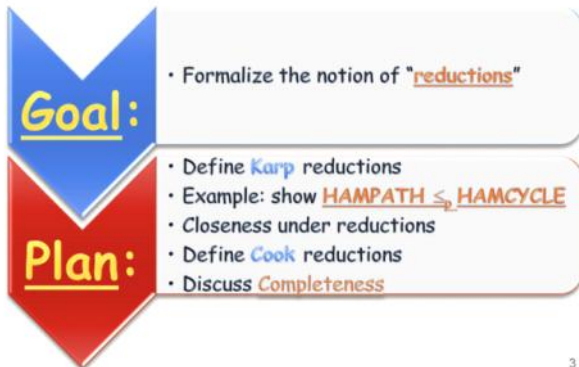


NPC notes

This lecture is about NP-Completeness, and has three parts:

- Reductions,
- the Cook-Levin Theorem
- and NPC problems.

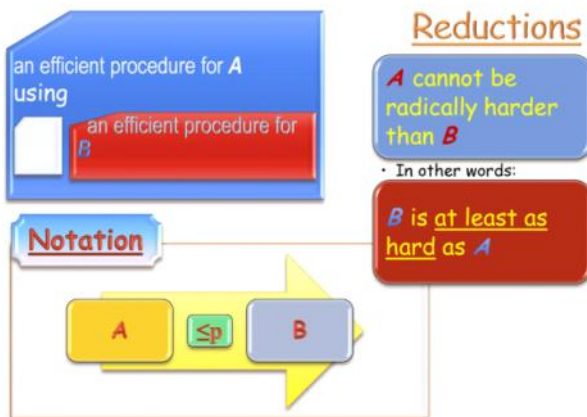
Let us now discuss in more details how to use reductions to bound problems' complexities.



3

We'll discuss:

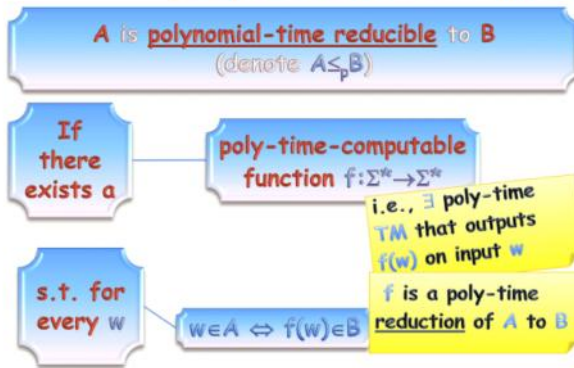
- Karp reductions,
- closeness of classes under reductions,
- and mention also a more general type of reductions.



4

Recall that the general type of reductions presented earlier, from a problem **A** to a problem **B**, required us to show a procedure for **A**, which calls on a procedure for **B**, and so that assuming an efficient procedure for **B**, the procedure for **A** is also efficient.

Karp reductions -Definition

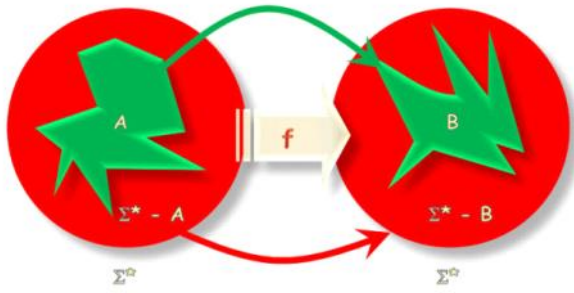


5

Let us now define a special case of these reductions, referred to as a Karp reduction.


In this type of reduction, one constructs an efficient reduction-function, which translates an instance of the problem A to an instance of the problem B, while maintaining the two outcomes are the same.

Karp Reductions -Illustrated



6

Namely, the reduction-function results, for any instance of the language A, with an instance of the language B; while for any input outside the language A, the reduction returns a string outside the language B.




