Chapter 8: The Theory of NP-Completeness

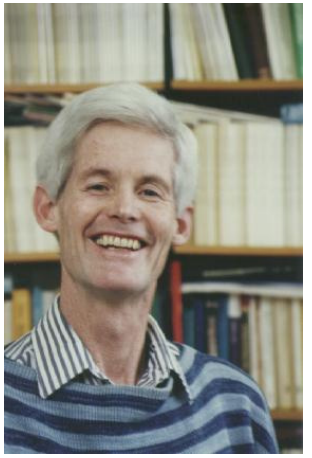
Chin Lung Lu

Department of Computer Science

National Tsing Hua University

Stephen Cook

- A professor at Toronto University and the author of the famous paper [1] for the theory of NP completeness
- The 1982 recipient of the Turing award



[1] The complexity of theorem proving procedures. STOC, pp.151–158, 1971.

Informal discussion of NP-completeness

- The theory of NP-completeness is important since it helps us identify a large class of difficult problems.
- The difficult problems are those problems whose lower bound seems to be in the order of an exponential function.
- In other words, the theory of NP-completeness has identified a large class of problems which seem to have no polynomial time algorithms to solve them.

Informal discussion of NP-completeness (cont'd)

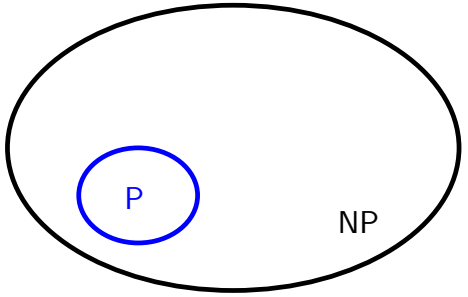
- Roughly speaking, we may say that theory of NP-completeness first points out that many problems are NP (non-deterministic polynomial-time) problems.
- However, not all NP problems are difficult (many are easy).

  Example 1 Searching problem $\in$ NP problems

  Example 2 Minimum spanning tree problem $\in$ NP problems
- In fact, the above two problems also belong to P (polynomial-time) problems.

Informal discussion of NP-completeness (cont'd)

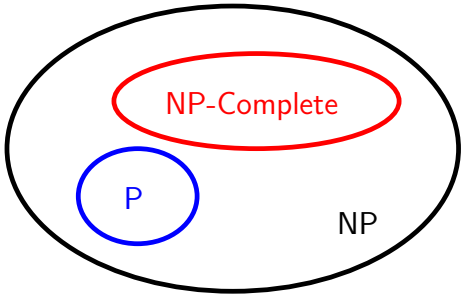
- The set of NP problems contains the set of P problems.



Commonly believed relationship between P and NP

Informal discussion of NP-Completeness (cont'd)

- Moreover, it has been shown that there is also a large class of problems, which are NP-complete problems, contained in the set of NP problems.



Commonly believed relationship among P, NP and NP-complete

Informal discussion of NP-completeness (cont'd)

- The set of known NP-complete problems is very large and also keeps growing all the time.
  - Example 1 Satisfiability problem
  - Example 2 Traveling salesperson problem
  - Example 3 Bin packing problem
- Up to now, none of the NP-complete problems can be solved by any polynomial-time algorithm in worst cases.
- That is, the current best algorithm to solve any NP-complete problem has exponential-time complexity in the worst case.

Informal discussion of NP-completeness (cont'd)

- According to the NP-completeness theory, if any NP-complete problem can be solved in polynomial time, all NP problems can be solved in polynomial time.
- Or, equivalently, if any NP-complete problem can be solved in polynomial time, then NP=P.

Informal discussion of NP-completeness (cont'd)

- Because it is very unlikely that all NP problems can be solved by polynomial-time algorithms, it is unlikely that any NP-complete problem can be solved by any polynomial-time algorithm.
- However, the theory of NP-completeness has not claimed that all NP-complete problems can never be solved by polynomial-time algorithms.
- It merely says that it is quite unlikely that any NP-complete problem can be solved in polynomial number of steps.

Optimization problems

Definition:
An optimization problem is a problem whose solution needs to be optimized.

Example: Traveling salesperson problem

Given a set of cities, find a shortest round-trip tour that visits each city exactly once and returns to the starting city.

Decision problems

Definition:
The decision problem is a problem whose solution is "yes" or "no".

Example: Traveling salesperson decision problem

Given a set of cities and a constant $c > 0$, is there a round-trip tour visiting each city exactly once and returning to the starting city whose total length is less than or equal to c?

Decision vs optimization problems

- The optimization problems are more difficult to solve than their corresponding decision problems.

Example:

- If we can solve the traveling salesperson problem, then we can solve its corresponding decision problem, but not vice versa.
- Hence, we can conclude that the traveling salesperson problem is more difficult than the traveling salesperson decision problem.
- On the other hand, if the traveling salesperson decision problem can not be solved by polynomial-time algorithms, we can conclude that the traveling salesperson problem cannot be solved by polynomial-time algorithms.

Decision vs optimization problems (cont'd)

- Note that in discussing NP problems, we shall only discuss decision problems.

Boolean formula

- Let x_i denote a Boolean variable whose value is either true (denoted by T) or false (denoted by F).
- Let $-x_i$ denote the negation of x_i.
- A literal is either a Boolean variable x_i or its negation $-x_i$.
- The symbol $\vee$ denotes "or" operation and $\wedge$ (or &) denotes "and" operation.
- A clause is a disjunction of literals (i.e., $L_1 \vee L_2 \vee \cdots$), where no clause contains a literal and its negation simultaneously.
- A formula is in its conjunctive normal form (CNF) if and only if it is in the form of c_1 & c_2 & $\ldots$ & c_m (or $c_1 \wedge c_2 \wedge \ldots \wedge c_m$), where each c_i is a clause for $1 \leq i \leq m$.
- Since every Boolean formula can be transformed into a CNF formula, we assume that all formulas are already in CNF.

