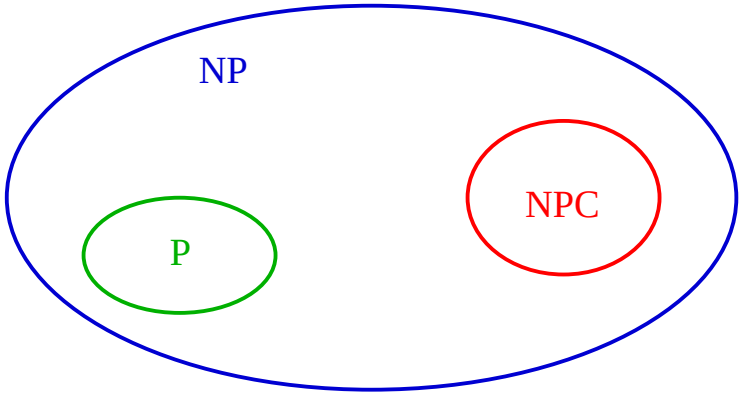
P, NP, and NP-complete

- **P**: Polynomial Time Solvable (using a Turing Machine)
- **NP**: Nondeterministic Polynomial Time Solvable (using a Nondeterministic Turing Machine)

  Since a nondeterministic turning machine is "more powerful" than a turing machine, therefore, $\mathbf{P} \subset \mathbf{NP}$.

- **NP-complete**: A subset of **NP** that "seems" to be hardest to solve. (We will give more accurate definition later.)
- The most-likely relationship of the three problem classes.



Why do we study complexity classes?

- We have studied techniques for solving algorithmic problems efficiently (Divide-and-Conquer, Dynamic Programming)

- *Question 1:* Can we solve all problems efficiently?

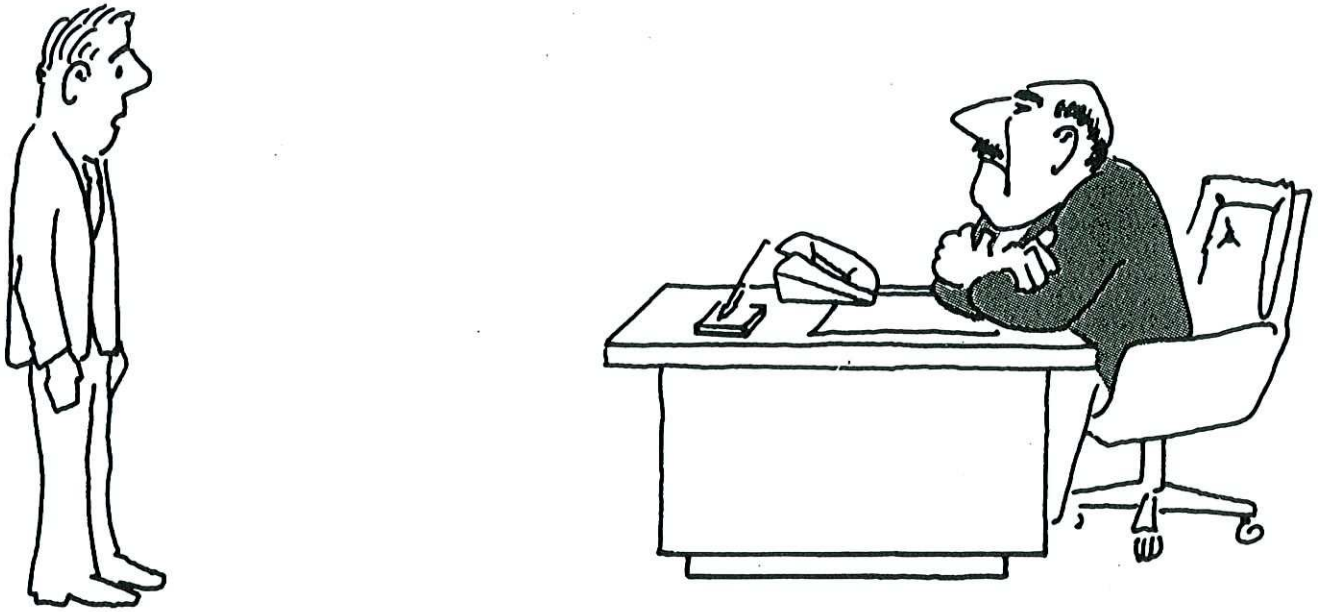
- *Answer:* No. There are problems that have no good general efficient solution! (one can prove that there is no good solution!)

