Chapter 9

Computational Complexity; $\mathcal{P}$ and $\mathcal{NP}$

9.1 The Class $\mathcal{P}$

In the previous two chapters, we clarified what it means for a problem to be decidable or undecidable.

In principle, if a problem is decidable, then there is an algorithm (i.e., a procedure that halts for every input) that decides every instance of the problem.

However, from a practical point of view, knowing that a problem is decidable may be useless, if the number of steps (*time complexity*) required by the algorithm is excessive, for example, exponential in the size of the input, or worse.

For instance, consider the *traveling salesman problem*, which can be formulated as follows:

We have a set $\{c_1, \ldots, c_n\}$ of cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, the *distance matrix*, where d_{ij} denotes the distance between c_i and c_j, which means that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all $i \neq j$.

The problem is to find a *shortest tour* of the cities, that is, a permutation π of $\{1, \ldots, n\}$ so that the *cost*

$$C(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}$$

is as small as possible (minimal).

One way to solve the problem is to consider all possible tours, i.e., $n!$ permutations.

Actually, since the starting point is irrelevant, we need only consider $(n-1)!$ tours, but this still grows very fast. For example, when $n = 40$, it turns out that $39!$ exceeds 10^{45}, a huge number.

Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$.

The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

Thus, to capture the essence of practically feasible algorithms, we must limit our computational devices to run only for a number of steps that is bounded by a *polynomial* in the length of the input.

We are led to the definition of polynomially bounded computational models.

Definition 9.1. A deterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *polynomially decidable* if there is a polynomially bounded Turing machine that accepts L. The family of all polynomially decidable languages is denoted by $\mathcal{P}$.

Remark: Even though Definition 9.1 is formulated for Turing machines, it can also be formulated for other models, such as RAM programs.

The reason is that the conversion of a Turing machine into a RAM program (and vice versa) produces a program (or a machine) whose size is polynomial in the original device.

The following lemma, although trivial, is useful:

Lemma 9.1. *The class $\mathcal{P}$ is closed under complementation.*

Of course, many languages do not belong to $\mathcal{P}$. One way to obtain such languages is to use a diagonal argument. But there are also many natural languages that are not in $\mathcal{P}$, although this may be very hard to prove for some of these languages.

Let us consider a few more problems in order to get a better feeling for the family $\mathcal{P}$.

9.2 Directed Graphs, Paths

Recall that a *directed graph*, G, is a pair
$G = (V, E)$, where $E \subseteq V \times V$.
Every $u \in V$ is called a *node* (or *vertex*) and a pair $(u, v) \in E$ is called an *edge* of G.

We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n + 1$ edges ($n \geq 0$)

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 9.2. Given a graph G, an *Eulerian cycle* is a cycle in G that passes through all the nodes (possibly more than once) and every edge of G exactly once. A *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Eulerian Cycle Problem: Given a graph G, is there an Eulerian cycle in G?

Hamiltonian Cycle Problem: Given a graph G, is there an Hamiltonian cycle in G?

9.3 Eulerian Cycles

The following graph is a directed graph version of the Königsberg bridge problem, solved by Euler in 1736.

The nodes A, B, C, D correspond to four areas of land in Königsberg and the edges to the seven bridges joining these areas of land.

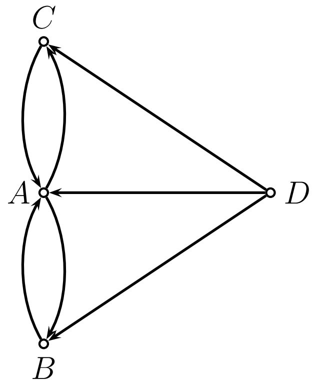


Figure 9.1: A directed graph modeling the Königsberg bridge problem

The problem is to find a closed path that crosses every bridge exactly once and returns to the starting point.

In fact, the problem is unsolvable, as shown by Euler, because some nodes do not have the same number of incoming and outgoing edges (in the undirected version of the problem, some nodes do not have an even degree.)

It may come as a surprise that the Eulerian Cycle Problem does have a polynomial time algorithm, but that so far, not such algorithm is known for the Hamiltonian Cycle Problem.

The reason why the Eulerian Cycle Problem is decidable in polynomial time is the following theorem due to Euler:

Theorem 9.2. *A graph $G = (V, E)$ has an Eulerian cycle iff the following properties hold:*

(1) The graph G is strongly connected.

(2) Every node has the same number of incoming and outgoing edges.

Proving that properties (1) and (2) hold if G has an Eulerian cycle is fairly easy. The converse is harder, but not that bad (try!).

Theorem 9.2 shows that it is necessary to check whether a graph is strongly connected. This can be done by computing the transitive closure of E, which can be done in polynomial time (in fact, $O(n^3)$).

Checking property (2) can clearly be done in polynomial time. Thus, the Eulerian cycle problem is in $\mathcal{P}$.

Unfortunately, no theorem analogous to Theorem 9.2 is know for Hamiltonian cycles.

9.4 Hamiltonian Cycles

A game invented by Sir William Hamilton in 1859 uses a regular solid dodecahedron whose twenty vertices are labeled with the names of famous cities.

The player is challenged to "travel around the world" by finding a closed cycle along the edges of the dodecahedron which passes through every city exactly once (this is the undirected version of the Hamiltonian cycle problem).

In graphical terms, assuming an orientation of the edges between cities, the graph D shown in Figure 9.2 is a plane projection of a regular dodecahedron and we want to know if there is a Hamiltonian cycle in this directed graph.

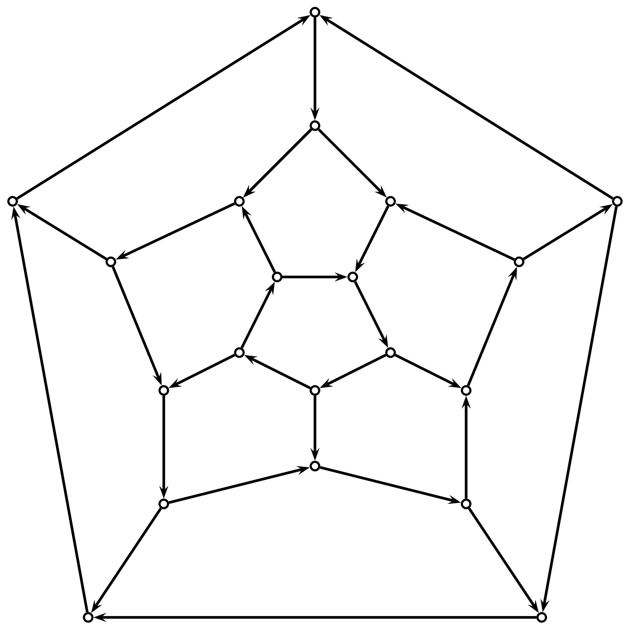


Figure 9.2: A tour "around the world."