Satisfiability (SAT) problem

Definition:
Given a Boolean formula, the satisfiability problem is to determine whether this formula is satisfiable or not.

- Example Consider the following formula:

$$\begin{array}{lll} & x_1 \vee x_2 \vee x_3 \\ \& & -x_1 \\ \& & -x_2 \end{array}$$

The following assignment makes the formula true:

$$\begin{array}{l} x_1 \leftarrow F \\ x_2 \leftarrow F \\ x_3 \leftarrow T \end{array}$$

- We use $(-x_1, -x_2, x_3)$ to denote $\{x_1 \leftarrow F, x_2 \leftarrow F, x_3 \leftarrow T\}$.

Satisfiability (SAT) problem (cont'd)

- If an assignment makes a formula true, then this assignment is said to satisfy the formula; otherwise, the assignment is said to falsify the formula.
- If there is at least one assignment that satisfies a formula, then we say that this formula is satisfiable; otherwise, this formula is unsatisfiable.

Example 1:

$$\begin{array}{lll} & x_1 \\ \& & -x_1 \end{array}$$

Example 2:

$$\begin{array}{lll} & x_1 \vee & x_2 \\ \& & x_1 \vee & -x_2 \\ \& & -x_1 \vee & x_2 \\ \& & -x_1 \vee & -x_2 \end{array}$$

Logical consequence

Definition:
A formula G is a logical consequence of a formula F if and only if whenever F is true, G is true (that is, every assignment satisfying F also satisfies G).

- Example Consider the following Boolean formula:

$$\begin{array}{lll} & -x_1 \vee x_2 & (1) \\ \& & x_1 & (2) \\ \& & x_3 & (3) \end{array}$$

Then x_2 is a logical consequence of the above formula.

Resolution principle

Definition of resolution principle:
Given two clauses c_1 and c_2 as follows:

$$\begin{array}{lllllll} c_1: & L_1 & \vee & L_2 & \vee & \ldots & \vee & L_j \\ c_2: & -L_1 & \vee & L_2' & \vee & \ldots & \vee & L_k' \end{array}$$

we can deduce a clause

$$(L_2 \vee \ldots \vee L_j) \vee (L_2' \vee \ldots \vee L_k')$$

as a logical consequence of c_1 & c_2, if it contains no pair of literals which are complementary to each other.

Resolution principle (cont'd)

Example 1:

- Consider the following clauses:

$$\begin{array}{llll} c_1: & -x_1 & \vee & x_2 \\ c_2: & x_1 & \vee & x_3 \end{array}$$

- Then $c_3: x_2 \vee x_3$ is a logical consequence of c_1 & c_2.

Definition of resolvent:
The clause c_3 generated by applying the resolution principle to c_1 and c_2 is called a resolvent of c_1 and c_2.

Resolution principle (cont'd)

Example 2:

- Consider another example with the following two clauses:

$$\begin{array}{ll} c_1: & x_1 \\ c_2: & -x_1 \end{array}$$

- Then the resolvent is an empty clause that contains no literal.
- We use $\varnothing$ to denote a empty clause.

Resolution principle (cont'd)

- If an empty clause can be deduced from a set of clauses by using the resolution principle, this set of clauses must be unsatisfiable.
- Example Consider the following set of clauses:

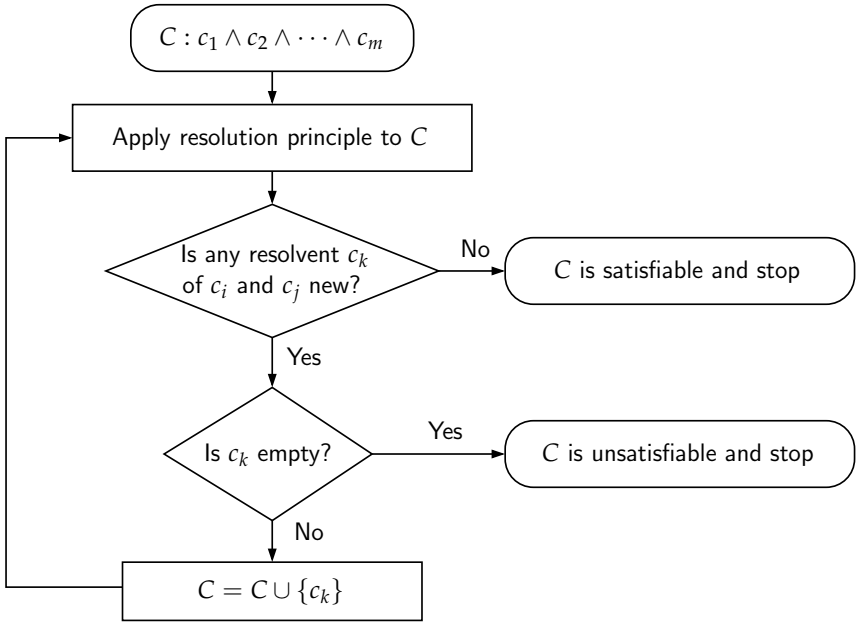
$$\begin{aligned} x_1 \vee x_2 &\quad (1)\\ x_1 \vee -x_2 &\quad (2)\\ -x_1 \vee x_2 &\quad (3)\\ -x_1 \vee -x_2 &\quad (4) \end{aligned}$$

We can deduce an empty clause as follows:

$$\begin{aligned} (1) \text{ and } (2) &\Rightarrow x_1 \quad (5)\\ (3) \text{ and } (4) &\Rightarrow -x_1 \quad (6)\\ (5) \text{ and } (6) &\Rightarrow \varnothing \quad (7) \end{aligned}$$

Deduction method to solve SAT problem

1. Given a set of clauses, we may repeatedly apply the resolution principle to deduce new clauses.
2. These new clauses are added to the original set of clauses and the resolution principle is applied to them again.
3. This process is terminated if an empty clause is generated or no new clauses can further be deduced.
4. If an empty clause is deduced, then the original set of clauses is unsatisfiable.
5. If no new clauses can be further deduced when the process is terminated, then this set of clauses is satisfiable.



