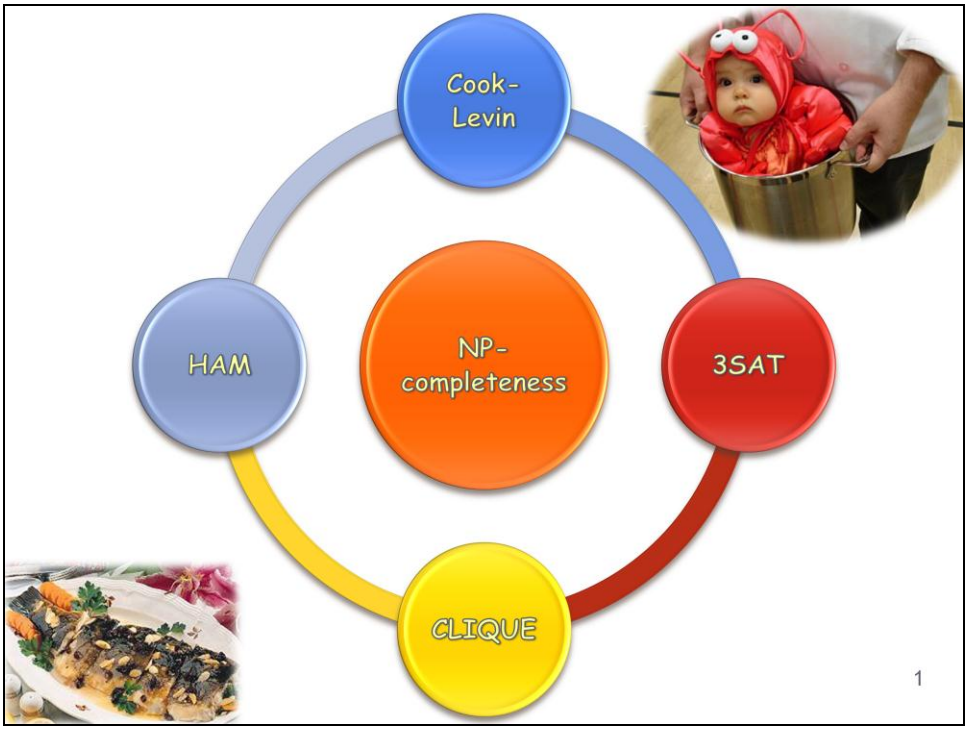
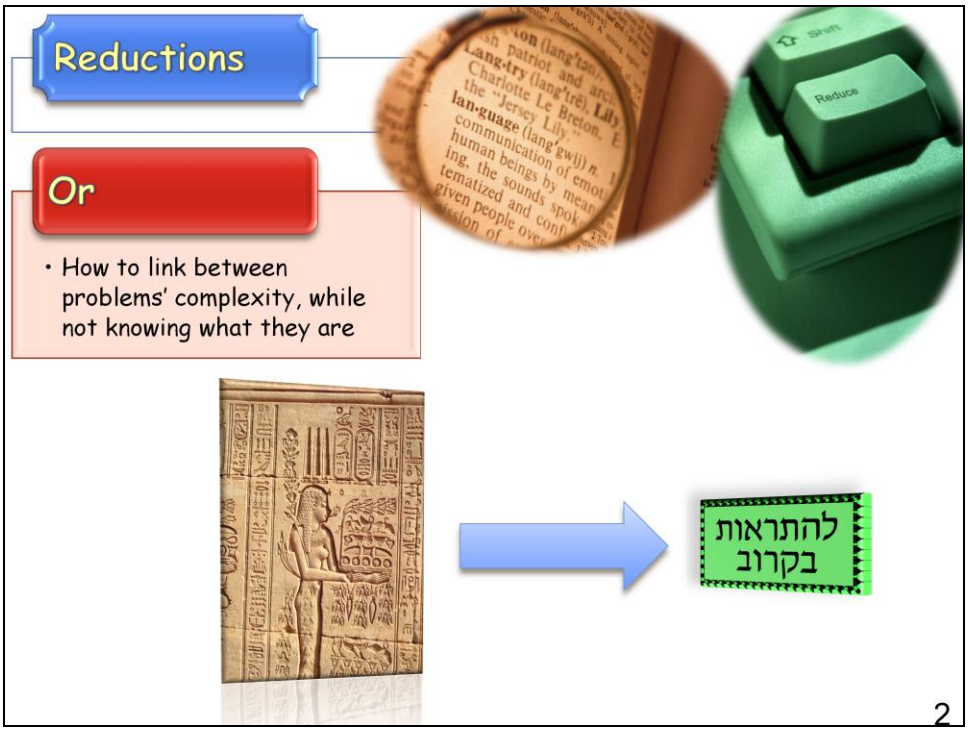
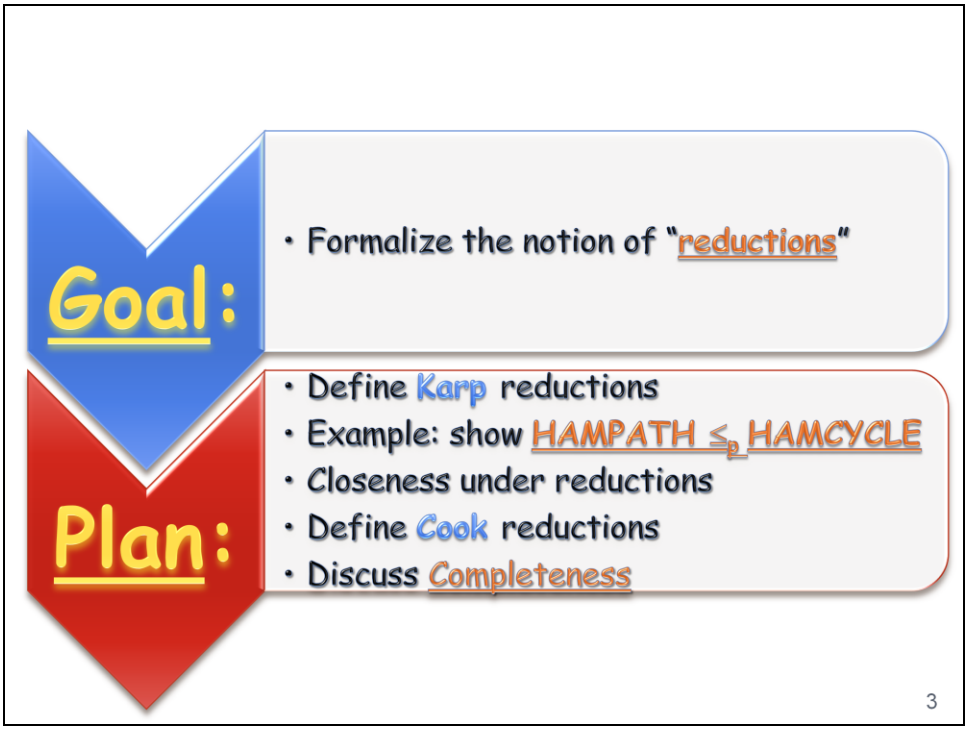


