Example computation

Consider the decidable language \( A = \{0^n1^n \mid n \geq 0\} \) and the following TM \( M_1 \) deciding \( A \):

\[
M_1 = \text{ "On input string } w:\text{"
}
\]

1. Scan across the tape and reject if a 0 appears after a 1.
2. Repeat as long as both 0-s and 1-s remain on the tape:
   a. Scan across the tape, crossing off a single 0 and a single 1.
   b. If 0-s still remain after all 1-s have been crossed off, or is 1-s still remain after all 0-s have been crossed off reject. Otherwise, if neither 0-s nor 1-s remain of the tape, accept.

Simplifying Conventions

1. The running time of an algorithm is a function of the length of the string representing the input on the algorithm.
2. In worst-case-analysis we consider the longest running time of all inputs of a particular length.
3. In average-case analysis we consider the average of all the running times of inputs of a particular length.

Analyzing an Algorithm

How much time does \( M_1 \) take to decide \( A \)?

It is

- Number of steps the TM has moved (this may depend on several parameters).
Big-O and small-O notation

- The exact running time of an algorithm may be a complex-expression. Therefore usually this is just estimated.
- One convenient form of the estimation is so called asymptotic analysis which determines the running time of the algorithm on large inputs.
- In the asymptotic analysis one may consider only the highest order term of the expression for the running time of the algorithm, disregarding both the coefficient of that term and any lower order terms.
- This is valid because the value of the highest order term dominates the value of other terms on large inputs.

Time Complexity of a TM

Let $M$ be a deterministic Turing machine that halts on all inputs. The running time or time complexity of $M$ is a function $f : \mathcal{N} \to \mathcal{N}$ where $f(n)$ is the maximum number of steps that $M$ 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 Turing machine.
- Customarily $n$ represents the length of the input.

Formally

Let $f$ and $g$ be functions, $f, g : \mathcal{N} \to \mathcal{R}$. We say that $f(n) = \mathcal{O}(g(n))$ if positive integers $c$ and $n_0$ exist such that $\forall n \geq n_0$, $f(n) \leq cg(n)$.

When $f(n) = \mathcal{O}(g(n))$ we say that $g(n)$ is an upper bound for $f(n)$; more precisely, $g(n)$ is an asymptotic upper bound for $f(n)$, which emphasizes the suppression of constant factors.

Examples

Consider the function $f(n) = 6n^3 + 2n^2 + 10n + 100$.
- Disregarding the coefficient 6, we say that $f$ is asymptotically at most $n^3$.
- The asymptotic notation, or big-O notation for describing the estimation defined by $f(n)$ is $f(n) = \mathcal{O}(n^3)$.
**Examples**

1. When we use logarithms, because \( \log_b n = \log_2 n / \log_2 b \), the base needs not be specified. Thus, if \( f_2(n) = 3n \log_2 n + 5n \log_2 (\log_2 n) + 2 \), we have \( f_2(n) = O(n \log n) \).

---

**Intuitively:**

- \( f(n) = O(g(n)) \) means that \( f \) is less than or equal to \( g \) if we disregard differences up to a constant factor.
- One may think at \( O \) as representing a constant difference.
- In practice most functions \( f \) encountered in algorithm analysis have an obvious highest order term \( h(n) \). In that case \( f(n) = O(h(n)) \).

---

**Polynomial and Exponential Bounds**

- Bounds of the form \( n^c \) for \( c > 0 \) are called polynomial.
- Bounds of the form \( a^{cn^d} \), for \( a > 1, c, d > 0 \), are called exponential bounds.

---

**Expressions of big-O**

- Consider \( f(n) = O(n^2) + n \), where each occurrence of \( O \) represents a different suppressed constant. Because \( O(n^2) \) dominates \( O(n) \), \( f(n) = O(n^2) \).
  However, if we also have \( g(n) = O(n^2) \), it cannot be concluded that \( f(n) = O(g(n)) \).
- When \( O \) occurs in the exponent, as in \( f(n) = 2^{O(n)} \), the same idea applies, i.e., \( f(n) = O(2^{cn}) \) for some constant \( c \).
- For expressions of the form \( f(n) = 2^{O(\log n)} \), using the identity \( n = 2^{\log_2 n} \), i.e., \( n^c = 2^{c \log_2 n} \), we can see that \( 2^{O(\log n)} \) represents an upper bound of \( n^c \) for some \( c \).
- Because \( O(1) \) represents a value that is never more than a constant, \( n^c O(1) \) represents the value \( n^c \) for some \( c \).
Examples

Check that:
1. \( \sqrt{n} = o(n) \)
2. \( n = o(n \log(\log(n))) \)
3. \( n \log(\log(n)) = o(n \log(n)) \)
4. \( n \log(n) = o(n^2) \)
5. \( n^2 = o(n) \)
6. \( f(n) \) is never \( o(f(n)) \)

Small-O notation

- Big-O notation says that a function is asymptotically \textit{no more than} another function
- Small-O notation says that a function is asymptotically \textit{less than} another function

Formally: for \( f, g: \mathbb{N} \to \mathbb{R} \), we say that \( f(n) = o(g(n)) \) if \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \). That is, \( f(n) = o(g(n)) \) means that for any real \( c > 0 \) there exists \( n_0 \) such that \( f(n) < cg(n) \) for all \( n > n_0 \).

On input of length \( n \):

Consider each of the three stages separately:
1. In stage 1 the machine scans the tape in \( n \) steps to verify that the input is of the form \( 0^*1^* \); repositioning the tape at the left end uses another \( n \) steps. So, the total number of steps is \( 2n \), i.e., \( O(n) \).
2. Each scan of the tape in stage 2 and 2(a) is performed in \( O(n) \). Because each scan crosses off a 0 and a 1, at most \( n/2 \) scans occur. I.e., the total number of steps is \( (n/2)O(n) = O(n^2) \).
3. In stage 3 the machine makes a single scan to decide whether to accept or reject taking \( O(n) \) steps.

Total number of steps is \( O(n) + O(n^2) + O(n) = O(n^2) \).

Analyzing Algorithms

Consider the TM algorithm \( M_1 \) that decides the language \( A = \{0^n1^n \mid n \geq 0\} \):

\( M_1 = \) "On input string \( w \):
1. Scan across the tape and \textbf{reject} if a 0 is found to the right of a 1
2. Repeat as long as both 0-s and 1-s remain of the tape:
   a. Scan across the tape, crossing off a single 0 and a single 1
3. If 0-s still remain after all 1-s have been crossed off, or is 1-s still remain after all 0-s have been crossed off \textbf{reject}. Otherwise, if neither 0-s nor 1-s remain of the tape, \textbf{accept}."
Let \( t : \mathcal{N} \to \mathcal{N} \) be a function. The time complexity class \( \text{TIME}(t(n)) \) is the collection of languages that are decidable by an \( O(t(n)) \) time TM.

Since language \( A = \{0^n1^n \mid n \geq 0\} \) is decided by \( M_1 \) in \( O(n^2) \) steps, \( A \in \text{TIME}(n^2) \).