To Do:

- Come up with a reduction-function f
- Show f is polynomial time computable
- Prove f is a reduction, i.e., show:
 - $w \in A \implies f(w) \in B$
 - $w \in \bar{A} \implies f(w) \in \bar{B}$

• We'll use reductions that, by default, would be of this type, which is called:

- Polynomial-time mapping reduction
- Polynomial-time many-one reduction
- Polynomial-time Karp reduction

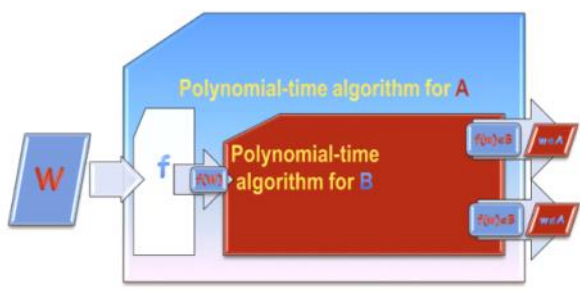
Reducing



Hence, for a reduction of that type to be proper, one has to show it is efficient and prove its soundness and completeness.


All reductions we'll present in the course, by default, will be of that type.

Reductions and Efficiency

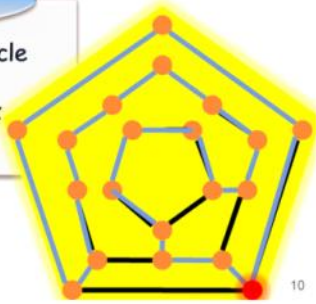


Let us now make sure that such a reduction implies that an efficient procedure for B entails an efficient procedure for A:

On input W we apply the reduction-function, then apply a procedure for B on its output, and simply return the outcome of that application.

<p><u>Hamiltonian Path Instance:</u></p> <ul style="list-style-type: none"> • A directed graph $G=(V,E)$ <p><u>Decision Problem:</u></p> <ul style="list-style-type: none"> • Is there a path in G, which goes through every vertex exactly once?  <p style="text-align: right;">9</p>	<p>Formally define Hamiltonian path.</p>
---	--



<p><u>Hamiltonian Cycle Instance:</u></p> <ul style="list-style-type: none"> • a directed graph $G=(V,E)$. <p><u>Decision Problem:</u></p> <ul style="list-style-type: none"> • Is there a simple cycle in G that paths through each vertex exactly once?  <p style="text-align: right;">10</p>	<p>Formally define Hamiltonian cycle.</p>
--	---

HAMPATH \leq_p HAMCYCLE

$$f((V, E)) = ((V \cup \{u\}, E \cup V \times \{u\}))$$

Completeness: →

- Given a Hamiltonian path (v_0, \dots, v_n) in G , (v_0, \dots, v_n, u) is a Hamiltonian cycle in G'

← **Soundness:**

- Given a Hamiltonian cycle (v_0, \dots, v_n, u) in G' , removing u yields a Hamiltonian path.

Let us now revisit the reduction, from Hamiltonian-path to Hamiltonian-cycle, previously described.

We simply add to the graph an extra vertex, which is adjacent to all other vertexes.

The completeness proof as well as a soundness proof are easy.

Check list

- ✓ Come up with a reduction-function f
- ✓ Show f is polynomial time computable

Prove f is a reduction, i.e., show:

- ✓ $w \in \text{HAMPATH} \rightarrow f(w) \in \text{HAMCYCLE}$
- ✓ $w \in \text{HAMPATH} \rightarrow f(w) \in \text{HAMCYCLE}$

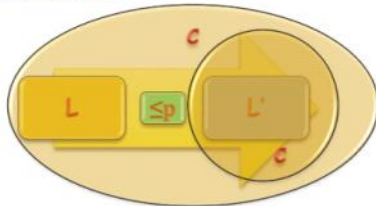
Let us now go over the checklists for making sure the reduction is proper:

- We have described the simple reduction function.
- Is it efficient? It clearly is.
- We also proved both its soundness and completeness.