- *Question 2:* How about the problem that seems to have no efficient solution and, at the same time, there is no proof that a good solution does not exist?

  † Maybe if we try even harder, we can find a good solution.

  † Maybe if we try even harder, we can prove that there is no good solution.

- NP-complete class helps us to "answer" question 2.



"I can't find an efficient algorithm, I guess I'm just too dumb."

"I can't find an efficient algorithm, because no such algorithm is possible!"

"I can't find an efficient algorithm, but neither can all these famous people."

Tractable

We say a problem is *tractable* if there is an algorithm which solves the problem in $O(p(n))$ time where

$p(n)$ is a polynomial in the size of the input n.

Input size: number of bits to represent the input (remember RAM model)

We can change $O(p(n))$ to $O(n^k)$. ($O(p(n)$ time $\Longleftrightarrow O(n^k)$ for some k.)

Input size is measured by bits, i.e., if the input string for a sorting algorithm consists of n different numbers then the input size is about $O(n\log n)$.

Class of **P**

† The class of all tractable problems is denoted by **P** (for polynomial time!).

† We consider **P** to be the class of problems which have efficient solutions (algorithms).

† Why do we want to use this definition? Is $O(n^{100})$ efficient?

(We have tried hard to improve an algorithm from $O(n^2)$ to $O(n\log n)$.)

- The definition is good for theoretical studies.
- Most tractable problems have practical solutions (low-degree polynomials).
- Running time more than polynomial is definitely not practical!

Class of **NP**

† For some problems, although finding a solution is difficult, verifying a correct solution is easy.

- E.g., factoring.

† If we want to study a computational problem, we want at least the correct solution of the problem can be verified efficiently (in polynomial time)

- Otherwise, if you tell me a solution, how do I believe you?
- You will need to spend tremendous time to convince me your solution is correct.
- Most computational problems we encounter in real life are in NP.

† The class of **NP** is the class of problems whose correct solutions can be verified in polynomial time.

† These problems are exactly the problems that can be solved in polynomial time by a "Non-deterministic Turing Machine", a machine that does not exist physically in this world.

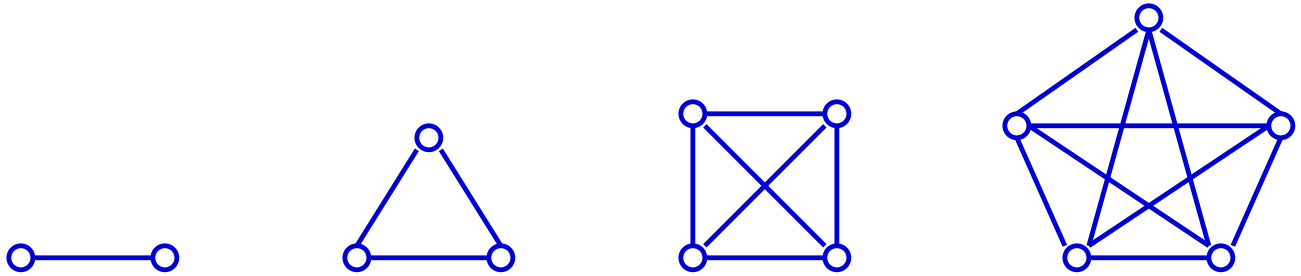
† $\mathbf{P} \subset \mathbf{NP}$.

† But we do not know if $\mathbf{P} = \mathbf{NP}$.

† What do you think?

An Example in NP

Clique Given an undirected graph $G = (V, E)$, a clique C in G is a subgraph of G such that all vertices in C are connected to all other vertices in C.



- Clique problem: Given $G = (V, E)$ and an integer k, determine whether G contains a clique of size $\geq k$.
- Solution: enumerate all possible groups of k vertices and then check if there is a clique.
- *Time:* at least $O(n^k)$. Note: k is not a constant.
  Let $k = n/2 \Longrightarrow$ at least $O(n^{n/2})$ not a polynomial time algorithm!!

Distinction of finding or verifying a solution

Here is a story which may help clarify this distinction of finding an answer and only verifying an answer:

In 1903, F.N. Cole factored $2^{67} - 1$. When E.T. Bell, the historian of mathematics, asked Cole how long it had taken him to find the factorization, he replied, "Three years of Sundays."