Finding a Hamiltonian cycle in this graph does not appear to be so easy!

A solution is shown in Figure 9.3 below:

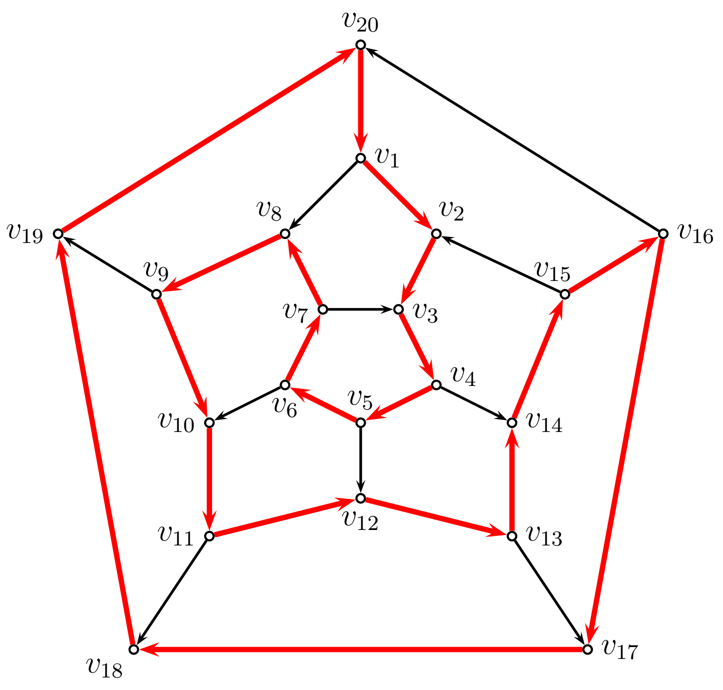


Figure 9.3: A Hamiltonian cycle in D.

A solution!

Remark: We talked about problems being decidable in polynomial time. Obviously, this is equivalent to deciding some property of a certain class of objects, for example, finite graphs.

Our framework requires that we first encode these classes of objects as strings (or numbers), since $\mathcal{P}$ consists of languages.

Thus, when we say that a property is decidable in polynomial time, we are really talking about the encoding of this property as a language. Thus, we have to be careful about these encodings, but it is rare that encodings cause problems.

9.5 Propositional Logic and Satisfiability

We define the syntax and the semantics of propositions in conjunctive normal form (CNF).

The syntax has to do with the legal form of propositions in CNF. Such propositions are interpreted as truth functions, by assigning truth values to their variables.

We begin by defining propositions in CNF. Such propositions are constructed from a countable set, **PV**, of *propositional (or boolean) variables*, say

$$\mathbf{PV} = \{x_1, x_2, \ldots, \},$$

using the connectives $\wedge$ (and), $\vee$ (or) and $\neg$ (negation).

We define a *literal (or atomic proposition)*, L, as $L = x$ or $L = \neg x$, also denoted by $\overline{x}$, where $x \in \mathbf{PV}$.

A *clause*, C, is a disjunction of pairwise distinct literals,

$$C = (L_1 \vee L_2 \vee \cdots \vee L_m).$$

Thus, a clause may also be viewed as a nonempty *set*

$$C = \{L_1, L_2, \ldots, L_m\}.$$

We also have a special clause, the *empty clause*, denoted $\perp$ or $\square$ (or $\{\}$). It corresponds to the truth value false.

A *proposition in CNF, or boolean formula*, P, is a conjunction of pairwise distinct clauses

$$P = C_1 \wedge C_2 \wedge \cdots \wedge C_n.$$

Thus, a boolean formula may also be viewed as a nonempty *set*

$$P = \{C_1, \ldots, C_n\},$$

but this time, the comma is interpreted as conjunction. We also allow the proposition $\perp$, and sometimes the proposition $\top$ (corresponding to the truth value true).

For example, here is a boolean formula:

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}.$$

In order to interpret boolean formulae, we use truth assignments.

We let BOOL $= \{\mathbf{F}, \mathbf{T}\}$, the set of truth values, where $\mathbf{F}$ stands for false and $\mathbf{T}$ stands for true.

A *truth assignment (or valuation)*, v, is any function $v\colon \mathbf{PV} \to \mathrm{BOOL}$.

Given a truth assignment, $v\colon \mathbf{PV} \to \mathrm{BOOL}$, we define the *truth value*, $\widehat{v}(X)$, of a literal, clause, and boolean formula, X, using the following recursive definition:

(1) $\widehat{v}(\perp) = \mathbf{F}$, $\widehat{v}(\top) = \mathbf{T}$.

(2) $\widehat{v}(x) = v(x)$, if $x \in \mathbf{PV}$.

(3) $\widehat{v}(\overline{x}) = \overline{v(x)}$, if $x \in \mathbf{PV}$, where $\overline{v(x)} = \mathbf{F}$ if $v(x) = \mathbf{T}$ and $\overline{v(x)} = \mathbf{T}$ if $v(x) = \mathbf{F}$.

(4) $\widehat{v}(C) = \mathbf{F}$ if C is a clause and iff $\widehat{v}(L_i) = \mathbf{F}$ for all literals L_i in C, otherwise $\mathbf{T}$.

(5) $\widehat{v}(P) = \mathbf{T}$ if P is a boolean formula and iff $\widehat{v}(C_j) = \mathbf{T}$ for all clauses C_j in P, otherwise $\mathbf{F}$.