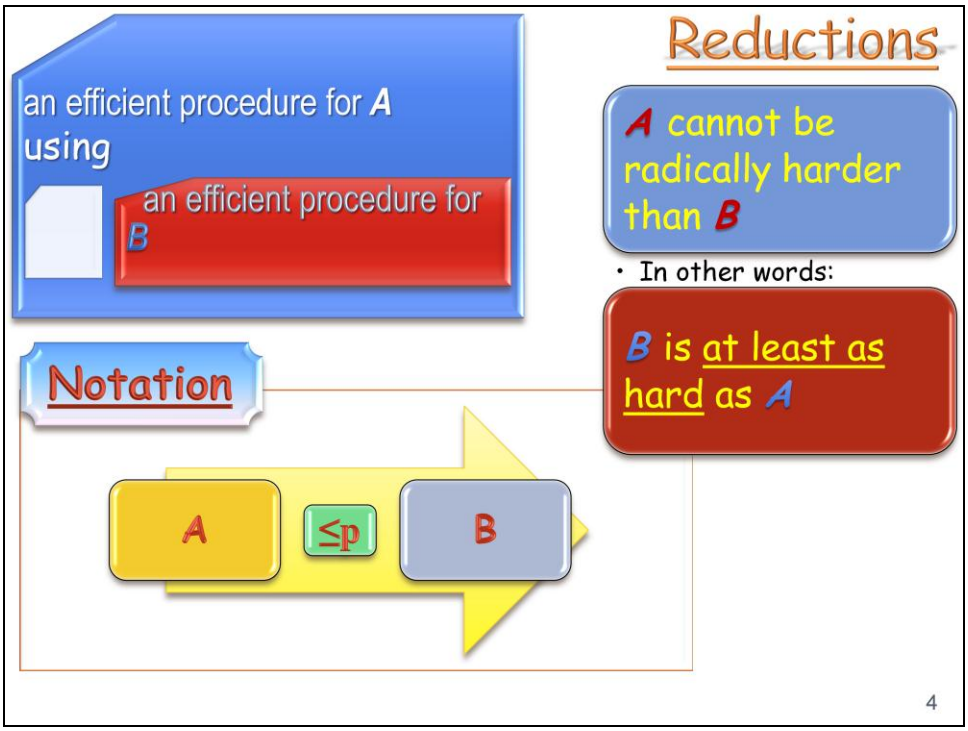
This lecture is about NP-Completeness, and has three parts: Reductions, the Cook-Levin Theorem and NPC problems.



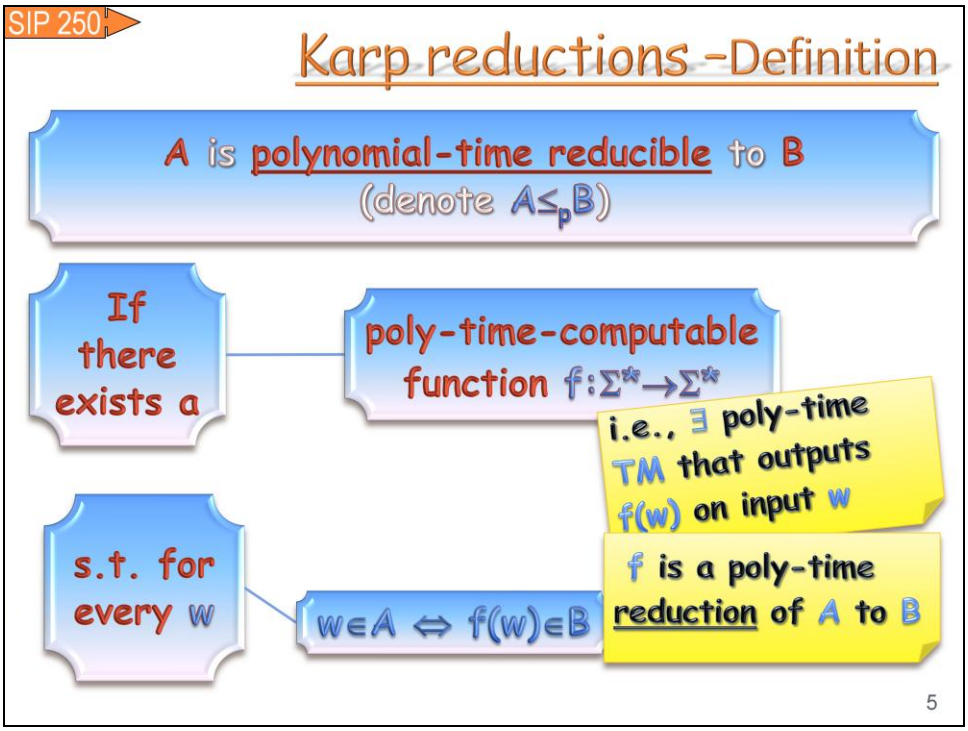
We'll now discuss in more details how to use reductions to bound problems' complexities.