\( \text{TIME}(n^2) \) contains all languages decidable in \( O(n^2) \) time.

Is there a TM that decides \( A \) asymptotically faster? I.e., is \( A \) in \( \text{TIME}(t(n)) \) for \( t(n) = o(n^2) \)? One can cross 2 0’s and 2 1’s in stage 2 which cuts the number of scans by half but running time does not change.

Deciding \( A \) in \( O(n) \)

The following two-tape TM \( M_3 \) decides \( A \) in linear time, i.e., in \( O(n) \) time.

\( M_3 = \) "On input string \( w \) on tape 1:
1. Scan across the tape 1 and reject if a 0 is found to the right of a 1
2. Scan across the 0’s on tape 1 until the first 1; at the same time copy the 0’s on tape 2.
3. Scan across the 1’s on tape 1 until the end of the tape is discovered.
   For each 1 read on tape 1, cross off a 0 on tape 2. If all 0’s are crossed off before all 1 are read reject
4. If all 0’s have been crossed off accept; if any 0’s remain reject"

Analysis

1. Stage 1 takes \( O(n) \) steps
2. Stage 2 takes \( (1 + \log_2 n)O(n) = O(n \log_2 n) \) steps
3. Stage 3 takes \( O(n) \) steps

Total number of steps is asymptotically \( O(n \log_2 n) \), that is, \( A \in \text{TIME}(n \log_2 n) \).

Note: any language that can be decided in \( o(n \log n) \) on a single-tape TM is regular; since \( A \) is not regular no faster algorithm performed by a single-tape TM exists to decide it.
Observations

There is an important difference between complexity theory and computability theory:

- The Church-Turing thesis implies that all reasonable models of computation are equivalent (i.e., for each model \( C \) there is an equivalent model \( C' \)).
- In complexity theory the choice of model affects the time complexity of the language decided by that model (i.e., languages decidable by model \( C \) in linear time are not necessarily decidable in linear time by the equivalent model \( C' \)).

Conclusions

- \( M_1 \) using one tape decides \( A \) in \( O(n^2) \)
- \( M_2 \) using one tape decides \( A \) in \( O(n \log n) \)
- \( M_3 \) using two tapes decides \( A \) in \( O(n) \)

Conclusion: time complexity of \( A \) on one-tape TM is \( O(n \log n) \); time complexity of \( A \) on two-tape TM is \( O(n) \)

Theorem 7.8

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.

Proof idea: We know how to convert a multitape TM \( M \) into a single-tape TM \( S \) that simulates \( M \). We show that each step of \( M \) can be simulated by \( S \) in time \( O(t(n)) \). Since there are \( O(t(n)) \) steps performed by \( M \) there will be \( O(t^2(n)) \) steps performed by \( S \)

Complexity Relations

Complexity relations among models show how choice of computational model can affect the time complexity of languages. For that we consider three computation models:

- Single-tape Turing machine
- Multiple-tape Turing machine
- Nondeterministic Turing machine
Proof of Theorem 7.8

For $M$ a $k$-tape TM that runs in $t(n)$ time, the single-tape TM $S$ operates as follows:

1. Initially $S$ puts its tape into the format that represents all the $k$-tapes of $M$ and then simulates $M$
2. To simulate one step of $M$, $S$ scan all the info stored on its tape to determine the symbol under $M$'s heads.
3. Then $S$ makes another pass over its tape to update the tape contents and head positions.
4. If one on $M$'s heads moves rightward onto a previously unread position on its tape, $S$ must increase the amount of space by shifting a portion of its tape one cell to the right.

Simulation time

To simulate each of $M$'s steps, $S$ performs two scans, each using $O(t(n))$ time, i.e., one step of $M$ is simulated by $S$ in $O(t(n))$ time.

Since by assumption $M$ performs $t(n)$ steps, total time taken by $S$ to simulate $M$ is $O(t(n))$ taken by the initial step plus $t(n)$ $O(t(n))$ to perform the simulation, i.e., time complexity of $S$ is $O(t^2(n))$.

Corollary

Let $t(n)$ and $s(n)$ be functions, where $t(n), s(n) \geq n$. If a multitape Turing machine takes $t(n)$ time and $s(n)$ space for any input of length $n$, then there is an equivalent $O(t(n)s(n))$ time single-tape Turing machine.

Analyzing $S$

For each step of $M$, $S$ makes two passes over the active portion of its tape: first to obtain the info necessary to determine the next move and second to carry out the move.

The upper-bound of the length of the active portion of $S$'s tape is the sum of the length of active portions of $M$'s $k$ tapes, which are bounded by $t(n)$, i.e., upper-bound of the active portion of $S$'s tape is $O(t(n))$. 

**Note**

- Definition of the running time of a nondeterministic TM is not intended to correspond to any real-world computing device.
- The running time of a nondeterministic TM is a useful abstraction that assists in characterizing the complexity of an important class of computational problems.

**Non-deterministic TM**

Let \( N \) be a non-deterministic TM that is a decider. The running time of \( N \) is the function \( f : \mathbb{N} \to \mathbb{N} \), where \( f(n) \) is the maximum number of steps that \( N \) makes on any branch of its computation, on any input of length \( n \), as shown here:

\[
\begin{array}{ccc}
\text{Deterministic} & \text{Non-deterministic} & \text{Deterministic} \\
\downarrow & \downarrow & \downarrow \\
\text{accept} & \text{accept} & \text{reject} \\
\end{array}
\]

\[
\begin{array}{ccc}
f(n) & \cdots & f(n) \\
\downarrow & \downarrow & \downarrow \\
\text{accept/reject} & \text{accept} & \text{reject} \\
\end{array}
\]

**Theorem 7.11**

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

**Proof idea:** Let \( N \) be a non-deterministic TM running in time \( t(n) \). We construct a deterministic TM \( D \) that simulates \( N \) by searching \( N \)'s non-deterministic computation tree.
Summary

Let \( t : \mathcal{N} \rightarrow \mathcal{N} \) be a function.

The time complexity class \( \text{TIME}(t(n)) \) is the collection of languages that are decidable by an \( O(t(n)) \) time TM.

The time complexity class \( \text{NTIME}(t(n)) \) is the collection of languages that are decidable by an \( O(t(n)) \) time NTM.

\( \text{TIME}(t(n)) \subset \text{NTIME}(t(n)) \subset \text{TIME}(2^{O(t(n))}) \) for some constant \( c \).

The Simulation

1. Visit all nodes at depth \( d \) of the computation tree before going to the depth \( d + 1 \), starting with the root; total number of nodes is less then twice the number of leaves, i.e., this is bound by \( O(b^t(n)) \).
2. Time for starting from the root and traveling down to a node is \( O(t(n)) \).
3. Therefore the running time of \( D \) is \( O(t(n)b^t(n)) = 2^{O(t(n))} \).

Note: TM \( D \) has three tapes. Converting \( D \) to a single-tape, by previous theorem, the running time at most squares.

Thus we have: \( (2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))} \).