Definition 9.3. We say that a truth assignment, v, *satisfies* a boolean formula, P, if $\widehat{v}(P) = \mathbf{T}$. In this case, we also write

$$v \models P.$$

A boolean formula, P, is *satisfiable* if $v \models P$ for some truth assignment v, otherwise, it is *unsatisfiable*. A boolean formula, P, is *valid (or a tautology)* if $v \models P$ for all truth assignments v, in which case we write

$$\models P.$$

One should check that the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}$$

is **un**satisfiable.

One may think that it is easy to test whether a proposition is satisfiable or not. Try it, it is not that easy!

As a matter of fact, the *satisfiability problem*, testing whether a boolean formula is satisfiable, also denoted SAT, is not known to be in $\mathcal{P}$.

Moreover, it is an NP-complete problem. Most people believe that the satisfiability problem is **not** in $\mathcal{P}$, but a proof still eludes us!

Before we explain what is the class $\mathcal{NP}$, let us remark that the satisfiability problem for clauses containing at most two literals (*2-satisfiability*, or 2-SAT) is solvable in polynomial time.

The first step consists in observing that if every clause in P contains at most two literals, then we can reduce the problem to testing satisfiability when every clause has exactly two literals.

Indeed, if P contains some clause (x), then any valuation satisfying P must make x true. Then, all clauses containing x will be true, and we can delete them, whereas we can delete $\overline{x}$ from every clause containing it, since $\overline{x}$ is false.

Similarly, if P contains some clause $(\overline{x})$, then any valuation satisfying P must make x false.

Thus, in a finite number of steps, either we get the empty clause, and P is unsatisfiable, or we get a set of clauses with exactly two literals.

The number of steps is clearly linear in the number of literals in P.

For the second step, we construct a directed graph from P. The nodes of this graph are the literals in P, and edges are defined as follows:

(1) For every clause $(\overline{x} \vee y)$, there is an edge from x to y and an edge from $\overline{y}$ to $\overline{x}$.

(2) For every clause $(x \vee y)$, there is an edge from $\overline{x}$ to y and an edge from $\overline{y}$ to x

(3) For every clause $(\overline{x} \vee \overline{y})$, there is an edge from x to $\overline{y}$ and an edge from y to $\overline{x}$.

Then, it can be shown that P is unsatisfiable iff there is some x so that there is a cycle containing x and $\overline{x}$.

As a consequence, 2-satisfiability is in $\mathcal{P}$.

9.6 The Class $\mathcal{NP}$, Polynomial Reducibility, $\mathcal{NP}$-Completeness

One will observe that the hard part in trying to solve either the Hamiltonian cycle problem or the satisfiability problem, SAT, is to *find* a solution, but that *checking* that a candidate solution is indeed a solution can be done easily in polynomial time.

This is the essence of problems that can be solved *non-determistically* in polynomial time: A solution can be guessed and then checked in polynomial time.

Definition 9.4. A nondeterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *nondeterministic polynomially decidable* if there is a polynomially bounded nondeterministic Turing machine that accepts L. The family of all nondeterministic polynomially decidable languages is denoted by $\mathcal{NP}$.

Of course, we have the inclusion

$$\mathcal{P} \subseteq \mathcal{NP},$$

but whether or not we have equality is one of the most famous open problems of theoretical computer science and mathematics.

In fact, the question $\mathcal{P} \neq \mathcal{NP}$ is one of the open problems listed by the CLAY Institute, together with the Poincaré conjecture and the Riemann hypothesis, among other problems, and for which *one million dollar* is offered as a reward!

It is easy to check that SAT is in $\mathcal{NP}$, and so is the Hamiltonian cycle problem.

As we saw in recursion theory, where we introduced the notion of many-one reducibility, in order to compare the "degree of difficulty" of problems, it is useful to introduce the notion of reducibility and the notion of a complete set.

Definition 9.5. A function $f\colon \Sigma^* \to \Sigma^*$ is *polynomial-time computable* if there is a polynomial $p(X)$ so that the following holds: There is a deterministic Turing machine M computing it so that for every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0 x$ is the starting ID.

Given two languages $L_1, L_2 \subseteq \Sigma^*$, *a polynomial reduction from L_1 to L_2* is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ so that for all $u \in \Sigma^*$,

$$u \in L_1 \quad \text{iff} \quad f(u) \in L_2.$$

For example, one can construct a polynomial reduction from the Hamiltonian cycle problem to SAT.

Remarkably, *every* language in $\mathcal{NP}$ can be reduced to SAT.

Intuitively, if L_1 is a hard problem and L_1 can be reduced to L_2, then L_2 is also a hard problem.

Thus, SAT is a hardest problem in $\mathcal{NP}$ (Since it is in $\mathcal{NP}$).

Definition 9.6. A language L is *$\mathcal{NP}$-hard* if there is a polynomial reduction from every language $L_1 \in \mathcal{NP}$ to L. A language L is *$\mathcal{NP}$-complete* if $L \in \mathcal{NP}$ and L is $\mathcal{NP}$-hard.

Thus, an $\mathcal{NP}$-hard language is as hard to decide as any language in $\mathcal{NP}$.

The importance of $\mathcal{NP}$-complete problems stems from the following theorem:

Theorem 9.3. *Let L be an $\mathcal{NP}$-complete language. Then, $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$.*

Next, we prove a famous theorem of Steve Cook and Leonid Levin (proved independently): SAT is $\mathcal{NP}$-complete.

9.7 The Cook–Levin Theorem: SAT is $\mathcal{NP}$-Complete

Instead of showing directly that SAT is $\mathcal{NP}$-complete, which is rather complicated, we proceed in two steps, as suggested by Lewis and Papadimitriou.

(1) First, we define a tiling problem adapted from H. Wang (1961) by Harry Lewis, and we prove that it is $\mathcal{NP}$-complete.

(2) We show that the tiling problem can be reduced to SAT.

We are given a finite set $\mathcal{T} = \{t_1, \ldots, t_p\}$ of *tile patterns*, for short, *tiles*. Copies of these tile patterns may be used to tile a rectangle of predetermined size $2s \times s$ $(s > 1)$.