Deduction method to solve SAT problem (cont'd)

- Example Let us consider a satisfiable set of three clauses:

		$-x_1$	$\vee$	$-x_2$	$\vee$ x_3	(1)
		x_1				(2)
		x_2				(3)
(1) and (2)	$\Rightarrow$	$-x_2$	$\vee$	x_3		(4)
(3) and (4)	$\Rightarrow$	x_3				(5)
(1) and (3)	$\Rightarrow$	$-x_1$	$\vee$	x_3		(6)

- Now no new clause can be deduced from clauses (1) to (6) any further.
- Therefore, we can conclude that this set of clauses is satisfiable.

Deduction method to solve SAT problem (cont'd)

- If we modify the above set of clauses by adding $-x_3$ into it, we have an unsatisfiable set of clauses:

		$-x_1$	$\vee$ $-x_2$	$\vee$ x_3	(1)
		x_1			(2)
		x_2			(3)
		$-x_3$			(4)
(1) and (2)	$\Rightarrow$	$-x_2$	$\vee$ x_3		(5)
(4) and (5)	$\Rightarrow$	$-x_2$			(6)
(3) and (6)	$\Rightarrow$	$\varnothing$			(7)

- This set of clauses is unsatisfiable because an empty clause is deduced.

Deduction method to solve SAT problem (cont'd)

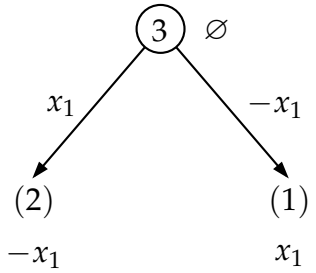
- The deduction method seems to have nothing to do with the finding of assignments for satisfying the formula.
- Below, we shall show that the deduction method is equivalent to the assignment finding approach.
- That is, the deduction of an empty clause is equivalent to the failure of finding any assignment satisfying all clauses.
- Conversely, the failure to deduce an empty clause is equivalent to finding at least one assignment satisfying all clauses.

Semantic tree

- Let us start with a simplest example as follows:

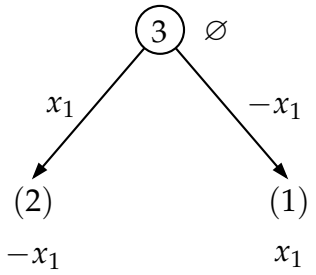
x_1	(1)
$-x_1$	(2)

- Then we construct a semantic tree as shown below:



Semantic tree (cont'd)

- The left-hand branch means that the assignment will contains x_1 (meaning $x_1 \leftarrow T$).
- The right-hand branch means that the assignment will contains $-x_1$ (meaning $x_1 \leftarrow F$).
- It can be seen that the left-hand assignment falsifies clause (2).
- Hence, we terminate the branch with (2).
- Similarly, we can terminate the right-hand branch with (1).



Semantic tree (cont'd)

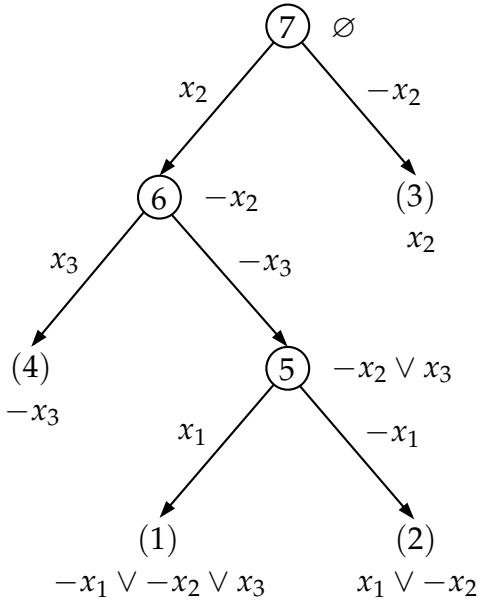
- Consider another example with the following set of clauses:

$$
\begin{array}{lll}
-x_1 \vee -x_2 \vee x_3 & & (1) \\
x_1 \vee -x_2 & & (2) \\
x_2 & & (3) \\
-x_3 & & (4)
\end{array}
$$

- An empty clause is deduced as follows.

$$
\begin{array}{lll}
(1) \text{ and } (2) \Rightarrow -x_2 \vee x_3 & & (5) \\
(4) \text{ and } (5) \Rightarrow -x_2 & & (6) \\
(3) \text{ and } (6) \Rightarrow \varnothing & & (7)
\end{array}
$$

Semantic tree (cont'd)



Rules for constructing semantic tree

Given a Boolean formula $C = c_1 \ \& \ \cdots \ \& \ c_m$, we can constructed a semantic tree ST of C by the following rules.

1. For each internal node of ST, there are two branches from this internal node. One of them is labeled with x_i and the other is labeled with $-x_i$, where x_i is a variable occurring in C.
2. A node is terminated when the assignment, which corresponds to the literals occurring in the path from the root to this node, falsifies c_j. Mark this node as a terminal node and attach c_j to this terminal node.
3. No path of ST may contain complementary pair so that each assignment is consistent.

Properties of semantic tree

- Each semantic tree is finite.
- If each terminal node of the semantic tree is attached with a clause, no assignment satisfying all clauses exists.
- Otherwise, there exists at least one assignment that satisfies all clauses and hence this set of clauses is satisfiable.