Polynomial time

Note: Polynomial differences in running time are considered to be small whereas exponential differences are considered to be large.

Example: Consider the difference between growth rate of polynomial (typically \( n^3 \)) and exponential (typically \( 2^n \))

\( n = 1000, n^3 = 1,000,000,000 \) a large but manageable number
\( n = 1000, 2^n \) is a number larger than the number of atoms in the universe, i.e., an unmanageable number.

Conclusion: polynomial time algorithms are fast enough for many purposes; exponential time algorithms are rarely useful.

Summary

Theorems 7.8 and 7.11 illustrate important distinction among models of TM:

- There is at most a polynomial difference between the time complexity of problems measured on deterministic single-tape and multitape TM.
- There is at most an exponential difference between the time complexity of problems measured on deterministic and nondeterministic TM.
Time complexity theory

The aim of a time complexity theory is to present fundamental properties of computation rather than properties of Turing machines or any other particular model. Therefore we focus on computations that are unaffected by polynomial differences in running time which are considered insignificant and thus are ignored. This allows us to develop a theory that doesn’t depend on the selection of a particular model of computation.

Source of complexity

- **Brute-force search**: exponential time algorithms that solve problems by exhaustively searching through a space of solutions.
- **Example**: factoring a number into its constituent primes by searching through all potential divisors.
- **Note**: sometime brute-force search may be avoided through a deeper understanding of the problem, which may reveal polynomial time algorithms.

Note

- When a problem is in $\mathcal{P}$ we have a method to solve it in time $n^k$ for some constant $k$. Whether $n^k$ is practical depends on $k$ and on application.
- **Example**: running time $n^{100}$ is unlikely to be of any practical use.
- But calling polynomial time the threshold of practical solvability has proven to be useful.
- Once a polynomial time has been found for a problem that required exponential time some key insight have been gained and further reduction in its time complexity usually follows.

Class $\mathcal{P}$

$\mathcal{P}$ is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine, i.e.,
\[ \mathcal{P} = \bigcup_k \text{TIME}(n^k) \]
Class $\mathcal{P}$ plays a central role in time complexity theory because:

1. $\mathcal{P}$ is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM, i.e., $\mathcal{P}$ is mathematically robust.
2. $\mathcal{P}$ roughly corresponds to the class of problems that are realistically solvable by computers, i.e., $\mathcal{P}$ is relevant from a practical standpoint.
Notational conventions

- We describe algorithms using numbered stages where a stage is analogous to a high-level step of a Turing machine, or a sequence of simple steps of a Turing machine.
- When we analyze an algorithm we need to do two things:
  1. give a polynomial upper bound (using \( O \) notation) on the number of stages that the algorithm uses when it run on an input of length \( n \).
  2. examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a deterministic model.
- When both tasks have been completed we can conclude that the algorithm runs in polynomial time because composition of polynomials is a polynomial.

Graph encodings

- Lists of nodes and edges
- Adjacent matrix \( M \) where \( M(i, j) = 1 \) if there is an edge from node \( i \) to node \( j \) and \( M(i, j) = 0 \) otherwise
- Running time of graph algorithms may be computed in number of nodes instead of the size of graph representation because graph representation is polynomial in number of nodes.

Notations

- We use the notation \( ⟨ \cdot ⟩ \) to indicate a reasonable encoding of one or more objects into a string, without specifying any particular encoding method.
- A reasonable method is one that allows for polynomial time encoding and decoding of objects into internal representations of computation model; familiar encoding methods for graphs, automata, etc., are reasonable.
- Unreasonable encoding method are those that generate exponential large representations, such as using unary strings \( \ldots 111 \ldots \) to encode natural numbers; instead, base \( k \) notation for \( k \geq 2 \) should be used.

The PATH problem

\[
\text{PATH} = \{(G, s, t) \mid \text{G is a direct graph that has a direct path from } s \text{ to } t\}
\]

Theorem 7.14 \( \text{PATH} \in \mathbb{P} \)

Proof idea: construct a polynomial time algorithm that decides \( \text{PATH} \).

Note: the brute-force algorithm that examines all potential paths in \( G \) and determine whether any is a direct path from \( s \) to \( t \) is not fast enough.
Proof

A polynomial time algorithm $M$ for $PATH$ follows:

$M =$ "On input $(G, s, t)$ where $G$ is a direct graph with nodes $s$ and $t$:
1. Place a mark on node $s$
2. Repeat until no additional nodes are marked:
   (a) Scan all the edges of $G$. If an edge $(a, b)$ is found going from a
       marked node $a$ to an unmarked node $b$, mark node $b$.
3. If $t$ is marked accept. Otherwise reject."

Brute-forth algorithm for $PATH$

1. A potential path is a sequence of nodes in $G$ having a length at most $m$ where $m$ is the number of nodes in $G$.
2. If a direct path exist from $s$ to $t$, one having a length at most $m$ exists because repeating a node is never necessary.
3. The number of potential paths is $O(k^m)$, which is exponential in number of nodes, where $k$ is the maximum number of successors.

Conclusion: to get a polynomial algorithm that decides $PATH$ we need to avoid the brute-forth approach.

Another example

$RELPRIME = \{ (x, y) \mid x$ and $y$ are relative prime $\}$

For $x, y \in \mathbb{N}$, $x$ and $y$ are relative prime if 1 is the largest integer that evenly divide $x$ and $y$.

Theorem 7.15 $RELPRIME \in \mathbf{P}$

Analyzing $M$

- Stages 1 and 3 are executed only once, hence they are bound by $O(m)$.
- Stage 2 runs at most $m$ times because each time except the last it marks an additional node of $G$. Hence, it is bound by $O(m)$
- The total time is bounded by $2O(m) + O(m) = O(m)$
A better idea

Use Euclidean algorithm that determines the greatest common divisor of two numbers, \(\gcd(x, y)\), to solve this problem. Then if \(\gcd(x, y) = 1\) accept, otherwise reject.

Euclidean algorithm, \(E\):
\[
E = \text{"On input } \langle x, y \rangle \text{ where } x, y \in \mathbb{N}:
\]
1. Repeat until \(y = 0\)
   (a) Let \(x := x \mod y\)
   (b) Exchange \(x\) with \(y\)
2. Output \(x\)."

Proof idea

- Brute-force algorithm to solve RELPRIME: search through all possible divisors of \(x\) and \(y\) and accept if none is greater than 1
- The magnitude of a number represented in any base \(k \geq 2\) is exponential in its representation base.
- That is, brute-force search is an exponential running time algorithm

Analyzing \(E\)

- Every execution of 1(a) (except possible first) cuts the value of \(x\) by at least half because \(x \mod y = \text{remainder}(x, y)\) and \(y \geq 2\)
- Since \(\text{remainder}(x, y) < y\) after stage 1(b) \(x < y\)
- The values of \(x\) and \(y\) are reduced by at least half every time through the loop 1
- The maximum number of times loop 1 is executed is less than \(2 \log_2 x\) and \(2 \log_2 y\). These logarithms are proportional to the length \(n\) of representations.