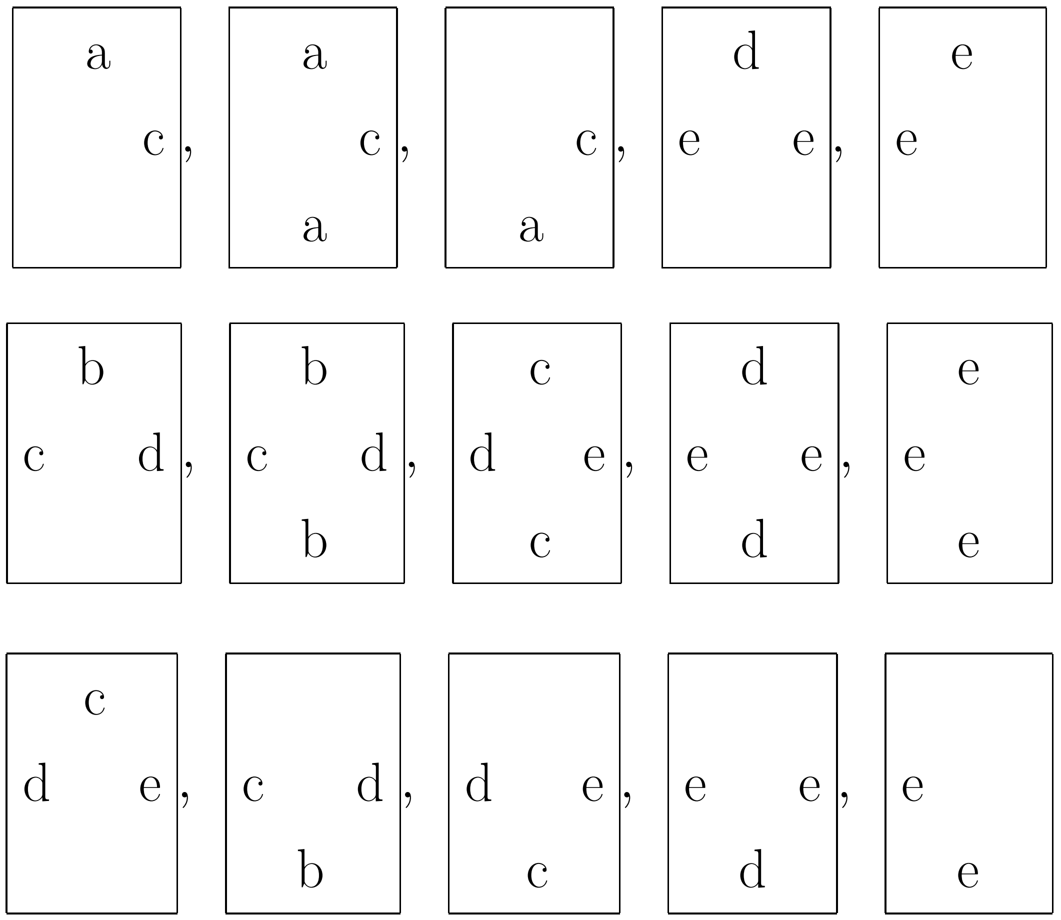
However, there are *constraints* on the way that these tiles may be adjacent horizontally and vertically.

The *horizontal constraints* are given by a relation $H \subseteq \mathcal{T} \times \mathcal{T}$, and the *vertical constraints* are given by a relation $V \subseteq \mathcal{T} \times \mathcal{T}$.

Thus, a *tiling system* is a triple $T = (\mathcal{T}, V, H)$ with V and H as above.

The bottom row of the rectangle of tiles is specified before the tiling process begins.

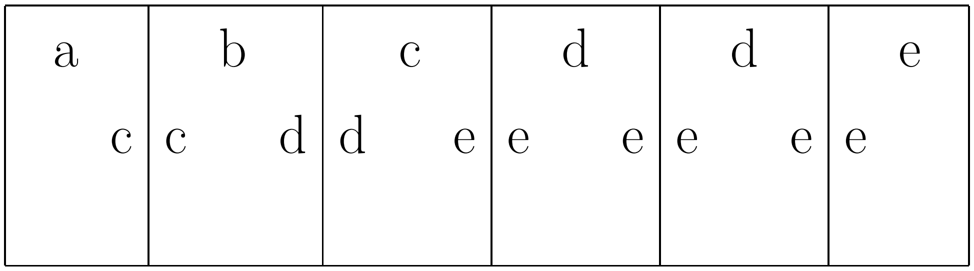
For example, consider the following tile patterns:



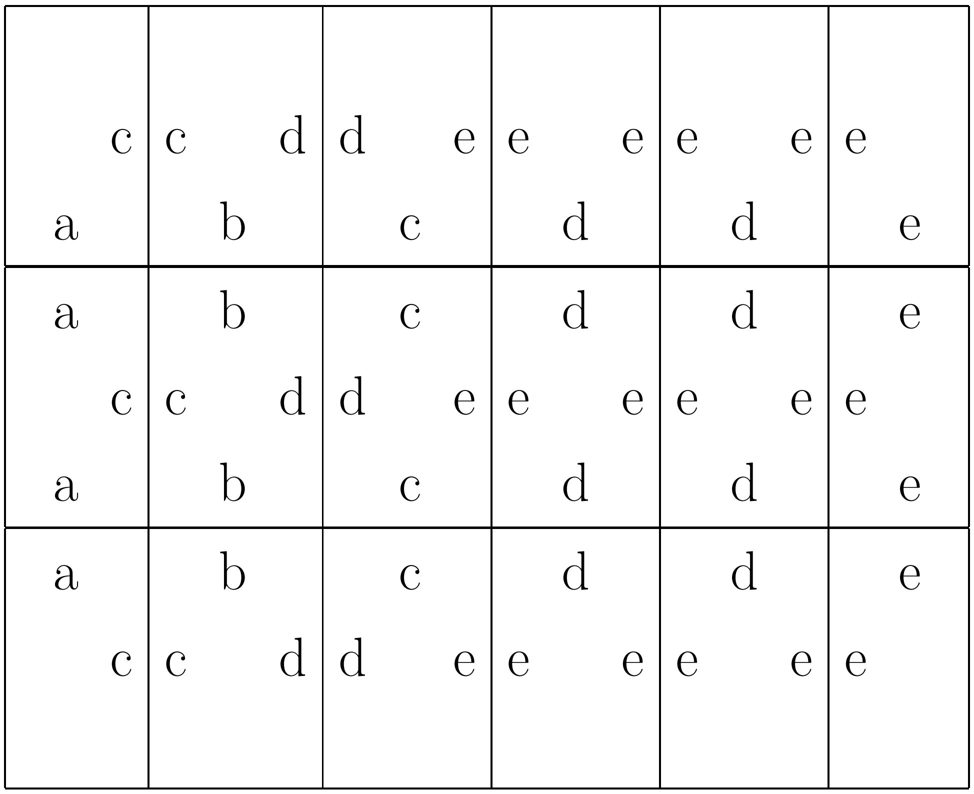
The horizontal and the vertical constraints are that the letters on adjacent edges match (blank edges do not match).