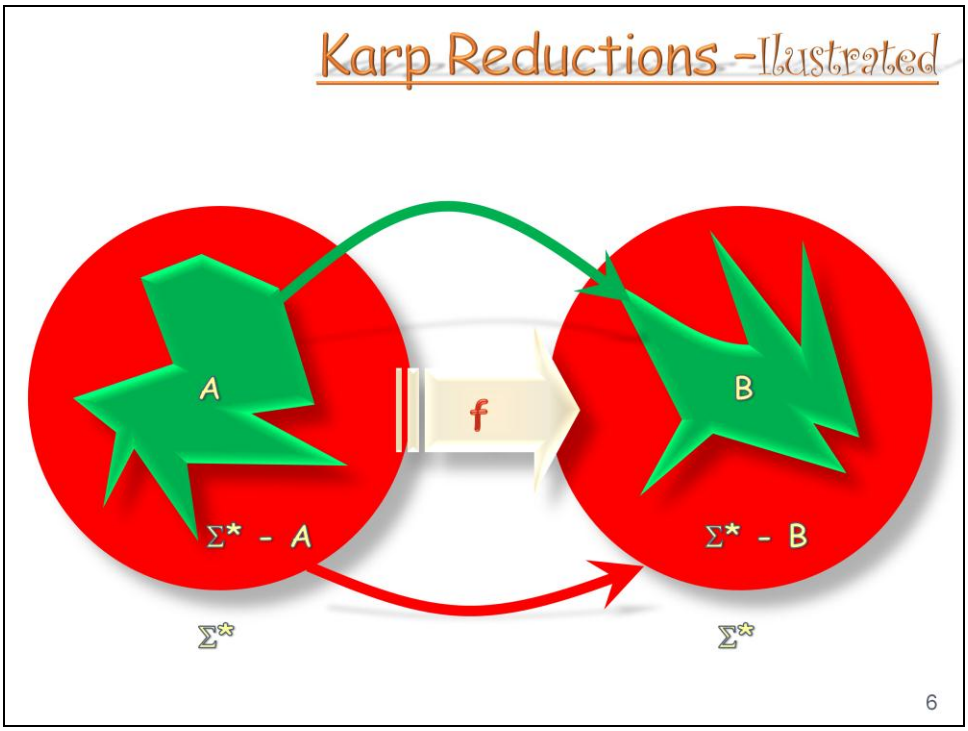
We'll discuss Karp reductions, discuss closeness of classes under reductions, and mention also the more general type of reductions.



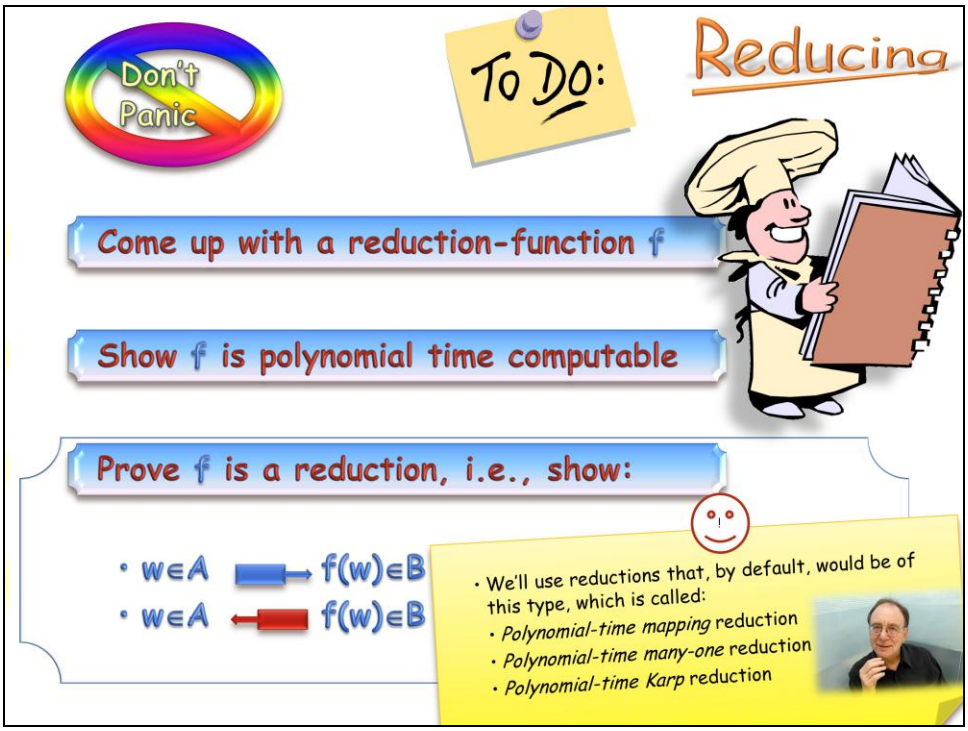
Recall the general type of reductions discussed earlier, from a problem A to a problem B, required us to show a procedure for A, which calls on a procedure for B, and so that assuming an efficient procedure for B, the procedure for A is also efficient.