Bell goes on to say:

"At the October, 1903 meeting in New York of the American Mathematical Society, Cole had a paper on the program with the modest title "On the factorization of large numbers". When the chairman called on him for his paper, Cole – who was already a man of very few words – walked to the board, and, saying nothing, proceeded to chalk up the arithmetic of raising 2 to the sixty-seven power. Then he carefully subtracted 1. Without a word he move over to a clear space on the board and multiplied out, by longhand,

$$193,707,721 \times 761,838,257,287.$$

The two calculation agreed. ...

For the first and only time on record, an audience of American Mathematical Society vigorously applauded the author of a paper delivered before it. Cole took his seat without having uttered a word. Nobody asked him a question."

† Note that it took Cole 150 days to FIND the factorization, but once it was found, it was easy to VERIFY and convince the members of the audience that the number was composite.

Class of **NP-complete**

- In order to find the "gap" between **P** and **NP**, people have been studying the "hardest" problems in **NP**.
- Cook [1971] proved that an NP problem, called SAT, is "very hard to solve". Roughly,
  - If there is a polynomial time algorithm to solve SAT, then every problem in **NP** can be solved in polynomial time.
- This problem is therefore **NP-complete**, and SAT is the first **NP-complete** problem. (The precise definition of **NP-complete** is given later).
- People have found numerous **NP-complete** problems (hundreds or even thousands).

- However, nobody has found any polynomial time algorithms for any **NP-complete** problems.
- Therefore, people tend to *believe* that **NP-complete** problems do not have polynomial time algorithms, though no one can prove this so far.
- As a result, if you want to convince your boss that the computational task he assigns to you is too difficult, prove it is NP-complete!!!
  - "I cannot solve it in polynomial time. But neither did those Turing award winners."
- Next let us consider how to prove NP-completeness.

Decision Problems

- Decision version: The answer is either *yes* or *no*.
  Example: Is there a clique of size $\geq k$?
- Optimization version: The maximum size of the cliques of the graph

(Can be converted to a series of decision problem)

Is there a clique of size $\geq 0.5n$ N
$\geq 0.25n$ Y
$\geq 0.375n$ Y
...

log$N$ times of the decision problem.

A fast solution to Decision Problem usually can lead fast solution to the corresponding Optimization Problem.

We will only consider **Decision Problems**.

This restriction makes the discussion and the theory simple. We sometimes also say an optimization problem is NP-complete if its decision version is NP-complete.

Language-recognition problem

- A decision problem can be viewed as a language-recognition problem.
- An input of the problem can be encoded by a binary string.
- Let U be the set of all possible inputs to the decision problem. Let $L \subseteq U$ be the set of all inputs for which the answer to the problem is YES.

  We call L the language corresponding to the problem.
- A language-recognition problem is to recognize whether or not an input string belongs to L

- We can use the terms problem and language interchangeably

In talk about the NP-complete class, we need to talk about "reduction".

Reduction – an old joke: A mathematician and her husband are asked the following question:
"suppose that you are in the basement and you want to boil water, what do you do?"

The mathematician says that she will go up to the kitchen and boil water there; her husband answers similarly.

Now they are both asked the following question:
"suppose that you are now in the kichen and you want to boil water, what do you do now?"

The husband: "it's easier – I'll just fill the kettle and boil the water."
The mathematician: "it's even easier than that – I'll go down to the basement and I already know how to solve the problem from there."

Polynomial-Time reductions

This is the main tool in NP-completeness!

Definition. Let L_1 be a language from the input space U_1, and L_2 be a language from the input space U_2.

L_1 is *polynomially reducible* to L_2, written as $L_1 \leq_p L_2$, if there exists a polynomial-time algorithm that converts each input $u_1 \in U_1$ to another input $u_2 \in U_2$ such that $u_1 \in L_1$ if and only if $u_2 \in L_2$.

The algorithm is polynomial in the size of u_1, therefore $|u_2|$ is a polynomial of $|u_1|$.

Lemma. If L_1 is polynomially reducible to L_2 and there is a polynomial-time algorithm for L_2, then there is a polynomial-time algorithm for L_1.

Proof: Given $u_1 \in U_1$, in polynomial-time we convert it to $u_2 \in U_2$.

Now we can use polynomial-time algorithm for L_2 to determine if $u_2 \in L_2$.

If $u_2 \in L_2$, then we return YES.