Properties of semantic tree (cont'd)

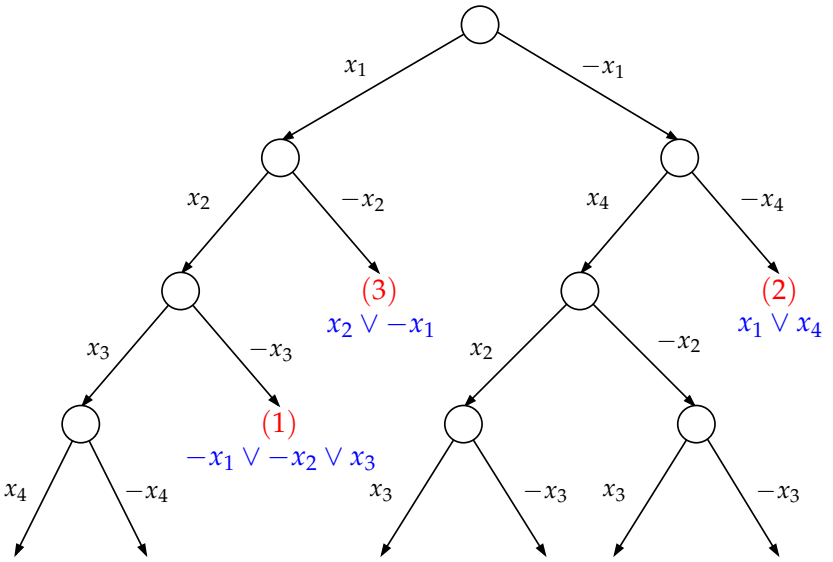
- Example Consider the following set of clauses:

$$\begin{aligned} -x_1 \ \vee \ -x_2 \ \vee \ x_3 \quad & (1) \\ x_1 \ \vee \ x_4 \quad & (2) \\ x_2 \ \vee \ -x_1 \quad & (3) \end{aligned}$$

- According to the semantic tree on the next slide, the following six assignments all satisfy the above formula.

1. (x_1, x_2, x_3, x_4)
2. $(-x_1, -x_2, x_3, x_4)$
3. $(-x_1, x_2, x_3, x_4)$
4. $(x_1, x_2, x_3, -x_4)$
5. $(-x_1, -x_2, -x_3, x_4)$
6. $(-x_1, x_2, -x_3, x_4)$

Properties of semantic tree (cont'd)



Time complexity of deduction algorithm

- According to the above discussion, the deduction method are equivalent to the assignment finding algorithm for solving the SAT problem.
- It is not hard to see that the worst-case time complexity of the deduction approach is exponential.
- Up to now, the best available algorithms for the SAT problem cost exponential time in worst cases.

SAT problem (cont'd)

Can be solved in polynomial time?

Question:

Is there any possibility that the SAT problem can be solved in polynomial time?

- By the theory of NP-completeness, if the SAT problem can be solved in polynomial time, then all NP problems can be solved in polynomial time.

Non-deterministic algorithm

- The notation NP denotes non-deterministic polynomial-time.
- A non-deterministic algorithm is an algorithm consisting of two phases: guessing and checking.
- Furthermore, it is assumed that a non-deterministic algorithm always make a correct guessing.

Non-deterministic algorithm (cont'd)

Example 1: SAT problem

Given a Boolean formula, is there an assignment satisfying the formula?

A non-deterministic algorithm for the SAT problem:

1. Guess an assignment.
2. Check whether this assignment satisfies the formula or not.

- A correct solution is always obtained by guessing.
- That is, if the formula is satisfiable, then a non-deterministic algorithm always guesses correctly and obtains an assignment satisfying this formula.

Non-deterministic algorithm (cont'd)

Example 2: Traveling salesperson decision problem

Given a set of cities and a constant $c > 0$, is there a round-trip tour visiting each city exactly once and returning to the starting city whose total length is less than or equal to c?

A non-deterministic algorithm for this problem:

1. Guess a tour.
2. Check whether this tour visits every city once and its length is shorter than the constant c.

Non-deterministic algorithm (cont'd)

- How can we always make a correct guess?
- In fact, non-deterministic algorithms do not exist and they will never exits.
- However, the concept of non-deterministic algorithms is useful because it will help us later define a class of problems, called NP problems.

NP algorithms and problems

Definition of NP algorithm:

If the checking of a non-deterministic algorithm is of polynomial time complexity, then this non-deterministic algorithm is called a non-deterministic polynomial-time algorithm (NP algorithm).

Definition of NP problem:

If a decision problem can be solved by an NP algorithm, then this decision problem is called a non-deterministic polynomial-time problem (NP problem).

Definition of P problem:

If a decision problem can be solved by a deterministic algorithm in polynomial time, then this decision problem is called a polynomial-time problem (P problem).

NP algorithms and problems (cont'd)

- By definition, every problem which can be solved in polynomial time by deterministic algorithms must be a non-deterministic polynomial-time problem.
- That is, $P \subseteq NP$.
- We focus on decision problems when talking about P and NP.

NP algorithms and problems (cont'd)

- The SAT and traveling salesperson decision problems are NP problems, because the checking stage for both problems is of polynomial time complexity.
- In fact, most solvable problems that one can think of are NP problems.

Undecidable problems

- A famous decision problem which is not an NP problem is the halting problem.

Definition of halting problem:

Given an arbitrary program with an arbitrary input data, will the program terminate or not?

- The halting problem is called an undecidable problem, because it cannot be solved by an NP algorithm.
- It was proved by Alan Turing in 1936.
- Undecidable problems are more difficult than NP problems.

Halting problem

Halting problem is undecidable:

Alan Turing proved in 1936 that there is no algorithm which can solve the halting problem for all possible inputs.