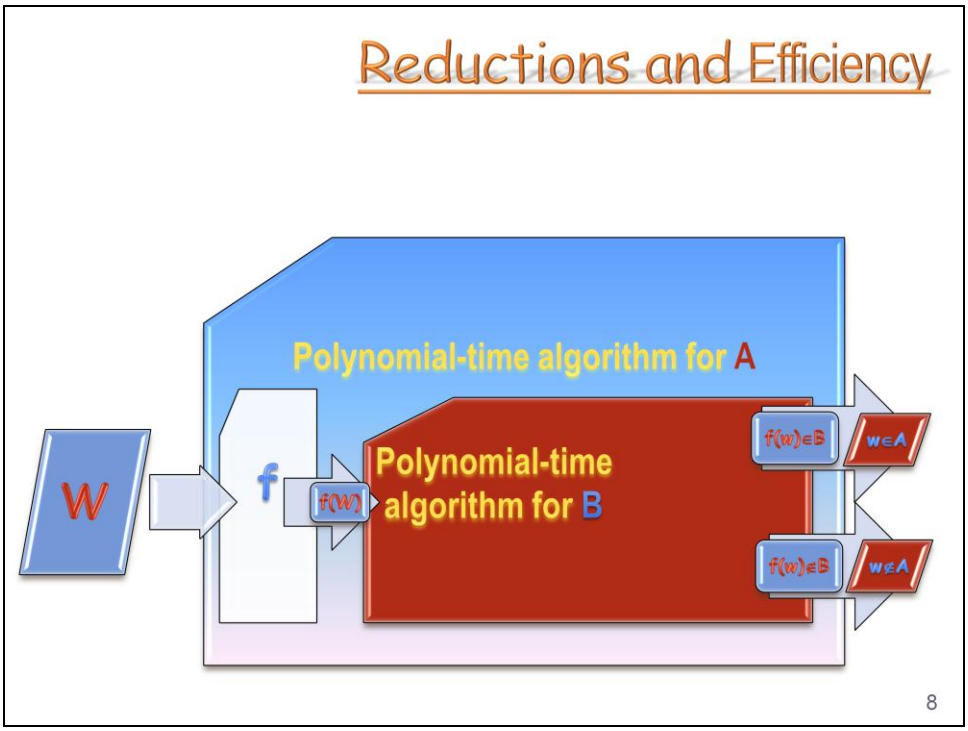
Let us now define a special type of reductions, referred to as Karp reductions. In this type of reduction, one constructs an efficient reduction-function, which translates an instance of the problem A to an instance of the problem B, while maintaining the two outcomes are the same.



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.



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 by default will be of that type.



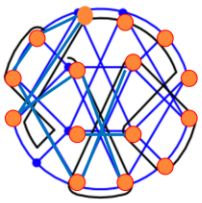
Let us now make sure that such a reduction implies that an efficient procedure for B entails an efficient procedure for A: on input W we apply the reduction-function, then apply B on its output, and simply return the outcome of that application.

Hamiltonian Path Instance:

- A directed graph $G=(V,E)$

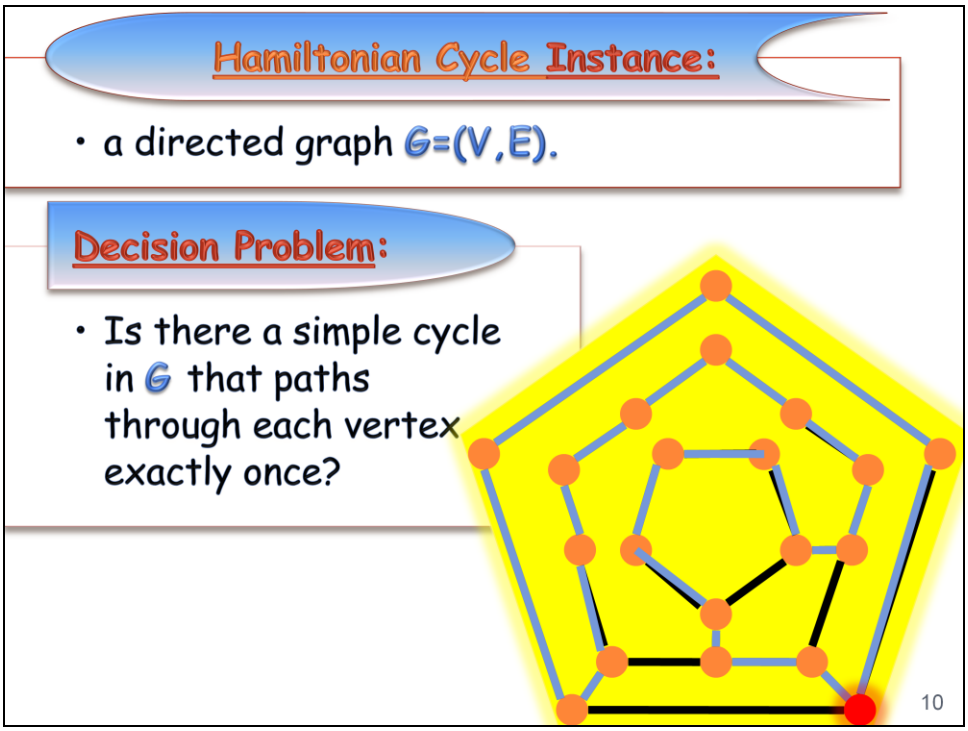
Decision Problem:

- Is there a path in G, which goes through every vertex exactly once?