Thus the number of times loop executes is bounded by \(O(n)\) and \(E\) is polynomial.

Algorithm solving RELPRIME

The following algorithm \(R\) solves RELPRIME using \(E\):
\[
R = \text{"On input } \langle x, y \rangle \text{ where } x, y \in \mathbb{N}:
\]
1. Run \(E\) on \(\langle x, y \rangle\)
2. If \(E\) returns 1 accept. Otherwise reject."

Note: clearly if \(E\) runs in polynomial time so does \(R\) hence we only need to analyze \(E\) for time complexity and correctness.
Why Complexity Theory

Complexities of many problems are linked! The discovery of a polynomial time algorithm for one such problem can be used to solve an entire class of problems.

Question

Why are we unsuccessful in finding polynomial time algorithms for some problems?

Possible answers:
- Perhaps such problems have, as yet undiscovered, polynomial time algorithm that rest on unknown principles.
- Some of such problems simply cannot be solved in polynomial time. They may be intrinsically difficult.

A Hamiltonian path

A graph that contains a Hamiltonian path between nodes $s$ and $t$.

Example

Hamiltonian path problem:
- A Hamiltonian path in a directed graph $G$ is a path that goes through each node of $G$ exactly once.
- Hamiltonian path problem consists of testing whether a directed Graph $G$ contains a Hamiltonian path connecting two specified nodes.
- $HAMPATH = \{(G, s, t) \mid G$ is a directed graph with a Hamiltonian path from $s$ to $t \}$
### Polynomial Verifiability

The *HAMPATH* problem has a feature called *polynomial verifiability* which is important for understanding its complexity:

*If a Hamiltonian path in a graph \( G \) is discovered (somehow) we could easily convince someone else of its existence, simply by presenting it!*

That is, verifying the existence of a Hamiltonian graph may be much easier than determining its existence.

### Verifier

A verifier for a language \( A \) is an algorithm \( V \) where:

\[
A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}.
\]

**Note:**
- Time of the verifier is measured only in terms of length of \( w \). A polynomial time verifier runs in polynomial time in the length of \( w \).
- A language \( A \) is polynomially verifiable if it has a polynomial time verifier.

### Note

Some problems may not be polynomially verifiable. For example *HAMPATH*, the complement of *HAMPATH*, is not polynomial time verifiable.

**Rationale:** Even if we could determine that a graph did not have a Hamiltonian path we don’t known a way for verifying its non-existence without using the same exponential time algorithm that determined its nonexistence!
**The Class NP**

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

**Note:**
- The term **NP** comes from nondeterministic polynomial time and is derived from an alternative characterization using nondeterministic polynomial time Turing machines.
- **NP** class is important because it contains many problems of practical interest.
- **HAMPATH, COMPOSITES** ∈ **NP**. Note that **COMPOSITES** ∈ **P** but proving it is more difficult.

**Observations**

- A verifier uses additional information, represented by $c$ in Verifier's definition.
- This info is called a certificate or proof, of membership.
- For polynomial verifiers the certificate has a polynomial length (in the length of $w$).
- **Examples:** the certificate for $(G, s, t) \in HAMPATH$ is a Hamiltonian path between $s$ and $t$; the certificate for $x \in COMPOSITES$ is a divisor $p$ of $x$.

**Theorem 7.20**

A language is in **NP** iff it is decided by some nondeterministic polynomial time Turing machine.

**Proof idea:**
- We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa.
- The verifier simulates the NTM by using the accepting branches as certificates.

**A NTM deciding HAMPATH**

$N_1 = \"$On input $(G, s, t)$ where $G$ is direct graph with nodes $s, t$:
1. Write a list of $m$ numbers, $p_1, p_2, \ldots, p_m$ where $m$ is the number of nodes in $G$. Each number is nondeterministically selected between 1 and $m$.
2. Check for repetitions in the list. If any are found reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fails reject
4. For each $i, 1 \leq i \leq m$, check whether $(p_i, p_{i+1})$ is an edge of $G$. If any are not, reject. Otherwise accept.\"$
Proof, continuation

Assume that $A$ is decided by a polynomial time NTM $N$ and construct a polynomial verifier $V_N$ that decides $A$.

$V_N =$ "On input $(w, c)$ where $w$ and $c$ are strings:
1. Simulates $N$ on input $w$, using each symbol of $c$ as a description of the nondeterministic choice to make at each step (see NTM computation simulation).
2. If this branch of $N$'s computation accepts, accept; otherwise reject"

Proof

$A \in \text{NP} \Rightarrow A$ is decidable by a polynomial time NTM:

Assume that $V$ is polynomial time verifier deciding $A$ in time $n^k$. The NTM $N_V$ equivalent to $V$ works as follows:

$N_V =$ "On input $w$ of length $n$:
1. Nondeterministically select a string $c$ of length at most $n^k$.
2. Run $V$ on input $(w, c)$
3. If $V$ accepts, accept; if $V$ rejects rejects."

Observations

- The class $\text{NP}$ is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomial equivalent
- When describing and analyzing nondeterministic polynomial time algorithm we follow the notational conventions set up for deterministic polynomial time algorithms
- Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model.
- Algorithm analysis shows that every branch uses at most polynomially many stages

Class $\text{NTIME}(t(n))$

The nondeterministic time complexity class $\text{NTIME}(t(n))$ is defined by:

$\text{NTIME}(t(n)) = \{ L \mid L$ is a language decide by an $O(t(n))$ NTM $\}$

Corollary: $\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

Proof: obvious from previous considerations
A clique in an undirected graph is a subgraph wherein every two nodes are connected by an edge. A $k$-clique is a clique containing $k$ nodes.

Clique problem:

$$\text{CLIQUE} = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}$$

**Theorem 7.24**

$$\text{CLIQUE} \in \text{NP}$$

**Proof idea:** the clique is the certificate.

The following is a verifier $V$ for CLIQUE:

$$V = \langle \langle G, k \rangle, c \rangle :$$

1. Test whether $c$ is a set of $k$ nodes in $G$.
2. Test whether $G$ contains all edges connecting nodes in $c$.
3. If both pass, accept; otherwise reject.
Theorem 7.25

SUBSET-SUM ∈ NP.

Proof idea: the subset is the certificate
The following is a verifier $V$ for SUBSET-SUM:

$V = \text{"On input } \langle \langle S, t \rangle, c \rangle:\$

1. Test whether $C$ is collection of numbers that sum to $t$
2. Test whether $S$ contains all the numbers in $C$.
3. If both pass, accept; otherwise, reject."

SUBSET-SUM Problem

A collection of $k$ integers $x_1, x_2, \ldots, x_k$ and a target number $t$ are given. We want to determine whether this collection contains a subcollection that adds up to $t$

$\text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{x_1, x_2, \ldots, x_k\} \text{ and for some } C = \{y_1, y_2, \ldots, y_l\} \subseteq S \text{ we have } \sum_{i=1}^{l} y_i = t \}$

Example: $\langle \{4, 11, 16, 21, 27\}, 25\rangle \in \text{SUBSET-SUM}$ because $4 + 21 = 25$.