- Let $Halt(A,I) = \begin{cases} 1, & \text{if } A(I) \text{ halts} \\ 0, & \text{otherwise (i.e., } A(I) \text{ does not halt)} \end{cases}$

Algorithm $T(A)$:

if $Halt(A,A) = 1$ **then** enter infinite loop **else** halt.

- Suppose that we run the algorithm T with input T.
- If $T(T)$ halts, then $Halt(T,T) = 1$ and consequently Algorithm T enters infinite loop on input T (i.e., $T(T)$ does not halt).
- If $T(T)$ does not halt, then $Halt(T,T) = 0$ and consequently Algorithm T halts on input T (i.e., $T(T)$ halts).

Cook's theorem

Cook's theorem:

NP = P if and only if the satisfiability problem is a P problem.

- "If NP = P, then the SAT problem is a P problem" is obvious, because the SAT problem is an NP problem.
- However, "If the SAT problem is a P problem, then NP = P" is not obvious at all.

Main spirit of Cook's theorem

Problem reduction:

- Suppose that we have an NP problem A which is quite difficult to solve.
- Instead of solving A directly, we create another NP problem A' and by solving A', we obtain the solution of A.
- Note that every problem we consider here is a decision problem.

Main spirit of Cook's theorem (cont'd)

- Since problem A is an NP problem, there must exist an NP algorithm B that solves this problem A.
- Because B is a non-deterministic polynomial-time algorithm, it is physically impossible and therefore we cannot use it.
- We shall construct a Boolean formula C corresponding to B such that C is satisfiable if and only if the non-deterministic algorithm B terminates successfully (i.e., returning an answer "yes").
- Note that if C is unsatisfiable, the non-deterministic algorithm B would terminate unsuccessfully (i.e., returning an answer "no").

Main spirit of Cook's theorem (cont'd)

- After constructing formula C, we shall temporarily forget about our original problem A and the non-deterministic algorithm B.
- Instead, we shall try to see whether C is satisfiable or not.
- If C is satisfiable, we say that the answer of problem A is "yes"; otherwise, the answer is "no".
- We can do this because of the property of formula C, that is, C is satisfiable if and only if B terminates successfully.

Main spirit of Cook's theorem (cont'd)

How to construct a Boolean formula from an NP algorithm?

Heart of Cook's theorem:

If the SAT problem can be solved in polynomial number of steps, then every NP problem can be solved in polynomial number of steps.

- In fact, the above approach is valid when we always can construct a Boolean formula C from an NP algorithm B such that C is satisfiable if and only if B terminates successfully.
- The question is how can we do this?

Searching problem: Case 1

Searching problem:

Given a set $S = \{x(1), x(2), \ldots, x(n)\}$ of n numbers, determine whether there exists a number in S that is equal to a number, say 7.

Example 1 (searching problem with answer "yes"):

Assume that $n = 2, x(1) = 7$ and $x(2) \neq 7$.

Non-deterministic algorithm of the searching problem:

$i =$ choice$(1, 2)$
If $x(i) = 7$ then SUCCESS
else FAILURE.

Searching problem: Case 1 (cont'd)

Constructing a Boolean formula

$$
\begin{aligned}
& i = 1 \vee i = 2 \\
\& \quad & i = 1 \rightarrow i \neq 2 \\
\& \quad & i = 2 \rightarrow i \neq 1 \\
\& \quad & (x(1) = 7 \ \& \ i = 1) \rightarrow \text{SUCCESS} \\
\& \quad & (x(2) = 7 \ \& \ i = 2) \rightarrow \text{SUCCESS} \\
\& \quad & (x(1) \neq 7 \ \& \ i = 1) \rightarrow \text{FAILURE} \\
\& \quad & (x(2) \neq 7 \ \& \ i = 2) \rightarrow \text{FAILURE} \\
\& \quad & \text{FAILURE} \rightarrow -\text{SUCCESS} \\
\& \quad & \text{SUCCESS} \quad \text{(To guarantee a successful termination)} \\
\& \quad & x(1) = 7 \quad \text{(Input data)} \\
\& \quad & x(2) \neq 7 \quad \text{(Input data)}
\end{aligned}
$$

Searching problem: Case 1 (cont'd)

Transforming into a CNF Boolean formula

$$
\begin{aligned}
& i = 1 \vee i = 2 && (1)\\
\& \quad & i \neq 1 \vee i \neq 2 && (2)\\
\& \quad & x(1) \neq 7 \vee i \neq 1 \vee \text{SUCCESS} && (3)\\
\& \quad & x(2) \neq 7 \vee i \neq 2 \vee \text{SUCCESS} && (4)\\
\& \quad & x(1) = 7 \vee i \neq 1 \vee \text{FAILURE} && (5)\\
\& \quad & x(2) = 7 \vee i \neq 2 \vee \text{FAILURE} && (6)\\
\& \quad & -\text{FAILURE} \vee -\text{SUCCESS} && (7)\\
\& \quad & \text{SUCCESS} && (8)\\
\& \quad & x(1) = 7 && (9)\\
\& \quad & x(2) \neq 7 && (10)
\end{aligned}
$$

Searching problem: Case 1 (cont'd)

Solving the CNF Boolean formula

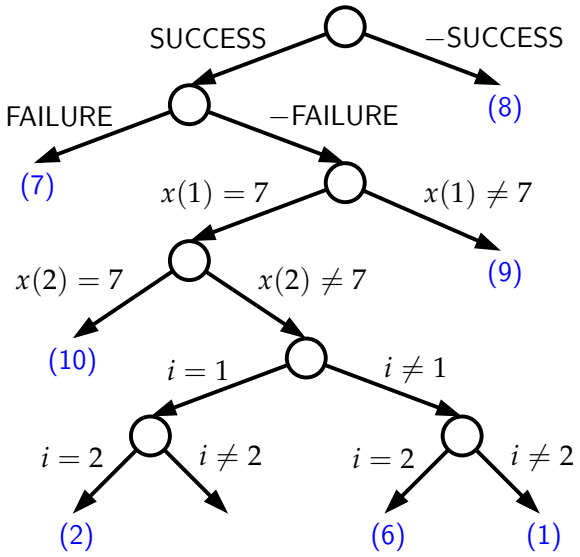
A satisfiable assignment for the previous CNF Boolean formula:

$i = 1$	satisfying	(1)
$i \neq 2$	satisfying	$(2), (4)$ and (6)
SUCCESS	satisfying	$(3), (4)$ and (8)
$-$FAILURE	satisfying	(7)
$x(1) = 7$	satisfying	(5) and (9)
$x(2) \neq 7$	satisfying	(4) and (10)

- The above satisfiable assignment tells us that the NP algorithm for the searching problem will terminate successfully and return "yes", since the literal "SUCCESS" is true.
- It also tells us the reason why the answer is "yes", which can be found in the assignment (i.e., $i = 1$ and $x(1) = 7$).

Searching problem: Case 1 (cont'd)

Corresponding semantic tree



Searching problem: Case 2

Example 2 (searching problem with answer "no"):

Assume that $n = 2$ and none of the two elements is equal to 7 (i.e., $x(1) \neq 7$ and $x(2) \neq 7$).

Non-deterministic algorithm of the searching problem:

$i = \text{choice}(1, 2)$
If $x(i) = 7$ then SUCCESS
else FAILURE.

- In this instance, the non-deterministic algorithm will terminate unsuccessfully.

Searching problem: Case 2 (cont'd)

Constructing a Boolean formula

$$
\begin{aligned}
& i = 1 \vee i = 2 \\
\& \quad & i = 1 \rightarrow i \neq 2 \\
\& \quad & i = 2 \rightarrow i \neq 1 \\
\& \quad & x(1) = 7 \ \& \ i = 1 \rightarrow \text{SUCCESS} \\
\& \quad & x(2) = 7 \ \& \ i = 2 \rightarrow \text{SUCCESS} \\
\& \quad & x(1) \neq 7 \ \& \ i = 1 \rightarrow \text{FAILURE} \\
\& \quad & x(2) \neq 7 \ \& \ i = 2 \rightarrow \text{FAILURE} \\
\& \quad & \text{FAILURE} \rightarrow -\text{SUCCESS} \\
\& \quad & \text{SUCCESS} \\
\& \quad & x(1) \neq 7 \\
\& \quad & x(2) \neq 7
\end{aligned}
$$

Searching problem: Case 2 (cont'd)

Transforming into a CNF Boolean formula

$$
\begin{aligned}
& i = 1 \vee i = 2 && (1) \\
\& \quad & i \neq 1 \vee i \neq 2 && (2) \\
\& \quad & x(1) \neq 7 \vee i \neq 1 \vee \text{SUCCESS} && (3) \\
\& \quad & x(2) \neq 7 \vee i \neq 2 \vee \text{SUCCESS} && (4) \\
\& \quad & x(1) = 7 \vee i \neq 1 \vee \text{FAILURE} && (5) \\
\& \quad & x(2) = 7 \vee i \neq 2 \vee \text{FAILURE} && (6) \\
\& \quad & -\text{FAILURE} \vee -\text{SUCCESS} && (7) \\
\& \quad & \text{SUCCESS} && (8) \\
\& \quad & x(1) \neq 7 && (9) \\
\& \quad & x(2) \neq 7 && (10)
\end{aligned}
$$

Searching problem: Case 2 (cont'd)

Solving the CNF Boolean formula

- The above clauses are unsatisfiable, because they will deduce an empty clause.

$$
\begin{aligned}
(9) \text{ and } (5) &\Rightarrow i \neq 1 \vee \text{FAILURE} && (11) \\
(10) \text{ and } (6) &\Rightarrow i \neq 2 \vee \text{FAILURE} && (12) \\
(7) \text{ and } (8) &\Rightarrow -\text{FAILURE} && (13) \\
(13) \text{ and } (11) &\Rightarrow i \neq 1 && (14) \\
(13) \text{ and } (12) &\Rightarrow i \neq 2 && (15) \\
(14) \text{ and } (1) &\Rightarrow i = 2 && (16) \\
(15) \text{ and } (16) &\Rightarrow \text{empty clause} && (17)
\end{aligned}
$$

Meaning of Cook's theorem

- From the above informal description, we can understand the meaning of Cook's theorem.
- For every NP problem A, we can transform its corresponding NP algorithm B to a Boolean formula C in a way such that C is satisfiable if and only if B terminates successfully (i.e., returning an answer "yes").
- Furthermore, it takes polynomial number of steps to complete this transformation.

Meaning of Cook's theorem (cont'd)

- Thus, if we are able to determine the satisfiability of a Boolean formula C in polynomial number of steps, we can say definitely that the answer of problem A is "yes" or "no".
- It is equivalent to say that if the SAT problem can be solved in polynomial number of steps, every NP problem can be solved in polynomial number of steps.
- In other words, if the SAT problem is in P, then NP = P.

Important notes of Cook's theorem

- Notice that Cook's theorem is valid under a constraint: It takes polynomial number of steps to transform an NP problem into a corresponding Boolean formula.
- Although we can construct a Boolean formula to describe the original problem, we are still unable to easily solve the original problem, since the satisfiability of the Boolean formula cannot be determined easily.

Important notes of Cook's theorem (cont'd)

- A non-deterministic algorithm ignores the time needed to find this solution as it claims that it always makes a correct guess.
- But, a deterministic algorithm cannot ignore this time.
- Cook's theorem tells us that the SAT problem is a very difficult problem among all NP problems, because if it can be solved in polynomial time, all NP problems can be solved in polynomial time.
- Is the SAT problem the only problem in NP with this property?