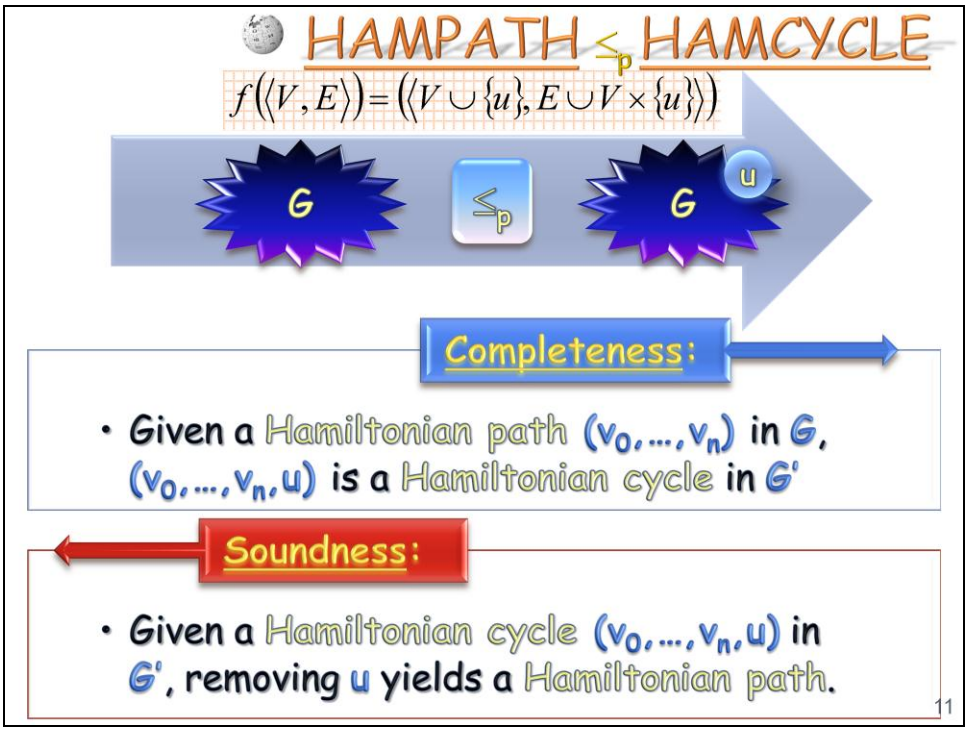


9

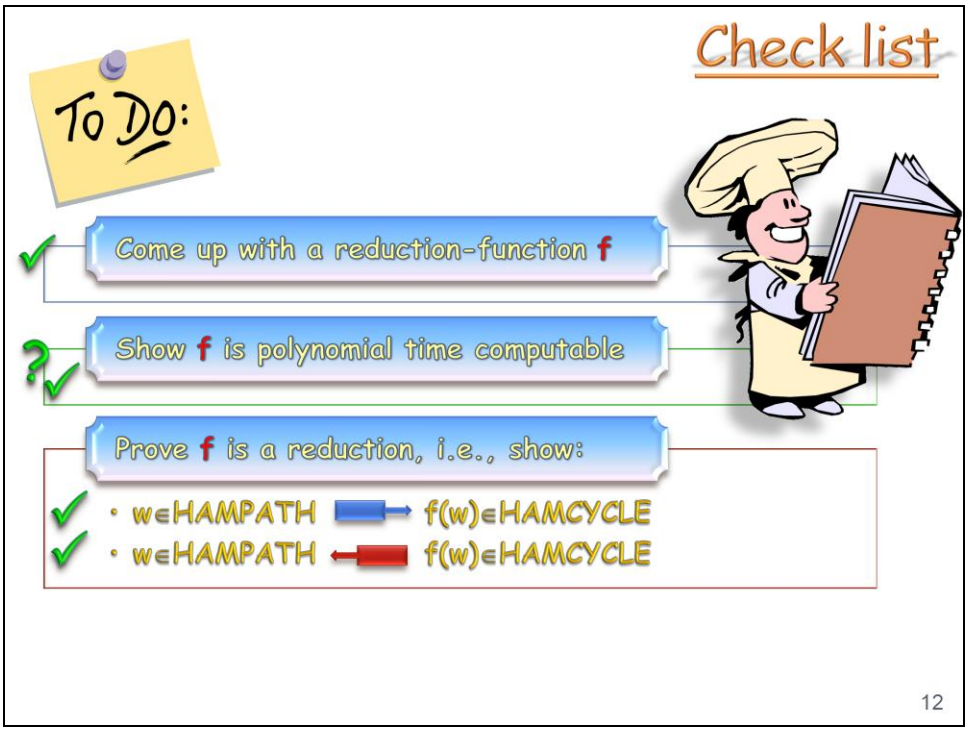
Formally define Hamiltonian path.



Formally define Hamiltonian cycle.