If $u_2 \notin L_2$, then we return NO. $\square$

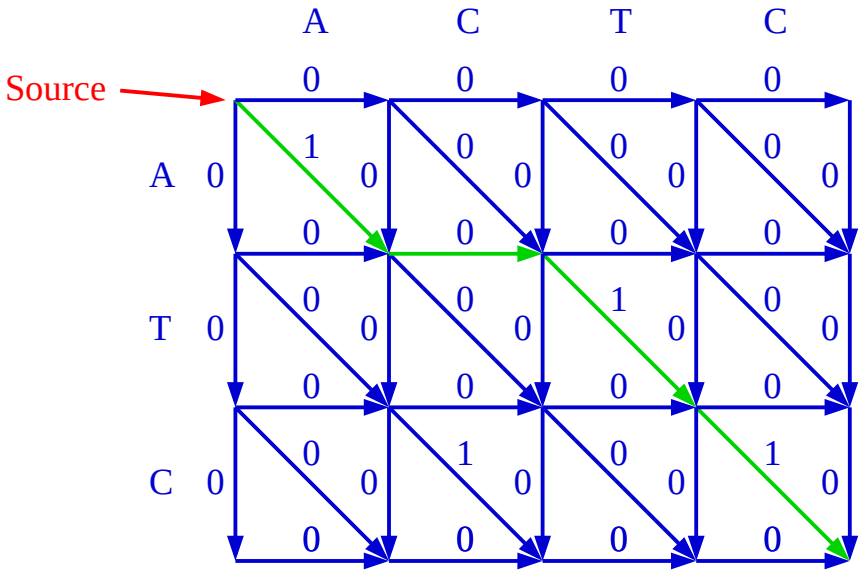
Lemma. If L_1 is polynomially ($O(n^k)$) reducible to L_2 and L_2 is polynomially ($O(n^l)$) reducible to L_3, then L_1 is polynomially reducible to L_3.

Proof: For $u_1 \in U_1$, reduce it to $u_2 \in U_2$, then reduce u_2 to $u_3 \in U_3$.
(1) "polynomial time": $O(n^k) + O(n^{k \times l}) = O(n^{k \times l})$.
(2) $u_1 \in L_1 \Leftrightarrow u_2 \in L_2 \Leftrightarrow u_3 \in L_3$
$\square$

An Example

Problem 1. Longest common subsequence
Problem 2. Longest paths (single source)
Claim: Problem 1 is polynomially reducible to Problem 2!

Given strings A and B, $|A| = n$, $|B| = m$, construct a directed graph of $O(nm)$ vertices and $O(nm)$ edges in $O(nm)$ time.



The graph is acyclic so the longest path can be found in $O(nm)$ time.
Therefore problem 1 can be solved in $O(nm)$ time!

- If L_1 is polynomially reducible to L_2 then we say L_2 is a *harder problem*

- The notion of reducibility is not symmetric. L_1 polynomially reducible to L_2 does not imply that L_2 is also polynomially reducible to L_1.

- L_1 and L_2 are *polynomially equivalent* if each is polynomially reducible to the other

- When we cannot solve a problem easily (efficiently), we try to find whether it is equivalent to another problem which is known to be hard.

NP-complete

NP-Hard

A problem X is called an NP-hard problem if every problem in NP is polynomially reducible to X.

NP-Complete

A problem X is called an NP-complete problem if
(1) X belongs to NP, and
(2) X is NP-hard

By the definition, if any NP-hard problem is ever proved to belong to P, then P= NP!

How to prove NP-completeness?

- Cook [1971] proved that SAT is NP-complete from the definition. He won Turing Award in 1982 because of this.

- Once we have one NP-complete problem, proving that other problems are NP-complete becomes easier!

Lemma. A problem X is an NP-complete problem if (1) X belongs to NP and (2) Y is polynomially reducible to X, for some problem Y that is NP-complete.

Proof. All NP problems are polynomially reducible to Y. Therefore they are polynomially reducible to X.

History

- Cook [1971] showed first NP-complete problem (SAT problem).
- Karp [1972] found 24 important problems to be NP-complete
- Many other problems have been proved to be NP-complete (hundreds, maybe even thousands).
- Examples:
  - Clique
  - Hamiltonian Cycle
  - TSP (travelling salesman problem)
  - Knapsack problem
- *Exercise: Why is that Knapsack is NP-complete and has a good dynamic programming algorithm?*

Cook's theorem - first NPC problem

The SAT problem is NP-complete

† Boolean expression:

Boolean variable: values 0 or 1

Operator: *and, or, complement.*

† Conjunctive normal form (CNF)= Boolean expression which is product (*and*) of several sums (*or*).