Let us try to find a 6×3 tiling with the initial row shown on the next page.

For $s = 3$, given the bottom row



we have the tiling shown below:



The problem is then as follows:

The Bounded Tiling Problem

Given any tiling system $(\mathcal{T}, V, H)$, any integer $s > 1$, and any initial row of tiles σ_0 (of length $2s$)

$$\sigma_0\colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \to \mathcal{T},$$

find a $2s \times s$-tiling σ extending σ_0, i.e., a function

$$\sigma\colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \times \{1, \ldots, s\} \to \mathcal{T}$$

so that

(1) $\sigma(m, 1) = \sigma_0(m)$, for all m with $1 \leq m \leq 2s$.

(2) $(\sigma(m, n), \sigma(m+1, n)) \in H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$.

(3) $(\sigma(m, n), \sigma(m, n+1)) \in V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$.

Formally, an *instance of the tiling problem* is a triple, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, where $(\mathcal{T}, V, H)$ is a tiling system, $\widehat{s}$ is the string representation of the number $s \geq 2$, in binary and σ_0 is an initial row of tiles (the bottom row).

For example, if $s = 1025$ (as a decimal number), then its binary representation is $\widehat{s} = 10000000001$. The length of $\widehat{s}$ is $\log_2 s + 1$.

Recall that the input must be a string. This is why the number s is represented by a string in binary.

If we only included a *single* tile σ_0 in position $(s+1, 1)$, then the length of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$ would be $\log_2 s + C + 2$ for some constant C corresponding to the length of the string encoding $(\mathcal{T}, V, H)$.

However, the rectangular grid has size $2s^2$, which is *exponential* in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$. Thus, it is impossible to check in polynomial time that a proposed solution is a tiling.

However, if we include in the input the bottom row σ_0 of length $2s$, then the size of the grid is indeed polynomial in the size of the input.

Theorem 9.4. *The tiling problem defined earlier is $\mathcal{NP}$-complete.*

Proof. Let $L \subseteq \Sigma^*$ be any language in $\mathcal{NP}$ and let u be any string in Σ^*. Assume that L is accepted in polynomial time bounded by $p(|u|)$.

We show how to construct an instance of the tiling problem, $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$, where $s = p(|u|) + 2$, and where the bottom row encodes the starting ID, so that $u \in L$ iff the tiling problem $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$ has a solution.

First, note that the problem is indeed in $\mathcal{NP}$, since we have to guess a rectangle of size $2s^2$, and that checking that a tiling is legal can indeed be done in $O(s^2)$, where s is *bounded by the the size of the input* $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, since the input contains the bottom row of $2s$ symbols (this is the reason for including the bottom row of $2s$ tiles in the input!).

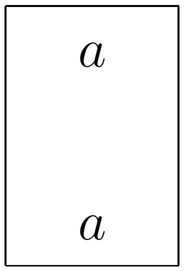
The idea behind the definition of the tiles is that, in a solution of the tiling problem, the labels on the horizontal edges between two adjacent rows represent a legal ID, $upav$.

In a given row, the labels on vertical edges of adjacent tiles keep track of the change of state and direction.

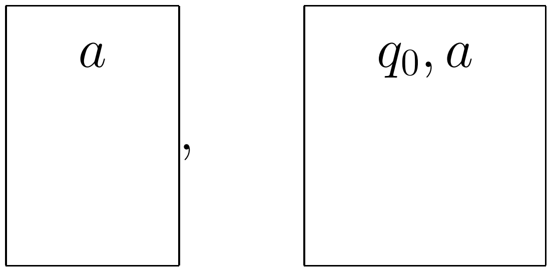
Let Γ be the tape alphabet of the TM, M. As before, we assume that M signals that it accepts u by halting with the output 1 (true).

From M, we create the following tiles:

(1) For every $a \in \Gamma$, tiles

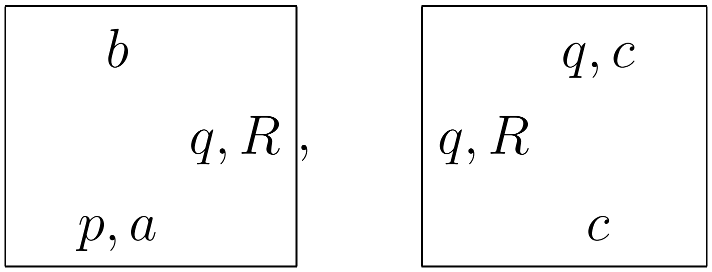


(2) For every $a \in \Gamma$, the bottom row uses tiles

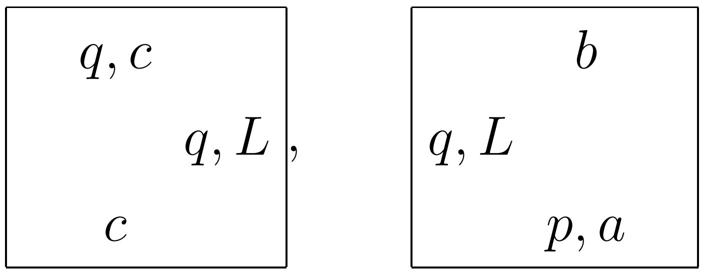


where q_0 is the start state.