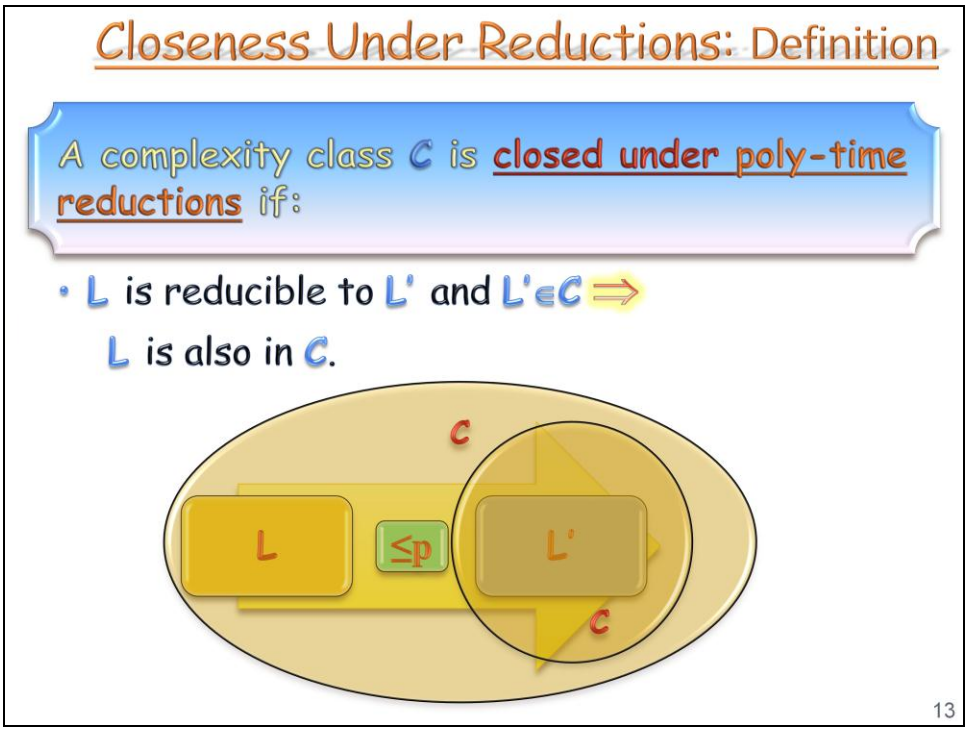
Let us now revisit the reduction, from Hamiltonian-path to Hamiltonian-cycle, previously described. We simply add to the graph an extra vertex, which is adjacent to all other vertexes. The completeness proof as well as a soundless proof are easy.



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.



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

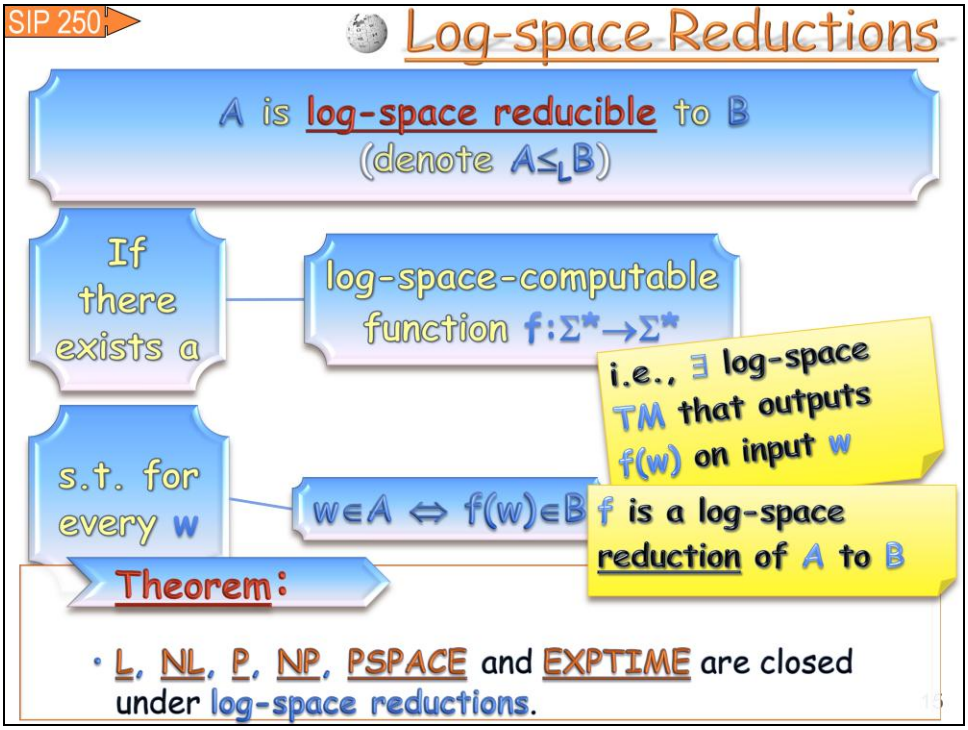
Proof:

- Do it yourself !!

14

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

Prove it!