Class of NP-complete problems

- There is a class of problems that are equivalent to one another in the sense that if any of them can be solved in polynomial time, then all NP problems can be solved in polynomial time.
- They are called the class of NP-complete problems.

Problem reduction

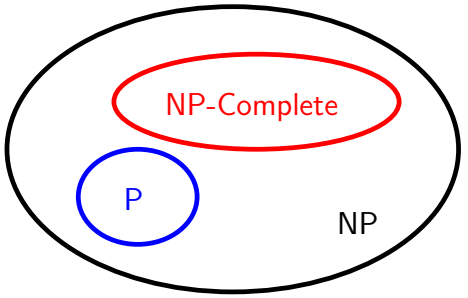
Definition:

- Let A_1 and A_2 be two problems.
- A_1 reduces to A_2, written as $A_1 \propto A_2$, if and only if A_1 can be solved in polynomial time by using a polynomial time algorithm which solves A_2.
- By definition, if $A_1 \propto A_2$ and there exists a polynomial time algorithm to solve A_2, there is a polynomial time algorithm to solve A_1.
- Every NP problem reduces to the satisfiability problem by the Cook's theorem.

NP-complete problem

Definition:

A problem A is NP-complete if $A \in$ NP and every NP problem reduces to A.



- If an NP-complete problem can be solved in polynomial time, then every NP problem can be solved in polynomial time.
- The SAT problem is an NP-complete problem by the Cook's theorem.

Transitive property of problem reduction

- Actually, to show that a problem A is NP-complete, we do not have to prove that all NP problems reduce to A.
- This is what Cook did when he showed the NP-completeness of the SAT problem.
- Now, we only have to use the transitive property of "$\propto$".

Transitive property of "reduce to" ("$\propto$"):

If $A_1 \propto A_2$ and $A_2 \propto A_3$, then $A_1 \propto A_3$.

Transitive property of problem reduction (cont'd)

Method to show NP-completeness of a problem:

If A_1 is an NP-complete problem, A_2 is an NP problem and we can prove that $A_1 \propto A_2$, then A_2 is an NP-complete problem.

- Because A_1 is an NP-complete problem, all NP problems $\propto A_1$.
- Because all NP problems $\propto A_1$ and $A_1 \propto A_2$, all NP problems $\propto A_2$.
- Because all NP problems reduce to A_2 and A_2 is also an NP problem, A_2 is an NP-compelte problem.

Difficulty equivalence for NP-complete problems

- If $A \propto$ the SAT problem and the SAT problem $\propto A$, then A is equivalent to the SAT problem when the degree of difficulty is concerned.
- The difficulty degrees of A and the SAT problem are equivalent.
- All NP-complete problems form an equivalent class.

NP-hard

- We have restricted NP problems to be decision problems.
- Actually, we can extend the concept of the NP-completeness to optimization problems by defining "NP-hardness".

Definition:
A problem A is NP-hard if every NP problem reduces to A.

- Note that A is not necessarily an NP problem and it actually may be an optimization problem.
- A problem is NP-complete if it is NP-hard and it is an NP.
- Hence, an optimization problem is NP-hard if its corresponding decision problem is NP-complete.

NP-hardness of halting problem

Theorem:
SAT problem $\propto$ halting problem.

Algorithm $\mathcal{A}(F)$, where F is a Boolean formula:

1. **for** each assignment a among 2^n possible assignments **do**
   if a satisfies F **then** $\mathcal{A}$ stops;
   end for
2. $\mathcal{A}$ enters an infinite loop;

- Clearly, if algorithm $\mathcal{A}$ stops, then F is satisfiable; otherwise, F is unsatisfiable.
- As a result, the halting problem is NP-hard.

How to prove NP-completenes of a problem?

Two steps to prove that a problem A is NP-complete:

1. We first prove that A is an NP problem.
2. We then prove that some NP-complete problem reduces to A.

3-Satisfiability (3-SAT) problem

Definition:
The 3-SAT problem is similar to the SAT problem with one more restriction: every clause contains exactly three literals.

Example:

$$\begin{array}{cccccc} & x_1 & \vee & -x_2 & \vee & -x_3 \\ \& & -x_1 & \vee & x_2 & \vee & -x_3 \\ \& & -x_1 & \vee & -x_2 & \vee & x_3 \end{array}$$

- It is obvious that the 3-SAT problem is an NP problem.

3-SAT problem is NP-complete

- We shall prove that SAT problem $\propto$ 3-SAT problem.
- That is, for every Boolean formula F_1 in the SAT problem, we can create another Boolean formula F_2, in which every clause has exactly three literals, such that F_1 is satisfiable if and only if F_2 is satisfiable.
- Example 1 Extend $F_1 = (1)\ \&\ (2)$ into $F_2 = (1')\ \&\ (2')$:

$F_1 = (1)\ \&\ (2)$:

$$\begin{array}{cccc} x_1 & \vee & x_2 & (1) \\ -x_1 & & & (2) \end{array}$$

$F_2 = (1')\ \&\ (2')$:

$$\begin{array}{cccccc} x_1 & \vee & x_2 & \vee & y_1 & (1') \\ -x_1 & \vee & y_2 & \vee & y_3 & (2') \end{array}$$

- Note that $(1)\ \&\ (2)$ is satisfiable $\Rightarrow$ $(1')\ \&\ (2')$ is satisfiable.