Example: $S = (x + y + \bar{z}) \cdot (\bar{x} + y + z) \cdot (\bar{x} + \bar{y} + \bar{z})$

x, y, z: Boolean variables

$+ =$ *or*, $\cdot =$ *and*, $\bar{x} =$ *complement of* x.

† Satisfiable.

A boolean expression is satisfiable if there is an assignment of 0's and 1's to its variables such that the expression is 1.

In the above example, let $x = 1, y = 1, z = 0$. With this assignment of values to the variables, S has value 1.

SAT problem

Given a boolean expression in CNF, determine whether the expression is satisfiable.

– SAT is in NP

We can "guess" a correct assignment and then test it.

– SAT is NP-hard

Well, let us omit this part. The basic idea is to describe the behavior of a Turing Machine using a boolean expression. Then prove that the Turing Machine outputs Yes if and only the boolean expression is satisfiable.

– Cook's theorem: The SAT problem is NP-complete. □

An easy example of NPC proof

- Suppose we have already known the NP-completeness of Hamiltonian Cycle problem. How do we prove that TSP is NP-hard.

  Hamiltonian Cycle Problem:
  Given an undirected graph G, does G have a simple cycle that contains each vertex?

  Traveling-Saleman Problem (TSP):
  Given a weighted, undiredted, complete graph G and k, does G have a simple cycle that contains each vertex (Traveling-Salesman tour) with a cost at most k?

- Let $G =< V, E >$ be a Hamiltonian Cycle problem. We construct a weighted complete graph $G' = (V, E')$, as follows:

$$E' = \{\{i, j\} : i, j \in V \text{ and } i \neq j\}$$

$$c(i, j) = \begin{cases} 0 & \text{if } \{i, j\} \in E \\ 1 & \text{if } \{i, j\} \notin E \end{cases}$$

Now G' and zero is the instance of TSP.

G contains a Hamiltonian Cycle if and only if G' has Traveling-Salesman tour with cost at most zero.

Why this is an easy proof?

Hamilton Cycle seems to be an easier version of Traveling-Salesman tour problem.

So we just reduced an easy problem to a harder problem.

A common mistake in proving NP-hard is that, instead of reducing the NP-hard problem to your problem, a proof of reducing your problem to the NP-hard problem is given.

More examples of NP-completeness proof

Clique:
1) Clique is NP
2) reduce SAT to clique problem.

- Let E be a CNF boolean expression

  $E = E_1 \cdot E_2 \cdot \ldots E_m$

  $E_i = x_{i_1} + x_{i_2} + \ldots x_{i_k}$

  i.e. $E_i = (x + y + z + \bar{w})$

- We associate a "column" of i_k vertices with the variables in E_i.

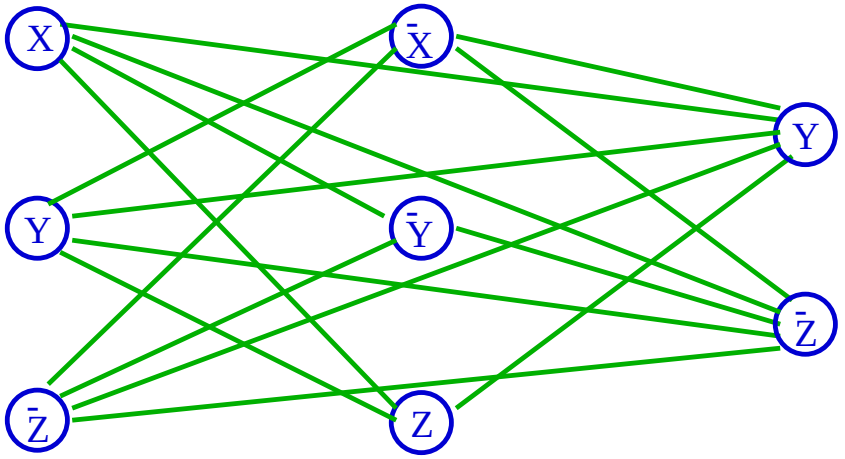
  Same variables in different clauses will correspond to different vertices in "different" columns.

- edges: vertices in the same "column" will NOT be connected to each other.

  Vertices in different "columns" are almost always connected unless they are in complementary form (i.e. x and $\bar{x}$)

An Example

$E = (x + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot (y + \bar{z})$



Given $E = E_1 \cdot E_2 \ldots E_m$ We can construct G in polynomial time and we choose $k = m$. We claim that G has a clique of size $\geq m$ if and only if E is satisfiable.