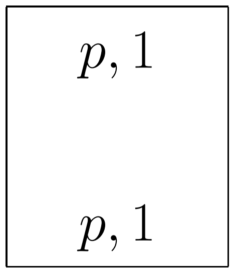
(3) For every instruction $(p, a, b, R, q) \in \delta$, for every $c \in \Gamma$, tiles



(4) For every instruction $(p, a, b, L, q) \in \delta$, for every $c \in \Gamma$, tiles



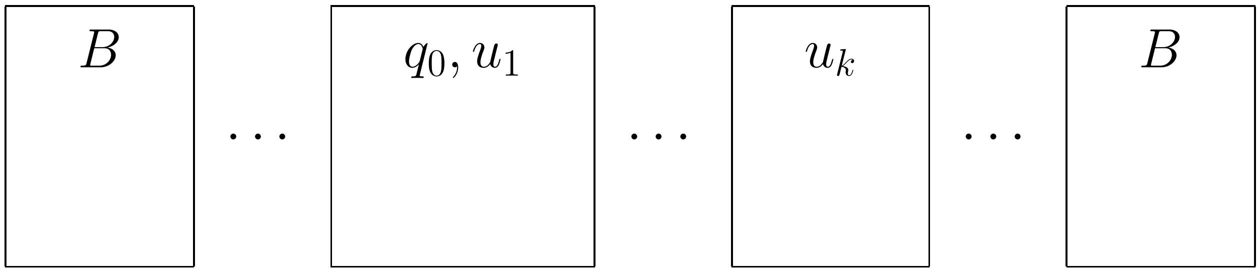
(5) For every halting state, p, tiles



The purpose of tiles of type (5) is to fill the $2s \times s$ rectangle iff M accepts u. Since $s = p(|u|) + 2$ and the machine runs for at most $p(|u|)$ steps, the $2s \times s$ rectangle can be tiled iff $u \in L$.

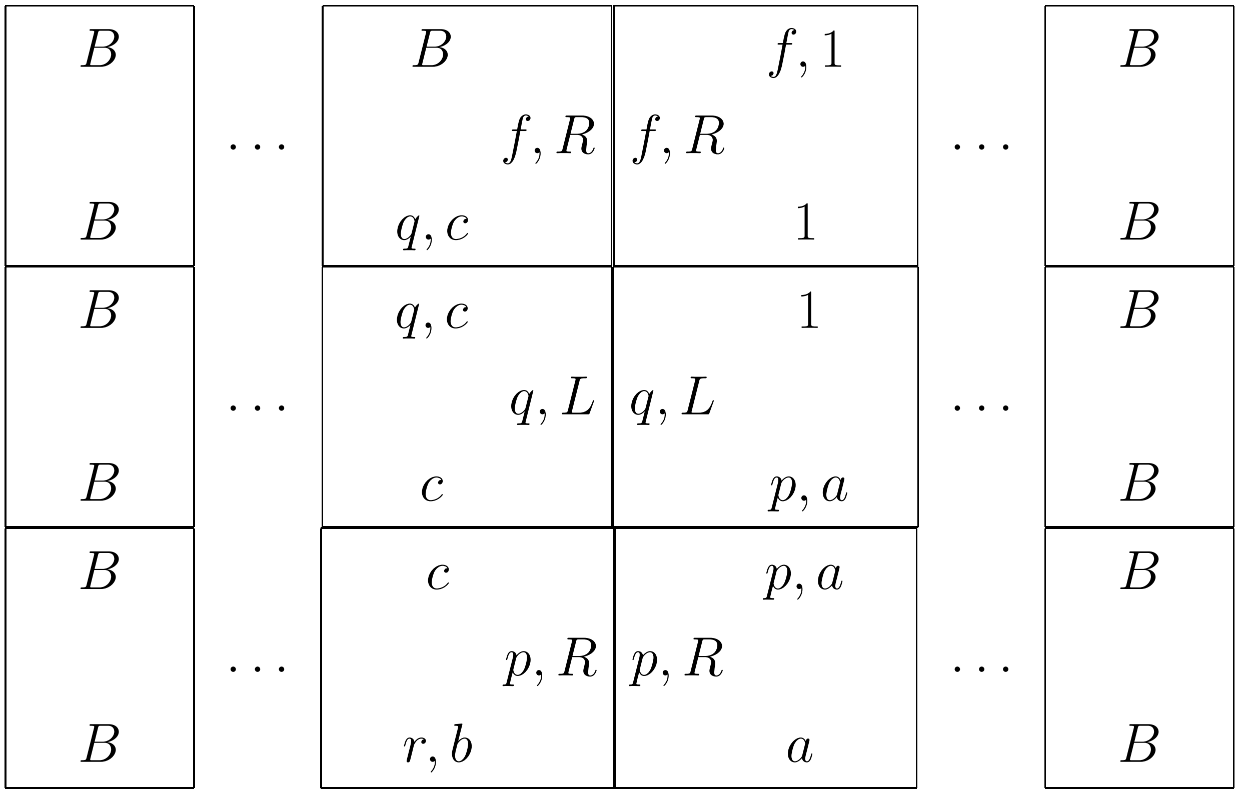
The vertical and the horizontal constraints are that adjacent edges have the same label (or no label).

If $u = u_1 \cdots u_k$, the initial bottom row σ_0, of length $2s$, is:



where the tile labeled q_0, u_1 is in position $s + 1$.

The example below illustrates the construction:



It is not hard to check that $u = u_1 \cdots u_k$ is accepted by M iff the tiling problem just constructed has a solution. This is because $s = p(|u|) + 2$ and the machine runs for at most $p(|u|)$ steps. So the $2s \times s$ rectangle can be tiled iff tiles of type (5) are used iff M accepts u (prints 1). $\square$

Remarks.

(1) The problem becomes harder if we only specify a *single* tile σ_0 as input, instead of a row of length $2s$. If s is specified in binary (or any other base, but not in tally notation), then the $2s^2$ grid has size exponential in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, and this tiling problem is actually $\mathcal{NEXP}$-complete!

(2) If we relax the finiteness condition and require that the entire upper half-plane be tiled, i.e., for every $s > 1$, there is a solution to the $2s \times s$-tiling problem, then the problem is undecidable.

In 1972, Richard Karp published a list of 21 $\mathcal{NP}$-complete problems.

We finally prove the Cook-Levin theorem.