3-SAT problem is NP-complete (cont'd)

- Example 2 Extend $F_1 = (1)\ \&\ (2)$ into $F_2 = (1')\ \&\ (2')$:

$F_1 = (1)\ \&\ (2)$:

$$\begin{array}{cc} x_1 & (1) \\ -x_1 & (2) \end{array}$$

$F_2 = (1')\ \&\ (2')$:

$$\begin{array}{cccccc} x_1 & \vee & y_1 & \vee & y_2 & (1') \\ -x_1 & \vee & y_3 & \vee & y_4 & (2') \end{array}$$

- $(1)\ \&\ (2)$ is unsatisfiable, but $(1')\ \&\ (2')$ is satisfiable.
- Hence, the above discussion shows us that we cannot arbitrarily add new literals to a formula without affecting its satisfiability.
- What we can do is to append new clauses which themselves are unsatisfiable to the original set of clauses.

SAT $\propto$ 3-SAT

- If the original clause contains only one literal L_1, we append it with the following four clauses:

$$\begin{array}{ccc} y_1 & \vee & y_2 \\ -y_1 & \vee & y_2 \\ y_1 & \vee & -y_2 \\ -y_1 & \vee & -y_2 \end{array}$$

Example:

- Suppose the original clause is x_1.
- Then the newly created clauses are as follows:

$$\begin{array}{ccccc} x_1 & \vee & y_1 & \vee & y_2 \\ x_1 & \vee & -y_1 & \vee & y_2 \\ x_1 & \vee & y_1 & \vee & -y_2 \\ x_1 & \vee & -y_1 & \vee & -y_2 \end{array}$$

SAT $\propto$ 3-SAT (cont'd)

- If the original clause contains two literals, we append it with the following two clauses:

$$y_1$$
$$-y_1$$

Example:

- Suppose the original clause is $x_1 \vee x_2$.
- Then the newly created clauses are as follows:

$$x_1 \vee x_2 \vee y_1$$
$$x_1 \vee x_2 \vee -y_1$$

SAT $\propto$ 3-SAT (cont'd)

- If the original clause contains three literals, we do nothing.
- If the original clause contains more than three literals, we break it into new clauses by appending the following clauses:

$$\begin{aligned} y_1 & \\ -y_1 & \vee y_2 \\ -y_2 & \vee y_3 \\ \vdots & \vdots \quad \vdots \\ -y_{i-1} & \vee y_i \\ -y_i & \end{aligned}$$

SAT $\propto$ 3-SAT (cont'd)

Example:

- Consider the clause: $x_1 \vee -x_2 \vee x_3 \vee x_4 \vee -x_5$.
- We then add two new varaibles y_1 and y_2 to create three new clauses containing exactly three literals:

$$x_1 \vee -x_2 \vee y_1$$
$$x_3 \vee -y_1 \vee y_2$$
$$x_4 \vee -x_5 \vee -y_2$$

SAT $\propto$ 3-SAT (cont'd)

Theorem:

- Let S denote the original set of clauses in the SAT problem.
- Let S' denote the set of transformed clauses with each clause containing three literals.
- S is satisfiable if and only if S' satisfiable.

Proof: ($\Rightarrow$) If S is satisfiable, then S' is satisfiable.

- Let I be an assignment satisfying S.
- I satisfies all clauses of S' that are created from the clauses of S containing ≤ 3 literals.
- Let C be a clause in S with > 3 literals.
- Let $T(C)$ be the set of clauses transformed from C.

SAT $\propto$ 3-SAT (cont'd)

Proof: ($\Rightarrow$) If S is satisfiable, then S' is satisfiable.

- Example Let $C = x_1 \vee x_2 \vee -x_3 \vee x_4 \vee -x_5$. Then

$$T(C) = \begin{cases} x_1 \vee x_2 \vee y_1 \\ -x_3 \vee -y_1 \vee y_2 \\ x_4 \vee -x_5 \vee -y_2 \end{cases}$$

- Note that I must satisfy at least one clause, say C_i, of $T(C)$.
- Expand I into I' such that I' satisfies all clauses of $T(C)$ by repeating the following process until every clause of $T(C)$ is satisfied by I':

Process to expand I into I':

Assign the last literal (or last two literals) of C_i false, which will continue to satisfy another clause C_j in $T(C)$, and repeat this process until every clause is satisfied.

SAT $\propto$ 3-SAT (cont'd)

Proof: ($\Rightarrow$) If S is satisfiable, then S' is satisfiable.

- Example Let $C = x_1 \vee x_2 \vee -x_3 \vee x_4 \vee -x_5$. Then

$$T(C) = \begin{cases} x_1 \vee x_2 \vee y_1 & (1) \\ -x_3 \vee -y_1 \vee y_2 & (2) \\ x_4 \vee -x_5 \vee -y_2 & (3) \end{cases}$$

- Suppose that $I = \{x_1\}$ satisfies (1) by letting $x_1 \leftarrow \mathit{true}$.
- Let $I' = I$.
- $y_1 \leftarrow \mathit{false}$ satisfies (2) $\Rightarrow I' = \{x_1, -y_1\}$.
- $y_2 \leftarrow \mathit{false}$ satisfies (3) $\Rightarrow I' = \{x_1, -y_1, -y_2\}$.
- Clearly $I' = \{x_1, -y_1, -y_2\}$ satisfies $T(C)$.

SAT $\propto$ 3-SAT (cont'd)

Proof: ($\Leftarrow$) If S' is satisfiable, then S is satisfiable.

- Note that for S', the newly appended clauses are unsatisfiable.
- Hence, the assignment satisfying S' cannot contain y_i's only.
- Example Let $C = x_1 \vee x_2 \vee -x_3 \vee x_4 \vee -x_5$. Then

$$T(C) = \begin{cases} x_1 \vee x_2 \vee y_1 & (1) \\ -x_3 \vee -y_1 \vee y_2 & (2) \\ x_4 \vee -x_5 \vee -y_2 & (3) \end{cases}$$

- In other words, any assignment satisfying $T(C)$ must satisfy at least one literal of C.