("$\Longleftarrow$") If E is satisfiable then
there is an assignment such that each clause contains at least one variable with value 1.
We choose one vertex per "column". The vertex corresponding to the variable with value 1.

$\Longrightarrow$ These m vertices form a clique since any two of them must have an edge in common.
(If they do not have an edge in common, then they must be x and $\bar{x}$. But x and $\bar{x}$ cannot be true at the same time.)

"$\Longrightarrow$" If G has a clique $\geq m$

– We assign each variable corresponding to each vertex in the clique value 1

– there will be no conflict in the assignment since in the clique every vertex is connected to every other vertex.

– if there are variables left, they can be assigned arbitrarily

In the previous example,

$\{x, \bar{y}, \bar{z}\}$ is a clique, we assign $x = 1$, $\bar{y} = 1$, $\bar{z} = 1$

$\{y, \bar{x}, y\}$ is a clique, we assign $y = 1$, $\bar{x} = 1$, $z = 1$ or $z = 0$.

3SAT is NP-complete

The Problem: Given a boolean expression in CNF such that each clause contains exactly three variables, determine whether it is satisfiable.

1) 3SAT is clearly in NP since SAT is in NP
2) reduce SAT to 3SAT

• Given $E = E_1 \cdot E_2 \ldots E_m$

Let $E_i = (x_1 + x_2 + \ldots x_k)$ $k \geq 4$

Translate E_i into F_i (use new variables $y_1, y_2, \cdots, y_{k-3}$)

$$F_i = (x_1 + x_2 + y_1) \cdot (x_3 + \bar{y_1} + y_2) \cdot (x_4 + \bar{y_2} + y_3) \cdots (x_{k-1} + x_k + \overline{y_{k-3}})$$

Example

$$E_i = (x_1 + x_2 + x_3 + x_4 + x_5 + x_6)$$
$$F_i = (x_1 + x_2 + y_1)(x_3 + \bar{y_1} + y_2)(x_4 + \bar{y_2} + y_3)(x_5 + x_6 + \bar{y_3})$$

$$E_i = (x_1 + x_2 + x_3 + x_4 + x_5)$$
$$F_i = (x_1 + x_2 + y_1)(x_3 + \bar{y_1} + y_2)(x_4 + x_5 + \bar{y_2})$$

$$E_i = (x_1 + x_2 + x_3 + x_4)$$
$$F_i = (x_1 + x_2 + y_1)(x_3 + x_4 + \bar{y_1})$$

Claim: E_i is satisfiable if and only if F_i is satisfiable.

E_i satisfiable $\Longrightarrow$

This means that $\exists x_i = 1$.
Assign $y_j = 1$ for $j \leq i-2$, and $y_j = 0$ for $i-1 \leq j \leq k-3$.
This implies F_i is satisfiable.

F_i satisfiable $\Longrightarrow$

If E_i is not satisfiable $\Longrightarrow x_i = 0$ for all i.
This implies $F_i = (y_1)(\bar{y_1} + y_2)(\bar{y_2} + y_3) \ldots (\overline{y_{k-3}})$
which is not satisfiable, a contradiction!

Special Cases

For $E_i = (x_1 + x_2)$

$$F_i = (x_1 + x_2 + z) \cdot (x_1 + x_2 + \bar{z})$$

For $E_i = x$,

$$F_i = (x + y + z) \cdot (x + \bar{y} + z) \cdot (x + y + \bar{z}) \cdot (x + \bar{y} + \bar{z})$$

It is clear that E_i satisfiable iff F_i satisfiable.

Therefore, an instance of SAT can be polynomially reduced to an instance of 3-SAT.

Theorem 1. *3-SAT is NP-complete.*

Note: 2SAT is in P!!!

3-coloring

- Let $G = (V, E)$ be an undirected graph.

  A valid coloring of G is an assignment of colors to vertices such that each vertex is assigned one color and no two adjacent vertices have the same color.

- 3-Coloring problem

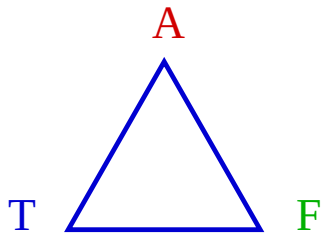
  Given an undirected graph $G = (V, E)$, determine whether G can be coloured with three colors.

- 3-Coloring is in NP

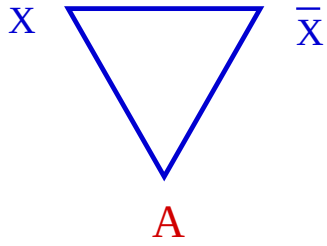
  Guess a 3-coloring and then check that it is a valid coloring in polynomial time.