Theorem 9.5. *(Cook, 1971, Levin, 1973) The satisfiability problem* SAT *is $\mathcal{NP}$-complete.*

Proof. We reduce the tiling problem to SAT. Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we introduce boolean variables

$$x_{mnt},$$

for all m with $1 \leq m \leq 2s$, all n with $1 \leq n \leq s$, and all tiles $t \in \mathcal{T}$.

The intuition is that $x_{mnt} = \mathbf{T}$ iff tile t occurs in some tiling σ so that $\sigma(m, n) = t$.

We define the following clauses:

(1) For all m, n in the correct range, as above,

$$(x_{mnt_1} \vee x_{mnt_2} \vee \cdots \vee x_{mnt_p}),$$

for all p tiles in $\mathcal{T}$.

This clause states that every position in σ is tiled.

(2) For any two distinct tiles $t \neq t' \in \mathcal{T}$, for all m, n in the correct range, as above,

$$(\overline{x}_{mnt} \vee \overline{x}_{mnt'}).$$

This clause states that a position may not be occupied by more than one tile.

(3) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m+1\,nt'}).$$

This clause enforces the horizontal adjacency constraints.

(4) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m\,n+1\,t'}).$$

This clause enforces the vertical adjacency constraints.

(5) For all m with $1 \leq m \leq 2s$,

$$(x_{m1\sigma_0(m)}).$$

This clause states that the bottom row is correctly tiled with σ_0.

It is easily checked that the tiling problem has a solution iff the conjunction of the clauses just defined is satisfiable. Thus, SAT is $\mathcal{NP}$-complete. $\square$

We sharpen Theorem 9.5 to prove that 3-SAT is also $\mathcal{NP}$-complete. This is the satisfiability problem for clauses containing at most three literals.

We know that we can't go further and retain $\mathcal{NP}$-completeteness, since 2-SAT is in $\mathcal{P}$.

Theorem 9.6. *(Cook, 1971) The satisfiability problem* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. We have to break "long clauses"

$$C = (L_1 \vee \cdots \vee L_k),$$

i.e., clauses containing $k \geq 4$ literals, into clauses with at most three literals, in such a way that satisfiability is preserved.

For example, consider the following clause with $k = 6$ literals:

$$C = (L1 \vee L2 \vee L3 \vee L4 \vee L5 \vee L6).$$

We create 3 new boolean variables y_1, y_2, y_3, and the 4 clauses

$$\begin{array}{c}(L_1 \vee L_2 \vee y_1),\ (\overline{y_1} \vee L_3 \vee y_2),\\ (\overline{y_2} \vee L_4 \vee y_3),\ (\overline{y}_3 \vee L_5 \vee L_6).\end{array}$$

Let C' be the conjunction of these clauses.

We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable but C is not. If so, in any truth assigment v, $v(L_i) = \mathbf{F}$, for $i = 1, 2, \ldots, 6$.

To satisfy the first clause, we must have $v(y_1) = \mathbf{T}$.

Then to satisfy the second clause, we must have $v(y_2) = \mathbf{T}$, and similarly satisfy the third clause, we must have $v(y_3) = \mathbf{T}$.

However, since $v(L_5) = \mathbf{F}$ and $v(L_6) = \mathbf{F}$, the only way to satisfy the fourth clause is to have $v(y_3) = \mathbf{F}$, contradicting that $v(y_3) = \mathbf{T}$.

Thus, C is indeed satisfiable.

Let us now assume that C is satisfiable. This means that there is a smallest index i such that L_i is satisfied.

Say $i = 1$, so $v(L_1) = \mathbf{T}$. Then if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 2$, so $v(L_1) = \mathbf{F}$ and $v(L_2) = \mathbf{T}$.

Again if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 3$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, and $v(L_3) = \mathbf{T}$.

If we let $v(y_1) = \mathbf{T}$ and $v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 4$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, and $v(L_4) = \mathbf{T}$.

If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 5$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, and $v(L_5) = \mathbf{T}$.

If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Say $i = 6$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, $v(L_5) = \mathbf{F}$, and $v(L_6) = \mathbf{T}$.

Again, if we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Therefore if C is satisfied, then C' is satisfied in all cases.

In general, for every long clause (with $k \geq 4$), create $k-3$ new boolean variables $y_1, \ldots y_{k-3}$, and the $k-2$ clauses

$$\begin{aligned}&(L_1 \lor L_2 \lor y_1), (\overline{y_1} \lor L_3 \lor y_2), (\overline{y_2} \lor L_4 \lor y_3), \cdots,\\ &(\overline{y}_{k-4} \lor L_{k-2} \lor y_{k-3}), (\overline{y}_{k-3} \lor L_{k-1} \lor L_k).\end{aligned}$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable, but that C is not. Then, for every truth assignment v, we have $v(L_i) = \mathbf{F}$, for $i = 1, \ldots, k$.

However, C' is satisfied by some v, and the only way this can happen is that $v(y_1) = \mathbf{T}$, to satisfy the first clause. Then, $v(\overline{y_1}) = \mathbf{F}$, and we must have $v(y_2) = \mathbf{T}$, to satisfy the second clause.

By induction, we must have $v(y_{k-3}) = \mathbf{T}$, to satisfy the next to the last clause. However, the last clause is now false, a contradiction.

Thus, if C' is satisfiable, then so is C.

Conversely, assume that C is satisfiable. If so, there is some truth assignment, v, so that $v(C) = \mathbf{T}$, and thus, there is a smallest index i, with $1 \leq i \leq k$, so that $v(L_i) = \mathbf{T}$ (and so, $v(L_j) = \mathbf{F}$ for all $j < i$).

Let v' be the assignment extending v defined so that

$$v'(y_j) = \mathbf{F} \quad \text{if} \quad \max\{1, i-1\} \leq j \leq k-3,$$

and $v'(y_j) = \mathbf{T}$, otherwise.

It is easily checked that $v'(C') = \mathbf{T}$. $\square$

Another version of 3-SAT can be considered, in which every clause has exactly three literals. We will call this the problem *exact* 3-SAT.