Note: $\{x_1, x_2, \ldots, x_k\}$ and $\{y_1, y_2, \ldots, y_l\}$ may be multisets.

Observations

- It seems that verifying that something is not present is more difficult that that it is present.
- Hence, CLIQUE and SUBSET-SUM are not obviously members of NP.
- A separate complexity class denoted coNP contains languages that are complements of languages in NP class.
- We don’t know whether NP is different from coNP.

Alternative proof

We can also prove this theorem given the NTM $N$ that decides SUBSET-SUM:

$N = \text{"On input } \langle S, t \rangle:\$

1. Nondeterministically select a subset $c$ of numbers in $S$
2. Test whether $c$ is a collection of numbers that sum to $t$
3. If the test passes, accept; otherwise, reject."
Observations

- If $P = \text{NP}$ then any polynomial verifiable problem would be polynomially decidable.
- Most researchers believe that $P \neq \text{NP}$ because people have invested enormous effort to find polynomial time algorithms for some problems in $\text{NP}$ without success.
- A proof that $P \neq \text{NP}$ would mean that no fast algorithm exists to replace brute-force search for some problems. This may be beyond scientific reach.
- Best method known for solving problems in $\text{NP}$ deterministically is based on deterministic simulation of NTM, which is exponential, i.e., $NP \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{nk})$. But we don’t know whether $\text{NP}$ is contained in a smaller deterministic time complexity class.

The P versus NP question

- $P$: the class of languages for which membership can be decided quickly
- $\text{NP}$: the class of languages for which membership can be verified quickly

Question: is $P = \text{NP}$?

This is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics.

Theoretical Importance

- Research trying to show that $P \neq \text{NP}$ may focus on an $\text{NP}$-complete problem.
- If any problem in $\text{NP}$ requires more than polynomial time, an $\text{NP}$-complete does.
- Research trying to show that $P = \text{NP}$ only need to find a polynomial time algorithm for one $\text{NP}$-complete problem.

Advance on the P versus NP

- Certain problems in $\text{NP}$ have their individual complexity related to that of the entire class (Stephen Cook and Leonid Levin, 1970s)
- If a polynomial time algorithm exists for any of these problems, all problems in $\text{NP}$ class would be polynomial time solvable.
- These problems are called $\text{NP}$-complete and the phenomenon of $\text{NP}$-completeness is important for both theoretical and practical reasons.
Example NP-complete problem

- A Boolean formula is an expression involving Boolean variables and operations. For example, \( \Phi = (\bar{x} \land y) \lor (x \land \bar{z}) \) is a Boolean formula.
- A Boolean formula is satisfiable if some assignments of 1 and 0 (true and false) to its variables makes the formula evaluates to 1. For example, the assignment \( x = 0, y = 1, z = 0 \) makes \( \Phi \) evaluate to 1.
- The satisfiability problem (SAT) is to test whether a Boolean formula is satisfied, i.e., \( SAT = \{ \langle \Phi \rangle \mid \Phi \text{ is a satisfiable boolean formula} \} \).

Practical Importance

- The phenomenon of NP-completeness may prevent wasting time searching for a non-existent polynomial time algorithm to solve a particular problem.
- Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, showing that it is NP-complete will suffice, because we believe that \( P \neq NP \).

Comments on reducibility

- When problem \( A \) reduces to problem \( B \) a solution to \( B \) can be used to solve \( A \).
- Polynomial time reducibility is a reducibility method that takes the efficiency of computation into account.
- When problem \( A \) is efficiently reducible to problem \( B \), an efficient solution to \( B \) can be used to solve \( A \) efficiently.

Cook-Levin Theorem

\( SAT \in P \) iff \( P = NP \).

Proof idea: Use polynomial time reducibility method.
Polynomial time computability

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

Note: $f$ is called the polynomial time reduction of $A$ to $B$.

Polynomial time reducibility

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

Observations

A polynomial time reduction of $A$ to $B$ provides an efficient way of converting membership testing in $A$ to membership testing in $B$.

To test whether $w \in A$ we use the reduction $f$ to map $w$ into $f(w)$ and test $f(w) \in B$.

If language $A$ is polynomial reducible to language $B$ which has a polynomial time solution then $A$ has a polynomial time solution.

Theorem 7.31

If $A \leq_P B$ and $B \in P$ then $A \in P$.

Proof: Let $M$ be a polynomial time algorithm deciding $B$ and $f$ be a polynomial time reduction from $A$ to $B$. Then the algorithm $N$ is a polynomial time decider of $A$:

$N =$ "On Input $w$:
1. Compute $f(w)$
2. Run $M$ on input $f(w)$ and output whatever $M$ outputs."

Note: Since composition of two polynomial is polynomial, $N$ is a polynomial time algorithm.
Definition of NP-Completeness

A language $B$ is NP-complete if it satisfies two conditions:
1. $B \in \text{NP}$
2. Every $A \in \text{NP}$ is polynomial time reducible to $B$

Theorem 7.35
If $B$ is NP-complete and $B \in P$ then $P = \text{NP}$.

Proof: this follows directly from the definition of $\text{NP}$-completeness.

The Cook-Levin Theorem

Theorem 7.37 $\text{SAT}$ is NP-complete

Proof idea:
- Show that $\text{SAT} \in \text{NP}$, which is easy
- Show that any language $A \in \text{NP}$ is polynomial time reducible to $\text{SAT}$
- The reduction of $A$ takes an NTM $N$ and a string $w$ and produces a Boolean formula $\Phi$ that simulates the NTM $N$ that decides $A$ operating on $w$.
- If $N$ accepts, $\Phi$ has a satisfying assignment that corresponds to that computation; if $N$ doesn’t accept, no assignment satisfies $\Phi$. Hence, $w \in A$ iff $\Phi$ is satisfiable.

Theorem 7.36
If $B$ is $\text{NP}$-complete and $B \leq_p C$ for $C \in \text{NP}$ then $C$ is $\text{NP}$-complete.

Proof: Since $C \in \text{NP}$ we only need to show that every $A \in \text{NP}$ is polynomial time reducible to $C$.
- Since $B$ is $\text{NP}$-complete $A$ is polynomial time reducible to $B$
- Since $B$ is polynomial time reducible to $C$, $A$ is polynomial time reducible to $C$ by first reducing it to $B$ and then reducing its image to $C$.
- Hence, every language in $\text{NP}$ is polynomial time reducible to $C$.
**Proof**

1. \( SAT \in \text{NP} \): a nondeterministic polynomial time machine can guess an assignment to the variables of a given formula \( \Phi \) and accept if assignment satisfies \( \Phi \).

2. Let \( A \in \text{NP} \): show that \( A \) is polynomial time reducible to \( SAT \). For a NTM \( N \) that decides \( A \) in \( n^k \) time for some constant \( k \), construct a formula \( \Phi \) that simulates \( N \).

3. Construction of \( \Phi \): based on organizing the computation performed by \( N \) into an \( n^k \times n^k \) tableau as seen in the next slide.