Reduce 3SAT problem to 3-coloring

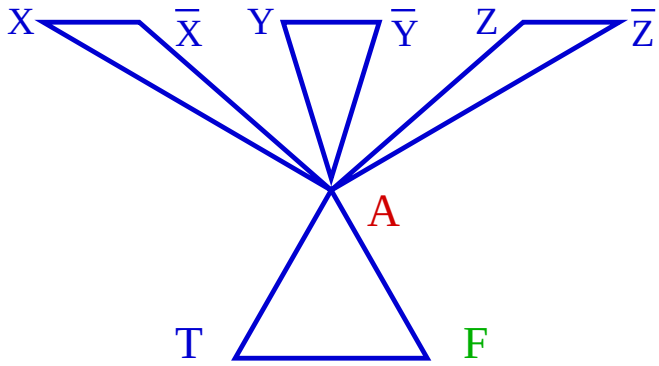
Main triangle M: use color T, F, A (T for true, F for false).



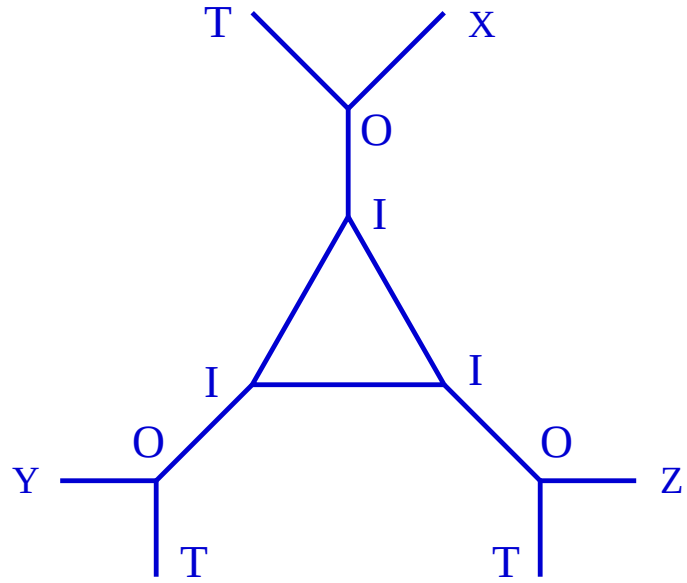
For each variable x, build a triangle M_x such that x and $\bar{x}$ have different colors.



If we have x, y, z, then



For each clause $(x + y + z)$
build a graph to impose the condition that at least one variable has value true.