Closeness Under Reductions: Definition

A complexity class C is **closed under poly-time reductions** if:

- L is reducible to L' and $L' \in C \Rightarrow L$ is also in C .



13

Now that we have formally defined the notion of an efficient reduction, we may consider classes that are closed under such reductions.

Some classes are possibly not closed under efficient reductions: It may be the case that we are able to efficiently reduce one language, not in the class C , to another language, which is in the class C .

Can you think of a class for which this could potentially happen?

Observation

Theorem:

- P , NP , $PSPACE$ and $EXPTIME$ are closed under polynomial-time Karp reductions

Proof:

- Do it yourself !!



14

Some of the classes we have defined so far are closed under efficient reductions.

Prove it!

Log-space Reductions

A is **log-space reducible** to B
(denote $A \leq_L B$)

If there exists a

log-space-computable function $f: \Sigma^* \rightarrow \Sigma^*$

i.e., \exists log-space TM that outputs $f(w)$ on input w

s.t. for every w

$w \in A \Leftrightarrow f(w) \in B$ f is a log-space reduction of A to B

Theorem:

- L , NL , P , NP , $PSPACE$ and $EXPTIME$ are closed under log-space reductions.

We can consider an even more efficient type of reductions, namely, those that can be carried out using only logarithmic size memory.

Can you think of a reduction that follows these guidelines?
Can you show that even more classes are closed under such reductions?
Is it clear that these reductions do what we expect them to do?
How does such a reduction output its outcome?



Reductions: General

Cook Reduction:

- Assuming an efficient procedure that decides B, construct one for A.

an efficient procedure for A using

an efficient procedure for B

Karp is a special case of Cook reduction:

It allows only 1 call to B, whose outcome must be outputted as is

16

Karp reduction is a special case of the general, Cook reduction. It insists that the procedure for B is called only once, and that the outcome is simply returned as is.

It is important to note that from now on we will use only that type of reduction for our definitions!

Some of the notions we will introduce do not make sense for the more general case!

Cook red. : HAMCYCLE \rightarrow HAMPATH.

- 1 Let $E' = E$
- 2 If $E' = \emptyset$ **reject**
- 3 choose (any) $\langle u, v \rangle$ in E'
- 4 If HAMPATH ($\langle V + \{w, z\}, E' + \{\langle w, u \rangle, \langle v, z \rangle\} \rangle$) **accept**
- 5 $E' = E' - \{\langle u, v \rangle\}$
- 6 Go to step 2

Here is a simple example of a Cook reduction, for the reduction in the other direction, from Hamiltonian cycle to Hamiltonian path.

Definition: C-complete **Completeness**

- For a class C of decision problems and a language $L \in C$, L is **C-complete** if:
 $L' \in C \Rightarrow L'$ is reducible to L .

Theorem:
• L is complete for classes $C, C' \Rightarrow C = C'$

Proof:
• All languages in C and in C' are reducible to L , which is in both. Since both are closed under reductions, they're the same ■

Theorem:
• Any $L \in NPC, L \in P \Rightarrow P = NP$

Now we're ready to define what it means for a problem to be complete for a class. A problem is complete for a class, if all problems in that class can be efficiently reduced to it.

Such a problem then becomes a representative of that class, in particular, if the problem is complete to more than one class, those classes must be the same.

It follows that: if any NP-complete language turns out to be in P, then $NP = P$!

Summary



Discussed types of **reductions**:

- Cook vs. Karp reductions
- Poly-time vs. log-space



Defined:

"completeness"

Find L and C s.t. L is C -complete



Discussed a way to show:

equality between **complexity classes**

The Cook/
Levin
theorem:

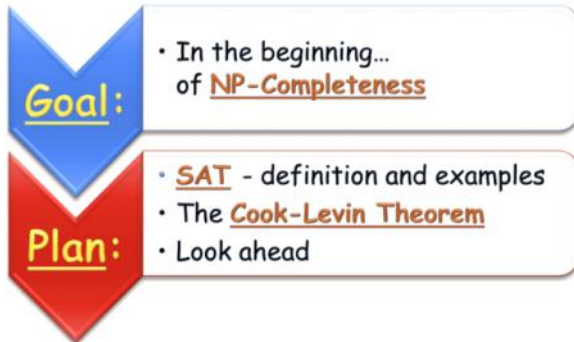


SAT is NP-Complete:



We're now going to prove one of the most basic Theorems of computer science --- proved by S. Cook and independently by L. Levin.

We're also going to see our first NP-complete problem.

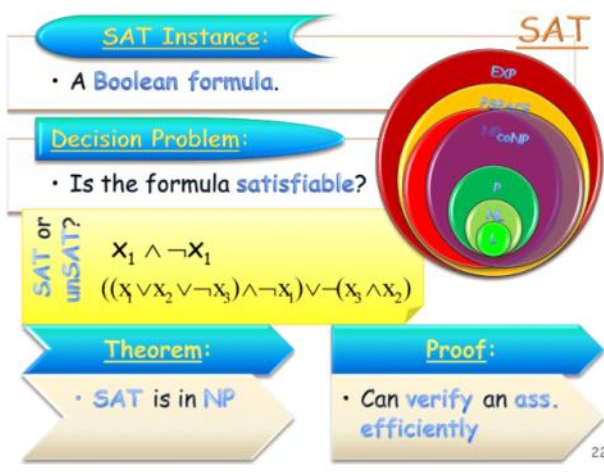


21

We'll define the SAT problem, and then proceed to prove that it is NP-complete.



[Cook-Levin Theorem](#)
[NP Completeness](#)



A SAT formula is a Boolean formula over Boolean variables.

The decision problem corresponding to it is, given such a formula as input, whether there exists an assignment to the variables that causes the formula to evaluate to true.

SAT is clearly in NP.

Cook/Levin

Theorem:

- SAT is NP-Complete

Proof Outline:

- Given an NP machine M and an input w , construct a Boolean formula $\varphi_{M,w}$.
 $\varphi_{M,w}$ satisfiable $\Leftrightarrow M$ accepts w .



SAT is, moreover, NP hard, which is a much more fundamental statement.

It being both in NP and NP hard, makes it NP-complete.

The proof proceeds by, given a TM M and any input string W , constructing a SAT formula (which depends on both M and W) that is satisfied if and only if the TM M accepts the string W .



A computation of a (non deterministic) TM can be described in a table, where the i 'th row corresponds to the configuration of the machine after i steps.

To describe a configuration, one specifies the content of each cell, as well as the machine's state, written (in our convention) to the left of where the machine's head is located.

For an NP TM, the size of the table is polynomial in the size of the input.

$Q = \{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\}$

$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, _ \}$

δ

- $\delta(q_0, 1) = \{(q_1, _R)\}$
- $\delta(q_1, 1) = \{(q_0, _R)\}$
- $\delta(q_0, 0) = \{(q_0, _R)\}$
- $\delta(q_1, 0) = \{(q_1, _R)\}$
- $\delta(q_0, _) = \{(q_{\text{accept}}, _L)\}$
- $\delta(q_1, _) = \{(q_{\text{reject}}, _L)\}$

Example

#	q_0	0	1	1	1	#
#	q_0	1	1	1	#	#
#	q_1	1	1	1	#	#
#	q_0	1	1	1	#	#
#	q_1	1	1	1	#	#
#	q_0	1	1	1	#	#
#	q_1	1	1	1	#	#

Let's see an example for a configurations' table for a very simple TM.

Can you say what language this TM accepts?

Go over this table and convince yourself that it is indeed legal, assuming the first configuration correctly corresponds to a given input.

$Q = \{q_0, q_1, q_{\text{accept}}, q_{\text{reject}}\}$

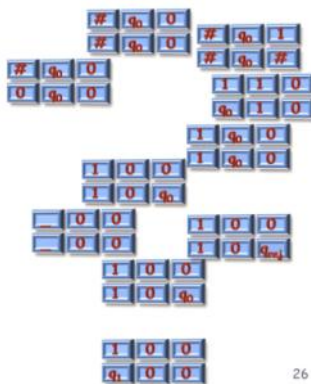
$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, _ \}$

δ

- $\delta(q_0, 1) = \{(q_1, _R)\}$
- $\delta(q_1, 1) = \{(q_0, _R)\}$
- $\delta(q_0, 0) = \{(q_0, _R)\}$
- $\delta(q_1, 0) = \{(q_1, _R)\}$
- $\delta(q_0, _) = \{(q_{\text{accept}}, _L)\}$
- $\delta(q_1, _) = \{(q_{\text{reject}}, _L)\}$

$$\Delta_M \subset (\Gamma \cup Q \cup \{\#\})^6$$



Let us concentrate for a moment on a 3 by 2 window of the configurations' table.

Which of the listed examples is legal? To figure that out systematically, one should start from a legal combination of five entries, and apply all possible options for these cells in the next configuration. If the machines head is nowhere in those five entries, the middle three entries should be copied as is. Otherwise, apply all possible transitions and register all possible combinations for the middle three entries.

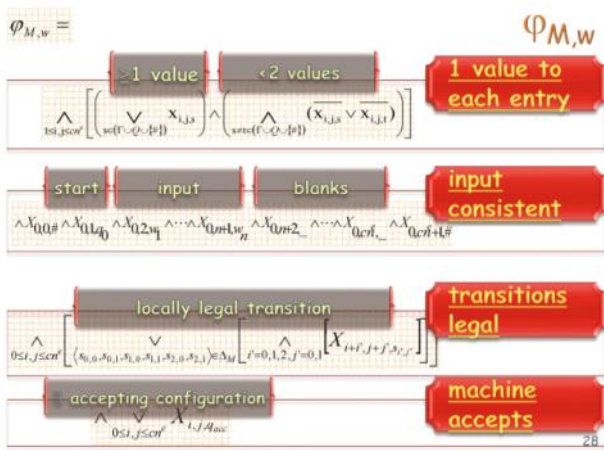
Clearly, the description of a legal computation would have all the local windows legal.

You should convince yourself that a table of which all local windows are legal indeed corresponds to a legal computation.



A SAT formula however has Boolean variables.

The Boolean variables of the formula constructed in the reduction are as follows:
 each corresponds to an entry of the table plus a potential value for that entry.



We are now ready to describe the formula that results from the reduction.

The first part of the formula verifies that the value assigned to the Boolean variables corresponds to an assignment of one value to each entry of the table.

The second part of the formula verifies that the first row of the table is legal and moreover that it corresponds to the input string W.

The third part of the formula verifies that all local windows are legal.

The fourth and last part of the formula verifies that the computation enters an accepting state.

$$\varphi_{M,w} = \bigwedge_{tsi,jscnf} \left[\left(\bigvee_{s \in \{r-1, r\}} x_{i,j,s} \right) \wedge \left(\bigwedge_{s \in \{r-1, r\}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,s}}) \right) \right]$$

$$\wedge x_{0,0,\#} \wedge x_{0,1,\#} \wedge x_{0,2,\#} \wedge \dots \wedge x_{0,w+1,\#} \wedge x_{0,w+2,\#} \wedge \dots \wedge x_{0,cn-1,\#} \wedge x_{0,cn+1,\#}$$

$$\bigwedge_{0 \leq i,j \leq cn} \left[\bigvee_{(s_0, s_1, s_2) \in \Delta_M} \left[\bigwedge_{r=0,1,2} \left[x_{i+r, j+r, s_r} \right] \right] \right]$$

$$\wedge \bigvee_{0 \leq i,j \leq cn} x_{i,j,0}$$

Q.E.D.

Claim:

- $\forall i,j$ transition is locally legal \Leftrightarrow tableau legal

Corollary:

- $\varphi_{M,w}$ Satisfiable $\Leftrightarrow W \in L_M$

Claim:

- Size of $\varphi_{M,w}$ polynomial in $|W|$



To complete the proof, one needs to make sure that the formula can be satisfied if and only if the input is accepted, and that it is of polynomial size in the size of the input.

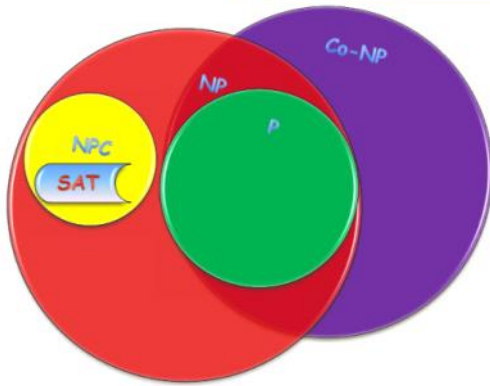
We have just shown SAT is NP-hard, as any NP language can be reduced to SAT

SAT is NPC



We have just shown that any language in NP can be efficiently reduced to SAT. This implies SAT is NP hard. Since we have already shown SAT is in NP, we conclude that SAT is NP-complete.

P, NP, co-NP and NPC



31

SAT is our first NP-complete language.

If it turns out to be in P, then the class NP and the class P are the same (and so is a class coNP).

If however SAT turns out not to be in the class P, it must be that the class P is different than the class NP. The class coNP in that case must be different than the class P, however it could still be the same as the class NP!

Henceforth, to show a problem A is NP-hard, it suffices to reduce SAT to A

NPC



32

Now that we have shown SAT is NP hard, to show other problems are NP hard, we may Karp-reduce SAT to them. We don't have to repeat this proof for every problem we wish to show NP-hard.

What would happen if we replace Karp-reduction to a more general case?

Furthermore, once we've shown A is NP-hard, we can reduce from it to show other problems NP-hard

NPC



This is true in general:
If we have proven some language to be NP hard, we may Karp-reduce it to other languages, to show them to be NP hard as well.

Summary



proved SAT is NP-Complete



Consider SAT the Genesis problem, and explored how to proceed and show other problems are NP-hard

Goal: • introduce some additional NP-Complete problems.

Plan: • 3SAT
• CLIQUE & INDEPENDENT-SET

35

We're now ready to prove some more problems are NP-complete:

We'll begin with the 3SAT problem, defined below.

Then go over CLIQUE and Independent-Set.



[Independent Set](#)

Recall: L is NPC iff

- L in NP
- L NP-hard - via Karp-reduction

SAT and NPC

So far we only showed one such problem: SAT

- which, however, is not up for the tasks ahead

Next we show a special case of SAT is NPC:

- 3SAT

3SAT Instance: Conjunctive Normal Form - 3 literals in each clause

- 3CNF formula

Decision Problem:

- Is it satisfiable?

3CNF: $(x_1 \vee y_1 \vee z_1) \wedge (x_2 \vee y_2 \vee z_2) \wedge (x_3 \vee y_3 \vee z_3)$

36

Recall that a language is NP-complete if it is in NP and is also NP-hard.

We've shown SAT is NP-hard, however, for forthcoming reductions such general formulas are not adequate.

For that purpose, we introduce a special case of SAT, namely that of 3SAT:

A 3SAT formula takes the form of CNF (= Conjunctive normal form, or in other words, an AND of OR clauses). It is further restricted so as to be a 3CNF, namely, allow only 3 literals in every OR clause.

The language 3SAT consists of all such formulas that have an assignment that satisfies them.

3SAT is NPC

Claim:

- 3SAT ∈ NP

3SAT is a special case of SAT.

Claim:

- 3SAT ∈ NP-hard

Proof:

- amend our SAT formula, so it becomes 3CNF
- First make it a CNF: use DNF → CNF on 3rd line

Does this suffice?

Are all others OK?

What is the size of new formula?

$$\varphi_{M,w} = \bigwedge_{1 \leq i, j \leq n} \left[\left(\bigvee_{s \in \{1,2,3\}} x_{i,j,s} \right) \wedge \left(\bigwedge_{s \in \{1,2,3\}} (\overline{x_{i,j,s}} \vee x_{i,j,s}) \right) \right]$$

$$\wedge x_{0,0} \wedge x_{0,1} \wedge x_{0,2} \wedge \dots \wedge x_{0,n-2} \wedge x_{0,n-1} \wedge \dots \wedge x_{0,n-1}$$

$$\bigwedge_{0 \leq i, j \leq n-1} \left[\bigvee_{(s_0, s_1, s_2) \in \Delta_M} \left[\bigwedge_{i=0,1,2, j=0,1} \left[x_{i+j, j, s_i} \right] \right] \right]$$

$$\bigwedge_{0 \leq i, j \leq n-1} x_{i, j, 0}$$

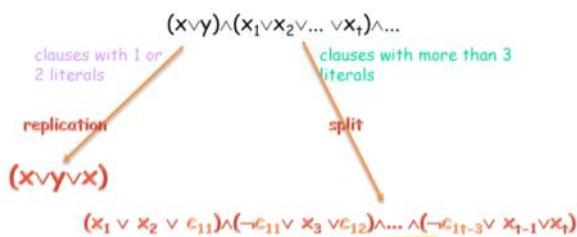
3SAT is clearly in NP as there exists a witness of membership that can be efficiently verified (even more generally, being a special case of SAT, it must be in NP).

To prove that 3SAT is NP-hard, it suffices to efficiently alter the SAT formula we had obtained previously into the proper form.

We start by converting it to a CNF formula. The only problematic part is the one that corresponds to the local windows.

Still, since for each window the formula size is constant, we can apply the DNF to CNF general (albeit with a potentially exponential blowup) translation, and be fine.

CNF → 3CNF



QED 3SAT is NP-Complete

Now we need to translate a general CNF to 3CNF.

For that purpose, one needs to replace each clause with a simple set of 3-wide clauses utilizing some extra variables, while maintaining satisfiability of the original clause.

This transformation is fairly simple

This completes the proof that 3SAT is NP-complete.

CLIQUE is NPC

CLIQUE instance:

- A graph $G=(V,E)$ and a threshold k

Decision problem:

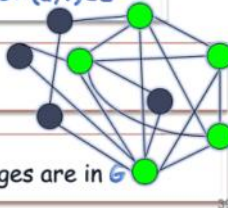
- Is there a set of nodes $C=\{v_1, \dots, v_k\} \subseteq V$, s.t. $\forall u, v \in C: (u, v) \in E$

Observation:

- $CLIQUE \in NP$

Proof:

- Given C , verify all inner edges are in G



Now let us consider the CLIQUE problem.

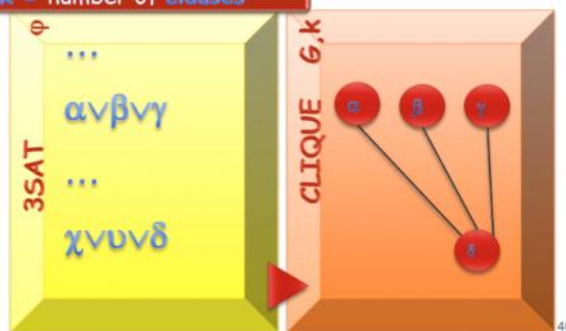
The basic question is simple: given a graph, what is the largest set of vertexes whose induced sub-graph is complete?

CLIQUE is clearly in NP: the proof of membership is simply a set of vertexes constituting a clique, which can be easily verified.

1 vertex for 1 occurrence
inconsistency \Leftrightarrow non-edge

k = number of clauses

$SAT \leq_p CLIQUE$



To show the CLIQUE problem is NP-hard, we'll reduce 3SAT to it.

The set of vertexes consists of one vertex for every occurrence of every variable in the formula.

Vertexes that correspond to the same clause are regarded as inconsistent, hence there are no edges between them.

The only other edges missing from the graph correspond to two different literals of the same variable.

The threshold for the size of the CLIQUE, k , is set to be the number of clauses of the 3SAT formula.

SAT \leq_p CLIQUE: proof



Completeness:

- Let A be a satisfying assignment to ϕ , $C(A)$ contains 1 v_a s.t. $A(v_a)$ for every clause

Soundness:

- In a clique C in G of size k , each variable has ≤ 1 of its literals-vertex in C
- extend to a satisfying assignment to ϕ

Since there are no edges between a triplet, a CLIQUE of size k must comprise one vertex in each triplet.

If there exists a satisfying assignment, one can pick one vertex for each clause, insisting it corresponds to a literal satisfied by the assignment.

These vertexes form a CLIQUE and they are all consistent (exactly one vertex for each triplet, and never a variable and its negation).

If there exists a CLIQUE of size k in the graph, every variable has at most one of its literals occurring in the CLIQUE. Assigning the variables so that those literals are TRUE (and assigning arbitrary values to all other variables) satisfies the 3SAT formula.

INDEPENDENT-SET is NPC

IS instance:

- A graph $G=(V, E)$ and a threshold k

Decision problem:

- Is there a set of nodes $I=\{v_1, \dots, v_k\} \subseteq V$, s.t. $\forall u, v \in I: (u, v) \notin E$

Observation:

- $IS \in NP$

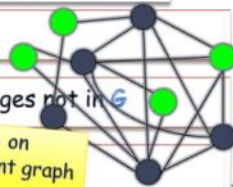
Proof:

- Given I , verify all inner edges not in G

Observation:

- IS is NP-hard

Clique=IS on complement graph

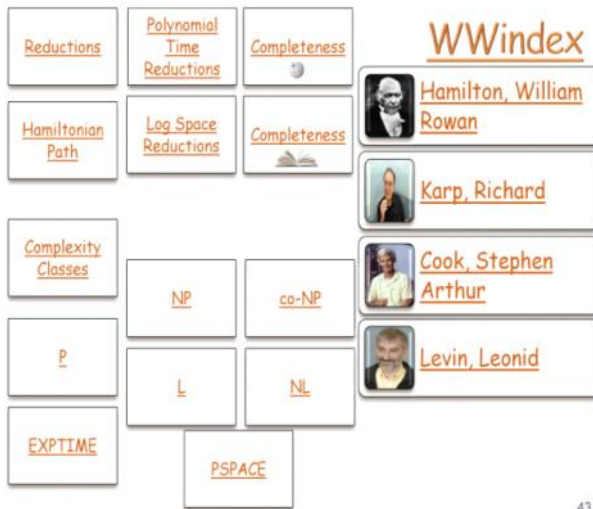


Let us now consider the problem of independence set:

Given any graph, what is the largest set of vertexes for which the induced sub-graph is empty?

The problem is clearly in NP and, in fact, also clearly NP-hard.

It is in fact the same as the CLIQUE problem only on the complement graph.



43

- [Reductions](#)
- [Polynomial Time Reductions](#)
- [Completeness](#)
- [Hamiltonian Path](#)
- [Log Space Reductions](#)
- [Complexity Classes](#)
- [P](#)
- [NP](#)
- [co-NP](#)
- [EXPTIME](#)
- [L](#)
- [NL](#)
- [PSPACE](#)
- [Completeness](#)
- [Hamilton, William Rowan](#)
- [Karp, Richard](#)
- [Cook, Stephen Arthur](#)



44

- [SAT](#)
- [Cook-Levin Theorem](#)
- [Cook-Levin Theorem](#)
- [NPC](#)
- [Clique](#)
- [Subset Sum](#)
- [CNF](#)
- [NP Hard](#)
- [Independent Set](#)
- [3SAT](#)