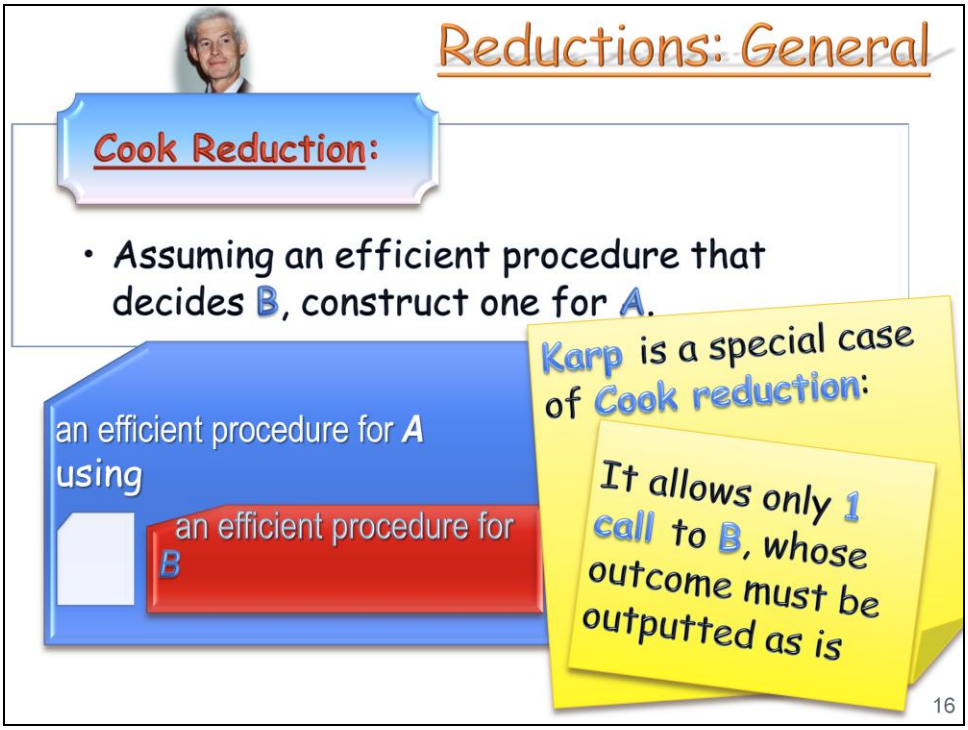


We can consider an even more efficient type of reductions, namely, those that can be carried out using only logarithmic size memory.

Can you think of a reduction that follows these guidelines?

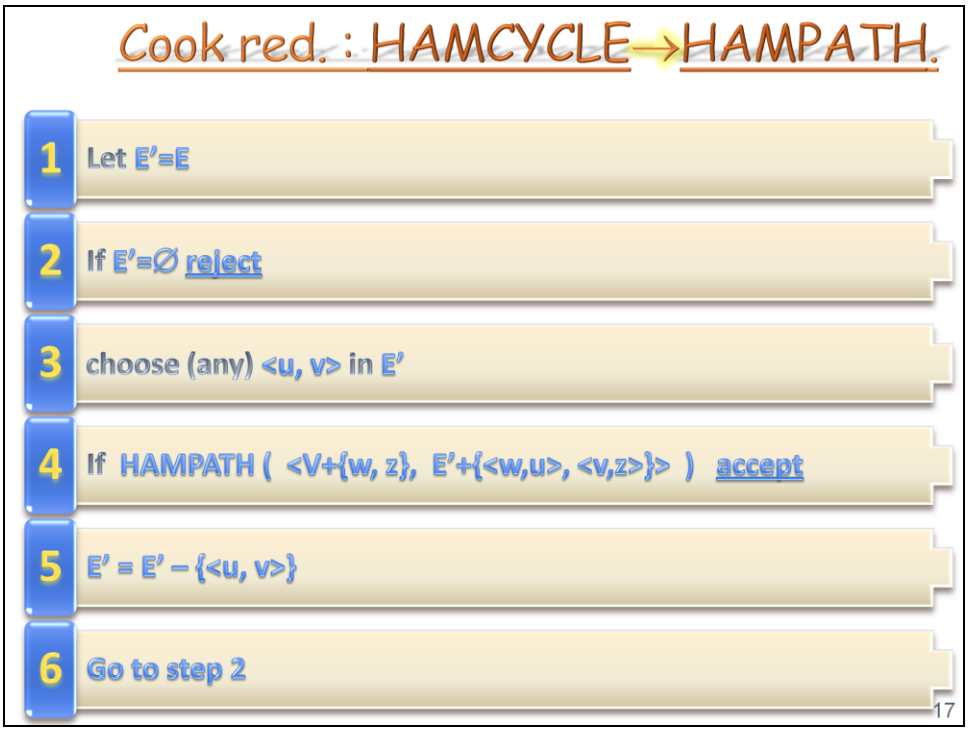
Can you show that even more classes are closed under such reductions?

Is it clear that these reductions do what we expect them to do? How does such a reduction output its outcome?

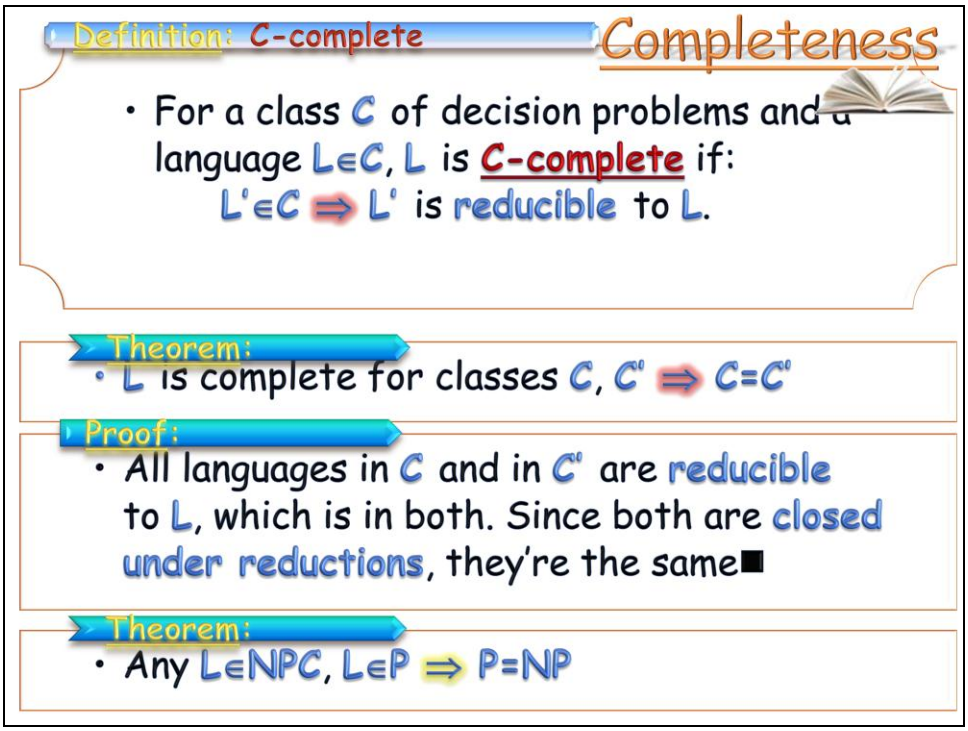


Karp reduction is a special case of the general, Cook reduction. It insists that the procedure for B is called only once, and that the outcome is simply returned as is.