---

**Assignment-tableau correspondence**

The assignment turns on exactly one variable for each cell, using the following constructs:

1. at least one variable that is associated with a cell is on, by: \( \bigvee_{s \in \mathcal{C}} x_{i,j,s} \)
2. no more than one symbol in each cell, by: \( \bigwedge_{s, t \in \mathcal{C}, s \neq t} (x_{i,j,s} \lor x_{i,j,t}) \)

Thus, any satisfying assignment specifies one symbol in each cell by:

\[
\Phi_{\text{cell}} = \bigwedge_{1 \leq i, j \leq n^k} \left( \bigvee_{s \in \mathcal{C}} x_{i,j,s} \right) \land \left( \bigwedge_{s \neq t \in \mathcal{C}} \left( x_{i,j,s} \lor x_{i,j,t} \right) \right)
\]

---

**Configurations tableau of \( N \)**

Observations: (1) Each configuration starts and ends with a # symbol.

(2) A tableau is accepting if any of its rows is an accepting configuration.
**Legal windows**

A $2 \times 3$ window of cells is legal if that window does not violate the actions specified by $N$'s transition function. To explain, consider the transitions:

$$\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}.$$ 

Examples of legal windows for this machine are:

(a) \[
\begin{array}{c|c}
q & \# \\
\hline
x & b \\
q & c \\
\end{array}
\]

(b) \[
\begin{array}{c|c}
q & \# \\
\hline
x & a \\
q & x \\
\end{array}
\]

(c) \[
\begin{array}{c|c}
q & \# \\
\hline
x & y \\
q & x \\
\end{array}
\]

(d) \[
\begin{array}{c|c}
q & \# \\
\hline
y & x \\
q & y \\
\end{array}
\]

(e) \[
\begin{array}{c|c}
q & \# \\
\hline
y & x \\
q & a \\
\end{array}
\]

(f) \[
\begin{array}{c|c}
q & \# \\
\hline
y & x \\
q & b \\
\end{array}
\]

**An accepting tableau**

is specified by $\Phi_{\text{start}}$, $\Phi_{\text{move}}$, $\Phi_{\text{accept}}$

- $\Phi_{\text{start}}$ ensures that the first row of the tableau is the starting configuration of $N$ on $w$ by the equality:

$$\Phi_{\text{start}} = x_{1,1} \# \land x_{1,2, q_0} \land x_{1,3, w_1} \land \ldots \land x_{1, n+3, w_n} \land \ldots \land x_{1, n+k, q_0} \land \ldots$$

- $\Phi_{\text{accept}}$ guarantees that an accepting configuration occurs in the tableau by placing $q_a$ in one of the cells by: $\Phi_{\text{accept}} = \bigvee_{1 \leq i, j \leq n^k} x_{i,j,q_a}$

- $\Phi_{\text{move}}$ guarantees that each row of the tableau correspond to a configuration that legally follows the preceding row's configuration according the $N$'s transition rules.

**Claim**

If the top row of the tableau is the start configuration and every window is legal then each row is a configuration that legally follows the configuration represented by the preceding row.

**Proof:** show the claim for any two adjacent configurations.

**Illegal windows**

- (a) is illegal because the central symbol on the top can’t be changed because it has no adjacent state
- (b) is illegal because the transition specifies that $b$ get changed to $c$ not to $a$
- (c) is illegal because two states appear in the bottom row.
Complexity of the reduction

- Tableau is $n^k \times n^k$ and thus contains $n^{2k}$ cells; each cell has $|C| = 1$ variables associated with it where $l$ depends only on $N$. Hence total number of variables is $O(n^{2k})$.
- Estimating the size of four components of $\Phi$: $\Phi_{cell}$ is a fixed fragment of $\Phi$ so its size is fixed and is $O(n^{2k})$; $\Phi_{start}$ has the size $O(n^k)$; $\Phi_{move}$ and $\Phi_{accept}$ have sizes $O(n^k)$. Hence total size of $\Phi$ is $O(n^{2k})$, i.e., size of $\Phi$ is polynomial in $n$.
- Each component of $\Phi$ can be produced in polynomial time. Therefore we conclude that we can construct a reduction that produces $\Phi$ from $w$ in polynomial time.

This concludes the proof of Cook-Levin Theorem, showing that $SAT$ is NP-complete.

Construction of $\Phi_{move}$

- $\Phi_{move}$ stipulates that all windows in the table are legal.
- Each window contains six cells which may be set in a fixed number of ways to yield a legal window. $\Phi_{move}$ says that the setting of those six cells is done this way by:

$$\Phi_{move} = \bigwedge_{1 \leq i,j \leq n} (\text{window}[i,j] \text{ is legal})$$

- Replace "window[i,j] is legal" with the following formula where $a_1, a_2, a_3, a_4, a_5, a_6$ are the contents of the six cells:

$$\vee_{a_1 \ldots a_6 \text{ legal}} (x_{i,j-1} \land x_{i,j}\land x_{i,j+1}\land x_{i+1,j-1}\land x_{i+1,j}\land x_{i+1,j+1})$$

$3SAT$ is NP-Complete

**Proof:** provide a polynomial time reduction from $SAT$ to $3SAT$.
- Transform first $\Phi_{SAT}$ into CNF.
- Represent each component of the CNF $(a_1 \lor a_2 \lor \ldots \lor a_n)$ by $n-2$ clauses: $(a_1 \lor a_2 \lor z_1) \land (z_1 \lor a_3 \lor z_2) \land \ldots \land (z_{n-3} \lor a_{n-1} \lor a_n)$

Conjunctive Normal Form

- **Literal:** a Boolean variable or a negated Boolean variable.
  Examples: $x$ and $\overline{x}$ are literals
- **Clause:** several literals connected with $\lor$.
  Example: $(x_1 \lor \overline{x_2} \lor x_3 \lor x_4)$ is a clause
- **Conjunctive normal form:** a Boolean formula which is a conjunction of several clauses (i.e., connected with $\land$).
  Example: $(x_1 \lor \overline{x_2} \lor x_3 \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor x_6)$ is a CNF-formula
- **3CNF-formula:** a CNF-formula where each clause has three literals.
  Example: $(x_1 \lor \overline{x_2} \lor x_3) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor x_6 \lor x_4) \land (x_4 \lor x_5 \lor x_6)$
The 3SAT problem is polynomial time reducible to CLIQUE, where CLIQUE = \{⟨G, k⟩ | G is an undirected graph with a k-clique \}.

Proof idea:
- The polynomial time reduction $f$ that we demonstrate from 3SAT to CLIQUE converts formulas to graphs.
- In the constructed graphs, cliques of a specialized size correspond to satisfying assignments of the formula.
- Structures within the graph are designed to mimic the behavior of the variables and clauses.

Nodes of $G$

- Nodes in $G$ are organized in $k$ groups of three nodes each called the triplets $t_1, t_2, \ldots, t_k$.
- Each triple corresponds to one of the clauses in $\Phi$ and each node in the triplet corresponds to a literal in the associated clause.
- Label each node of $G$ with its corresponding literal in $\Phi$.

Other NP-complete languages

To show that $A$ is NP-complete, we provide a polynomial time reduction from 3SAT (or any other NP-complete language) to $A$.

Proof

Let $\Phi$ be a formula with $k$ clauses:

$$\Phi = (a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land \ldots \land (a_k \lor b_k \lor c_k)$$

The reduction $f$ generates the string $⟨G, k⟩$ where $G$ is an undirected graph.
From 3SAT to Clique

3SAT formula $\Phi = (x_1 \lor x_1 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_2 \lor x_2)$ transformed in the graph:

Edges of $G$

Edges of $G$ connect all but two types of pairs of nodes in $G$:
- No edge is present between the nodes of the same triple.
- No edge is present between two nodes with contradictory labels, such as $x_2$ and $\bar{x}_2$.

Proof, continuation

$k$-clique in $G$ implies $\Phi$ is satisfiable. Suppose $G$ has a $k$-clique:
- No two of the clique nodes can occur in the same triplet because nodes in the same triplet are not connected. Therefore each the $k$-triplets contains contains one of the $k$-clique nodes.
- Assign truth value to the variables of $\Phi$ so that each literal labeling a clique node is made true. This is possible because two nodes with contradictory labels are not connected.
- This assignment satisfies the formula $\Phi$. Since each node corresponds to a clause that has a true value in it the clause is true; since classes are connected by $\land$ the formula is true.

Why this construction works?

$\Phi$ is satisfiable iff $G$ has a $k$-clique.

Proof: Suppose that $\Phi$ has a satisfying assignment.
- At least one literal is true in every clause (required by $\lor$)
- The nodes of $G$ are grouped into triples. In each triple of $G$ we select one node corresponding to a true literal in the satisfying assignment. If more literals are true in some clause we select the true literal arbitrarily.
- The nodes just selected form a $k$-clique; number of nodes is $k$ (there are $k$ clauses in $\Phi$) and each pair of selected nodes are joined, by construction.
- Selected node are not from the same triplet, by construction; they could not have contradictory labels because otherwise the associated labels would be both true in the satisfying assignment. Hence $G$ contains a $k$-clique.
**Definition of NP-Completeness**

A language $B$ is NP-complete if it satisfies two conditions:
1. $B \in \text{NP}$
2. Every $A \in \text{NP}$ is polynomial time reducible to $B$

**Theorem 7.35** If $B$ is NP-complete and $B \in P$ then $P = \text{NP}$.

**Conclusions**

- If CLIQUE is solvable in polynomial time, so is 3SAT.
- Polynomial time reducibility allows us to link these two very different problems.
- Similar link may be made among other problems.

**The Cook-Levin Theorem**

**Theorem 7.37** SAT is NP-complete.

**Proof idea:**
- Show that SAT $\in \text{NP}$, which is easy.
- Show that any language $A \in \text{NP}$ is polynomial time reducible to SAT.
- The reduction of $A$ takes a string $w$ and produces a Boolean formula $\Phi$ that simulates the NTM $N$ that decides $A$ operating on $w$.
- If $N$ accepts, $\Phi$ has a satisfying assignment that correspond to that computation; if $N$ doesn’t accept, no assignment satisfies $\Phi$. Hence, $w \in A$ iff $\Phi$ is satisfiable.

**Theorem 7.36**

If $B$ is NP-complete and $B \leq_P C$ for $C \in \text{NP}$ then $C$ is NP-complete.

**Proof:** Since $C \in \text{NP}$ we only need to show that every $A \in \text{NP}$ is polynomial time reducible to $C$.
- Since $B$ is NP-complete $A$ is polynomial time reducible to $B$.
- Since $B$ is polynomial time reducible to $C$, $A$ is polynomial time reducible to $C$ by first reducing it to $B$ and then reducing its image to $C$.
- Hence, every language in NP is polynomial time reducible to $C$. 
**Proof**

1. \( SAT \in NP \): a nondeterministic polynomial time machine can guess an assignment to the variables of a given formula \( \Phi \) and accept if assignment satisfies \( \Phi \).

2. Let \( A \in NP \): show that \( A \) is polynomial time reducible to \( SAT \). For a NTM \( N \) that decides \( A \) in \( n^k \) time for some constant \( k \), construct a formula \( \Phi \) that simulates \( N \).

   - Construction of \( \Phi \): based on organizing the computation performed by \( N \) into an \( n^k \times n^k \) tableau as seen in the next slide.

**Observations:**

1. Each configuration starts and ends with a \# symbol.
2. A tableau is **accepting** if any of its rows is an accepting configuration.

**Polynomial time \( f : A \to SAT \)**

- On input \( w \), \( f \) produces \( \Phi_w \).
- **Variables of \( \Phi_w \):** Let \( N = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r) \), and \( C = Q \cup \Gamma \cup \{\#\} \).
  - For each \( 1 \leq i, j \leq n^k \land s \in C \) we have a variable \( x_{i,j,s} \) in \( \Phi_w \).
- **Cells:** of the \((n^k)^2\) entries of a tableau is called a cell. \( \forall s \in C \)
  - \( x_{i,j,s} = 1 \) if cell \([i,j]\) = \( s \).
- **Formula \( \Phi_w \):** \( \Phi_w = \Phi_{cell} \land \Phi_{start} \land \Phi_{move} \land \Phi_{accept} \).

**Tableau(\( N,w \))**

- Every accepting tableau for \( N \) and \( w \) correspond to an accepting computation branch of \( N \) on \( w \).
- Problem of determining whether \( N \) accepts \( w \) is equivalent to the problem of determining whether an accepting tableau for \( N \) and \( w \) exists.
Assignment-tableau correspondence

The assignment turns on exactly one variable for each cell, using the following constructs:

1. at least one variable that is associated with a cell is on, by: $\bigvee_{s \in C} x_{i,j,s}$
2. no more than one variable is on for each cell, by: $\bigwedge_{s \neq t \in C} (x_{i,j,s} \lor x_{i,j,t})$

Thus, any satisfying assignment specifies one symbol in each cell by:

$$\Phi_{cell} = \bigwedge_{1 \leq i,j \leq n} \left[ (\bigvee_{s \in C} x_{i,j,s}) \land \left( \bigwedge_{s \neq t \in C} (x_{i,j,s} \lor x_{i,j,t}) \right) \right]$$

An accepting tableau

is specified by $\Phi_{start}$, $\Phi_{move}$, $\Phi_{accept}$

- $\Phi_{start}$ ensures that the first row of the tableau is the starting configuration of $N$ on $w$ by the equality:
  $$\Phi_{start} = x_{1,1,\#} \land x_{1,2,q_0} \land x_{1,3,w_1} \land \cdots \land x_{1,n+3,w_n} \land \cdots \land x_{1,n+1,\#} \land x_{1,n,\#}.$$
- $\Phi_{accept}$ guarantees that an accepting configuration occurs in the tableau by placing $q_a$ in one of the cells by: $\Phi_{accept} = \bigvee_{1 \leq i,j \leq n} x_{i,j,q_a}$
- $\Phi_{move}$ guarantees that each row of the tableau correspond to a configuration that legally follows the preceding row's configuration according the $N$'s transition rules.