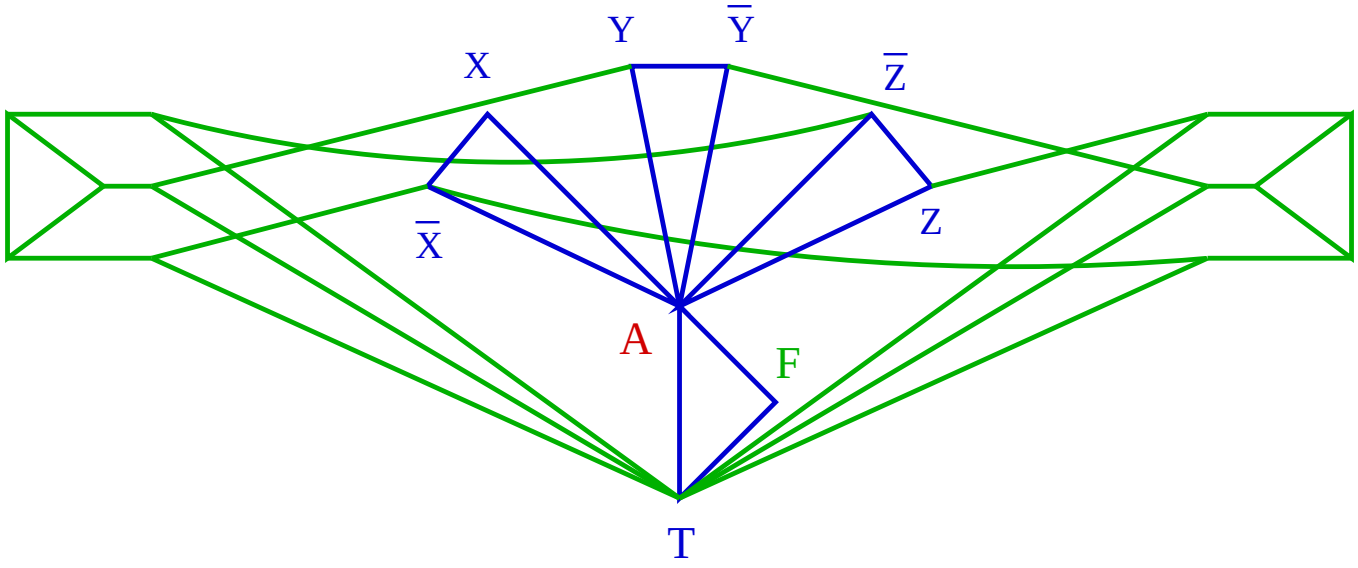


If all x, y, z are false, then we cannot 3-color this graph.

(x, y, z use F $\Longrightarrow$ O's use A $\Longrightarrow$ we cannot color all I's)

An Example

$$(\bar{x} + y + \bar{z}) \cdot (\bar{x} + \bar{y} + z)$$



If $E = E_1 \cdot E_2 \cdot \ldots E_m$ is satisfiable, color M as above. Color the outer vertex corresponding to variable with value 1 to F. Color all other outer vertices to A. Now we can color inner triangle with A, T, and F.

Therefore G is 3-colorable.

If G is colored with 3 colors,

from M we can determine colors T, F, A, and the construction of G guarantees that at least one vertex in each block will be colored by T. Therefore E is satisfiable since we can use T and F to assign values to all variables in E.

Consequently, E is satisfiable iff G is 3-colorable.

Graph 2-colroing is in P!!!

Dealing with NP-complete problems

- Proving a given problem is NP-complete does not make the problem go away!
- In many cases we still need to solve it!
- Since we (most probably) cannot solve the problem in polynomial time, we have to compromise
- We can compromise the optimality, the guaranteed efficiency and the completeness of the solution

- Approximately the right answer (compromise the optimality)
  - polynomial time
  - within 50 % and 30 % of the optimality is guaranteed
- Usually fast (compromise the efficiency)
  - always finds a correct solution
  - average time $<$ exponential
  - sometimes (rarely) exponential
- Almost surely (compromise the completeness)
  - polynomial time always
  - solution is correct if found
  - does not always find a solution

Approximation algorithms with guaranteed performance

- Guaranteed performance means the result is not too far from the optimal solution
- *Euclidean Traveling Salesman Problem*

Traveling Salesman Problem is NP-complete!

Let $C = \{c_1, c_2, \ldots c_n\}$ be a set of *cities*. A distance $D(c_i, c_j) > 0$ for all pairs (c_i, c_j) is defined.

Problem: Find a minimum length simple cycle that contains each vertex in G exactly once.

Euclidean Traveling Salesman Problem is NP-complete!
C is a set of points in the plane and d is the real distance between them. (distances satisfy triangle inequality)

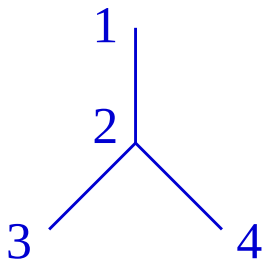
An Approximation Algorithm for ETSP

- ETSP tour: a simple cycle that contains each vertex exactly once.
- Compute the minimum spanning tree (edge weight = distance)
- The cost of MST is no more than the length of the best ETSP tour.
  - ETSP tour is a cycle
  - delete an edge from ETSP tour results in a tree
  - cost of this tree is greater than or equal to the length of MST
  - therefore the length of MST $\leq$ the length of ETSP tour.

† The MST is a tree not a tour (cycle). We have to modify it.

† Consider the depth-first traversal of MST.

† We can get a circuit from depth-first search traversal and edges in opposite direction whenever the search backtracks.



Circuit: (1, 2), (2, 3), (3, 2), (2, 4), (4, 2), (2, 1)

Tour: (1, 2), (2, 3), (3, 4), (4, 1)

† The cost of this circuit is no more than twice the length of best ETSP tour

† We convert this circuit into an ETSP tour by taking direct routes instead of always backtracking.

(The assumption of Euclidean is important here since it guarantees the direct route is always better.)

† The resulting tour has a length no more than twice the length of the best ETSP tour.

(This means guaranteed performance!)

† Complexity of the algorithm:

Time is determined by MST algorithm, which is bound by $O(n^2 \log n)$. (For Euclidean graph by $O(n \log n)$)

Example: