SAT $:= \{\varphi \in BF \mid \varphi \text{ is satisfiable } \}$. This language consists of exactly those boolean formulas that are satisfiable. If we want to check if $x \in BF$ belongs to SAT, we need to check if x is satiafiable.

The language SAT has been defined for any boolean formula. In order to make handling of the formulas easier, we could use some kind of normal form for them. A good approach is the **conjunctive normal form**. It also requires several definitions:

- A **literal** is either a boolean variable or its negation.

- A **disjunct** is $l_1 \vee l_2 \vee \cdots \vee l_r$, where $l_1, \ldots, l_r$ are literals.

- A boolean formula is in **conjuctive normal form** if it is of the form $D_1 \wedge D_2 \wedge \cdots \wedge D_m$, where $D_1, \ldots, D_m$ are disjuncts.
  - Let CNF be the language of all boolean formulas in conjuctive normal form.
  - Let k-CNF be the language of all boolean formulas in conjuctive normal form, where no disjunct has more than k literals.

Now we are going to define languages that contain only those satisfiable formulas that are in conjuctive normal form.

- CNFSAT is the language $\{\varphi \in \text{CNF} \mid \varphi \text{ is satisfiable } \}$.

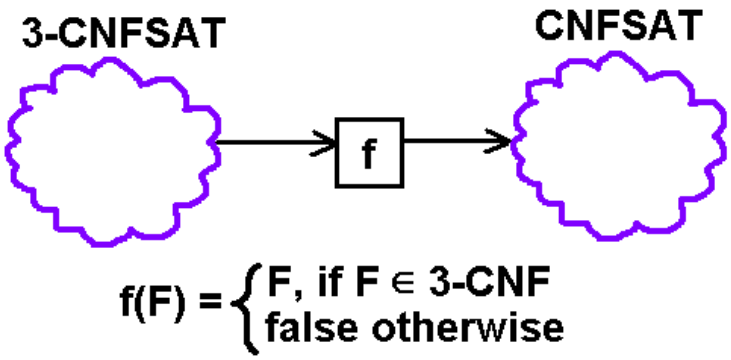
- k-CNFSAT is the language $\{\varphi \in k\text{-CNF} \mid \varphi \text{ is satisfiable } \}$.

If we prove that the SAT problem is reducible to CNFSAT problem, our life will be much easier.

Theorem 4 *If $k \geq 3$ then SAT $\leq_m^p$ k-CNFSAT $\leq_m^p$ CNFSAT $\leq_m^p$ SAT.*

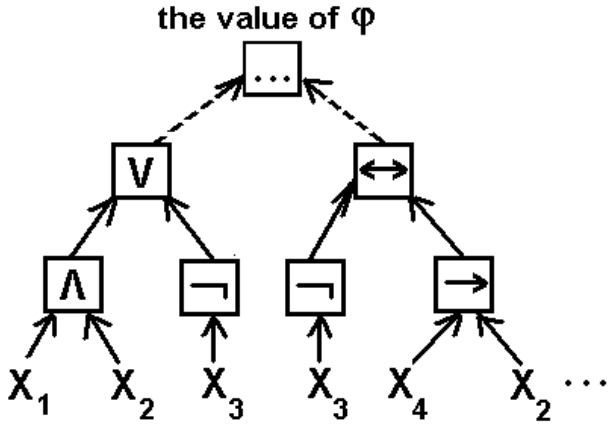
Proof: The most difficult part is to prove that SAT $\leq_m^p$ k-CNFSAT. It is obvious that k-CNFSAT $\leq_m^p$ CNFSAT since a conjunctive normal form with only k disjuncts in each conjunct is also a conjunctive normal form. In the same way, CNFSAT $\leq_m^p$ SAT since a formula in CNF is itself an instance of boolean formula.

Let us rewrite SAT $\leq_m^p$ k-CNFSAT as SAT $\leq_m^p$ 3-CNFSAT $\wedge$ 3-CNFSAT $\leq_m^p$ k-CNFSAT. The right part of this conjunction is true since 3 is an instance k. We can also easily show that 3-CNFSAT $\leq_m^p$ CNFSAT:



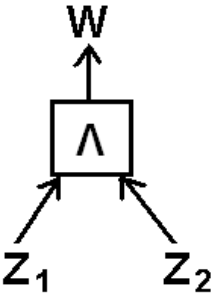
The only thing that requires more complex proof is that SAT $\leq_m^p$ 3-CNFSAT. We need to find an appropriate f. Its task would be to convert any boolean formula into its conjunctive normal form, where each disjunct would have not more than three literals.

Suppose that we are given a boolean formula φ. Let us consider its syntax tree. The nodes of this tree are boolean operations, and the leaves are variables. If any variable repeats in the formula several times, we still write all its instances as leaves.



Now we want to express our formula in 3-CNF. Let us take an arbitrary binary gate. It has two values coming in and one value going out. Let the incoming values be denoted as z_1 and z_2, and the outgoing value as w. If the incoming value is a variable, we write this variable in place of z_i. If it is not a variable, but is some expression, we do not have to write the whole expression in place of z_i! We denote it just as z_i, and express it as an output value of some other gate.

For example, let us take a conjunction gate. We are not interested if z_1 and z_2 are formulas or variables. We just know that they are some boolean values.



Although it is easier for us to write it as $z_1 \wedge z_2 \leftrightarrow w$, it is better to write this expression out according to the truth table: $(z_1 \wedge z_2 \rightarrow w) \wedge (z_1 \wedge \neg z_2 \rightarrow \neg w) \wedge (\neg z_1 \wedge z_2 \rightarrow \neg w) \wedge (\neg z_1 \wedge \neg z_2 \rightarrow \neg w)$.

Why this expression makes us happy? It is almost in conjuctive normal form. Each subexpression $(a \wedge b \rightarrow c)$ can be written out as:

$(a \wedge b \rightarrow c) \equiv$
$(\neg(a \wedge b) \vee c) \equiv$
$(\neg a \vee \neg b \vee c)$.

Each subexpression is a disjunct with at most three literals, and the whole gate formula is therefore in 3-CNF!

In the case of unary operations (negation), the truth table consists only of two rows, and it is very easy to write it as a disjunction if needed. We also may get rid of extra expressions for negation and use $\neg w$ instead of w when necessary.

This was for one gate, but we have more than one gate, and want to use all of them. For a gate formula, being true means existence in the syntax tree. In order to compose the combination of all the gates, we first write each of them out as a 3-CNF formula, and then make a conjunction of all of them. The conjuction of formulas that are in conjunctive normal form is itself also in conjunctive normal form, and the number of literals in disjuncts is still limited by three.

This would describe the syntax tree, but we have not added any information about the value of φ. We want to evaluate the whole syntax tree as true. The value of φ is an output of some gate, and we have already defined it somewhere in our CNF formula. Let it be denoted y_φ. We need to add y_φ to our conjunction as a disjunct. The formula that we get is now equivalent to the initial formula φ, and it is in 3-CNF.

Define $\Psi = \{\psi | \psi \text{ is a single gate CNF formula from } \varphi\}$. Our expression looks like this:

$$(\bigwedge_{\psi \in \Psi} \psi) \wedge y_\varphi$$

This conversion describes the reduction function f, and this is computable in polynomial time. The result of this conversion is in 3-CNF. If we get any boolean formula, we may convert it to 3-CNF formula using f, and thus reduce the satisfiability problem to 3-CNFSAT problem. □