Comments:

Windows (a) and (b) are legals because the transition allows $N$ to move this way.

Window (c) could be either illegal or illegal because $q_1$ appears to the right side of the top row and we don’t know what symbol is the head over.

Window (d) is legal because top and bottom are identical, what could happen if the head weren’t adjacent to the location of the window.

Window (e) is legal because state $q_1$ reading a $b$ might have been immediately to the right of the top row and would have moved to the left

Window (f) is legal because state $q_1$ might have been immediately to the left of the top row changing $b$ to $c$ and moving left.

Legal window

A 2 × 3 window of cells is legal if that window does not violate the actions specified by $N$’s transition function. To explain, consider the transitions:

$$\delta(q_1,a) = \{q_1,b,R\}, \delta(q_1,b) = \{q_2,c,L\}, \delta(q_1,R) = \{q_2,a,R\}\}$$

Examples of legal windows for this machine are:

```
(a) (b) (c)
(a) # b a # a b #
(b) a c a a q_0 # a b
(c) # b a a b a b b b
(d) # b a a b a b b b
```
**Claim**

If the top row of the tableau is the start configuration and every every window is legal then each row is a configuration that legally follows the configuration represented by the preceding row.

**Proof:** show the claim for any two adjacent configurations.

---

**Illegal windows**

The figure below shows illegal windows of $N$.

\[(a)\]  
\[\begin{array}{ccc} 
a & b & a \\
 a & a & a \\
\end{array}\] 

\[(b)\]  
\[\begin{array}{ccc} 
q & 1 & a \\
 a & a & a \\
\end{array}\] 

\[(c)\]  
\[\begin{array}{ccc} 
q & 1 & b \\
 q & 1 & b \\
\end{array}\]

- \[(a)\] is illegal because central symbol on top can’t be changed because has no adjacent state
- \[(b)\] is illegal because transition states that $b$ get changed to $c$ not to $a$
- \[(c)\] is illegal because two states appear in the bottom row.

---

**Complexity of the reduction**

- Tableau is $n^k \times n^k$ and thus contains $n^{2k}$ cells; each cell has $|C| = l$ variables associated with it where $l$ depends only on $N$. Hence total number of variables is $O(n^{2k})$.
- Estimating the size of four components of $\Phi$: $\Phi_{\text{cell}}$ is a fixed fragment of $\Phi$ so its size is fixed and is $O(n^{4k})$; $\Phi_{\text{start}}$ has the size $O(n^k)$; $\Phi_{\text{move}}$ and $\Phi_{\text{accept}}$ have sizes $O(n^{2k})$. Hence total size of $\Phi$ is $\mathcal{O}(n^{2k})$, i.e., size of $\Phi$ is polynomial in $n$.
- Each component of $\Phi$ can be produced in polynomial time. Therefore we conclude that we can construct a reduction that produces $\Phi$ from $N$ in polynomial time.

This concludes the proof of Cook-Levin Theorem, showing that $SAT$ is NP-complete.

---

**Construction of $\Phi_{\text{move}}$**

$\Phi_{\text{move}}$ stipulates that all windows in the table are legal.

- Each windows contain six cells which may be set in a fixed number of ways to yield a legal window. $\Phi_{\text{move}}$ says that the setting of those six cells is done this way by:

  $\Phi_{\text{move}} = \bigwedge_{1 \leq i,j \leq 2k} (\text{window}[i,j] \text{ is legal})$

- Replace $\text{window}[i,j] \text{ is legal}$ with the following formula where $a_1, a_2, a_3, a_4, a_5, a_6$ are the contents of the six cells:

  $\bigvee a_1 \ldots a_6 \text{ is legal} (x_{i,j} - 1, a_1 \land x_{i,j}, a_2 \land x_{i,j+1}, a_3 \land x_{i+1,j-1}, a_4 \land x_{i+1,j}, a_5 \land x_{i+1,j+1}, a_6)$
Theorem 7.32

3SAT problem is polynomial time reducible to CLIQUE

- The polynomial time reduction \( f \) that we demonstrate from 3SAT to CLIQUE converts formulas to graphs.
- In the constructed graphs, cliques of a specialized size correspond to satisfying assignments of the formula.
- Structures within the graph are designed to mimic the behavior of the variables and clauses.

Proof

Let \( \Phi \) be a formula with \( k \) clauses:

\[ \Phi = (a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land \ldots \land (a_k \lor b_k \lor c_k) \]

The reduction \( f \) generates the string \( \langle G, k \rangle \) where \( G \) is an undirected graph.
Example 3SAT reduction to graphs

3SAT formula \( \Phi = (x_1 \lor x_1 \lor x_2) \land (\bar{x}_1 \lor \bar{x}_2 \lor \bar{x}_2) \land (x_1 \lor x_2 \lor x_2) \) is transformed in the graph.

Edges of \( G \)

Edges of \( G \) connect all but two types of pairs of nodes in \( G \):
- No edge is present between the nodes of the same triplet.
- No edge is present between two nodes with contradictory labels, such as \( x_2 \) and \( \bar{x}_2 \).

Proof, continuation

A \( k \)-clique in \( G \) implies \( \Phi \) is satisfiable. Suppose \( G \) has a \( k \)-clique:
- No two of the clique nodes can occur in the same triplet because nodes in the same triplet are not connected. Therefore each the \( k \)-triplets contains contains one of the \( k \)-clique nodes.
- Assign truth value to the variables of \( \Phi \) so that each literal labeling a clique node is made true. This is possible because two nodes with contradictory labels are not connected.
- This assignment satisfies the formula \( \Phi \). Since each node corresponds to a clause that has a true value in it the clause is true; since clauses are connected by \( \land \) the formula is true.

Why this construction works?

Claim: \( \Phi \) is satisfiable iff \( G \) has a \( k \)-clique.

Proof: \( 3SAT \Rightarrow CLIQUE \). Suppose that \( \Phi \) has a satisfying assignment.
- At least one literal is true in every clause (required by \( \lor \))
- In each triple of \( G \) we select one node corresponding to a true literal in the satisfying assignment. If more literals are true in some clause we select the true literal arbitrarily.
- The nodes just selected form a \( k \)-clique; number of nodes is \( k \) (there are \( k \) clauses in \( \Phi \)) and each pair of selected nodes are joined, by construction.
- Selected node are not from the same triplet, by construction; they could not have contradictory labels because otherwise the associated labels would be both true in the satisfying assignment. Hence \( G \) contains a \( k \)-clique.
**Other NP-complete languages**

To show that $A$ is NP-complete provide a polynomial time reduction from $A$ to $3SAT$ or to other NP-complete languages.

**Conclusions**

- If CLIQUE is solvable in polynomial time, so is $3SAT$.
- Polynomial time reducibility allows us to link these two very different problems.
- Similar link may be made among other problems.