Theorem 9.7. *(Cook, 1971) The satisfiability problem for exact* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. A clause of the form (L) is satisfiable iff the following four clauses are satisfiable:

$$(L \vee u \vee v), (L \vee \overline{u} \vee v), (L \vee u \vee \overline{v}), (L \vee \overline{u} \vee \overline{v}).$$

A clause of the form $(L_1 \vee L_2)$ is satisfiable iff the following two clauses are satisfiable:

$$(L_1 \vee L_2 \vee u), (L_1 \vee L_2 \vee \overline{u}).$$

Thus, we have a reduction of 3-SAT to exact 3-SAT. $\square$

We now make some remarks on the conversion of propositions to CNF.

Recall that the set of propositions (over the connectives $\vee$, $\wedge$, and $\neg$) is defined inductively as follows:

(1) Every propositional letter, $x \in \mathbf{PV}$, is a proposition (an *atomic* proposition).

(2) If A is a proposition, then $\neg A$ is a proposition.

(3) If A and B are propositions, then $(A \vee B)$ is a proposition.

(4) If A and B are propositions, then $(A \wedge B)$ is a proposition.

Two propositions A and B are *equivalent*, denoted $A \equiv B$, if

$$v \models A \quad \text{iff} \quad v \models B$$

for all truth assignments, v.

It is easy to show that $A \equiv B$ iff the proposition

$$(\neg A \vee B) \wedge (\neg B \vee A)$$

is valid.

Every proposition, A, is equivalent to a proposition, A', in CNF.

There are several ways of proving this fact. One method is algebraic, and consists in using the algebraic laws of boolean algebra.

First, one may convert a proposition to *negation normal form*, or *nnf*. A proposition is in nnf if occurrences of $\neg$ only appear in front of propositional variables, but not in front of compound propositions.

Any proposition can be converted to an equivalent one in nnf by using the de Morgan laws:

$$\begin{aligned}\neg(A \vee B) &\equiv (\neg A \wedge \neg B)\\ \neg(A \wedge B) &\equiv (\neg A \vee \neg B)\\ \neg\neg A &\equiv A.\end{aligned}$$

Then, a proposition in nnf can be converted to CNF, but the question of uniqueness of the CNF is a bit tricky.

For example, the proposition

$$A = (u \wedge (x \vee y)) \vee (\neg u \wedge (x \vee y))$$

has

$$\begin{aligned}A_1 &= (u \vee x \vee y) \wedge (\neg u \vee x \vee y)\\ A_2 &= (u \vee \neg u) \wedge (x \vee y)\\ A_3 &= x \vee y,\end{aligned}$$

as equivalent propositions in CNF!

We can get a *unique* CNF equivalent to a given proposition if we do the following:

(1) Let $\mathrm{Var}(A) = \{x_1, \ldots, x_m\}$ be the set of variables occurring in A.

(2) Define a *maxterm w.r.t.* $\mathrm{Var}(A)$ as any disjunction of m pairwise distinct literals formed from $\mathrm{Var}(A)$, and not containing both some variable x_i and its negation $\neg x_i$.

(3) Then, it can be shown that for any proposition A that is not a tautology, there is a *unique* proposition in CNF *equivalent* to A, whose clauses consist of maxterms formed from $\mathrm{Var}(A)$.

The above definition can yield strange results. For instance, the CNF of any unsatisfiable proposition with m distinct variables is the conjunction of all of its 2^m maxterms!

The above notion does not cope well with minimality.

For example, according to the above, the CNF of

$$A = (u \wedge (x \vee y)) \vee (\neg u \wedge (x \vee y))$$

should be

$$A_1 = (u \vee x \vee y) \wedge (\neg u \vee x \vee y).$$

There are also propositions such that any equivalent proposition in CNF has size exponential in terms of the original proposition.

Here is such an example:

$$A = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n}).$$

Observe that it is in DNF.

We will prove a little later that any CNF for A contains 2^n occurrences of variables.

A nice method to convert a proposition in nnf to CNF is to construct a tree whose nodes are labeled with sets of propositions using the following (Gentzen-style) rules :

$$\frac{P, \Delta \qquad Q, \Delta}{(P \wedge Q), \Delta}$$

and

$$\frac{P, Q, \Delta}{(P \vee Q), \Delta}$$

where Δ stands for any set of propositions (even empty), and the comma stands for union. Thus, it is assumed that $(P \wedge Q) \notin \Delta$ in the first case, and that $(P \vee Q) \notin \Delta$ in the second case.

Since we interpret a set, Γ, of propositions as a disjunction, a valuation, v, satisfies Γ iff it satisfies *some* proposition in Γ.

Observe that a valuation v satisfies the conclusion of a rule iff it satisfies both premises in the first case, and the single premise in the second case.

Using these rules, we can build a finite tree whose leaves are labeled with sets of literals.

By the above observation, a valuation v satisfies the proposition labeling the root of the tree iff it satisfies all the propositions labeling the leaves of the tree.

But then, a CNF for the original proposition A (in nnf, at the root of the tree) is the conjunction of the clauses appearing as the leaves of the tree.

We may exclude the clauses that are tautologies, and we may discover in the process that A is a tautology (when all leaves are tautologies).

Going back to our "bad" proposition, A, by induction, we see that any tree for A has 2^n leaves.

However, it should be noted that for any proposition, A, we can construct in polynomial time a formula, A', in CNF, so that A is satisfiable iff A' is satisfiable, by creating *new* variables.

We proceed recursively. The trick is that we replace

$$(C_1 \wedge \cdots \wedge C_m) \vee (D_1 \wedge \cdots \wedge D_n)$$

by

$$(C_1 \vee y) \wedge \cdots \wedge (C_m \vee y) \wedge (D_1 \vee \overline{y}) \wedge \cdots \wedge (D_n \vee \overline{y}),$$

where the C_i's and the D_j's are clauses, and y is a new variable.

It can be shown that the number of new variables required is at most quadratic in the size of A.

Warning: In general, the proposition A' is *not* equivalent to the proposition A.

Rules for dealing for $\neg$ can also be created. In this case, we work with pairs of sets of propositions,

$$\Gamma \rightarrow \Delta,$$

where, the propositions in Γ are interpreted conjunctively, and the propositions in Δ are interpreted disjunctively.

We obtain a sound and complete proof system for propositional logic (a "Gentzen-style" proof system, see Gallier's *Logic for Computer Science*).