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!



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



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 that problem.

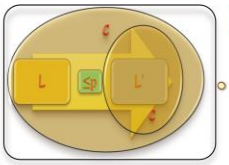
Such a problem then becomes a representative of that class, in particular, if the problem is complete to more than one class, those classes must be the same.

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:

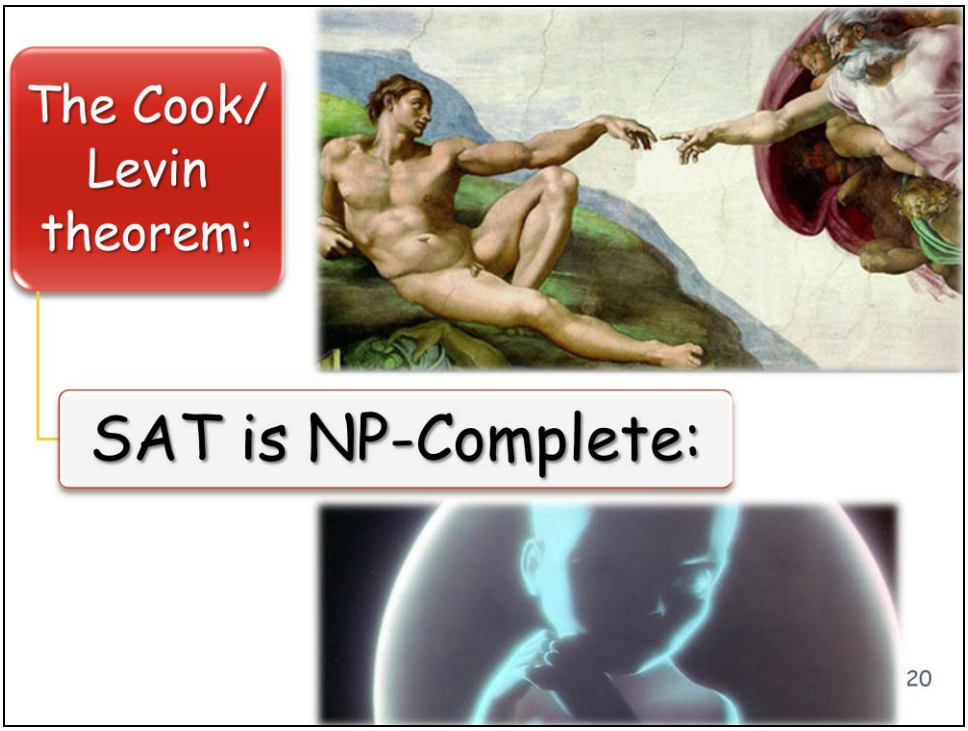
- "**completeness**"

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

Discussed a way to show:

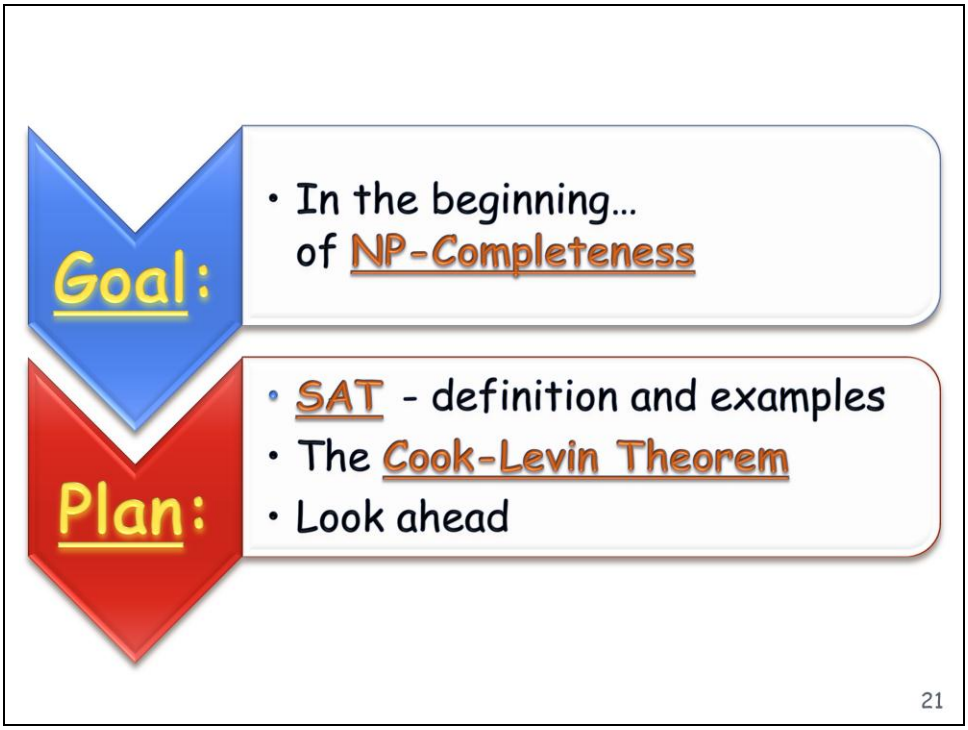
equality between **complexity classes**

19

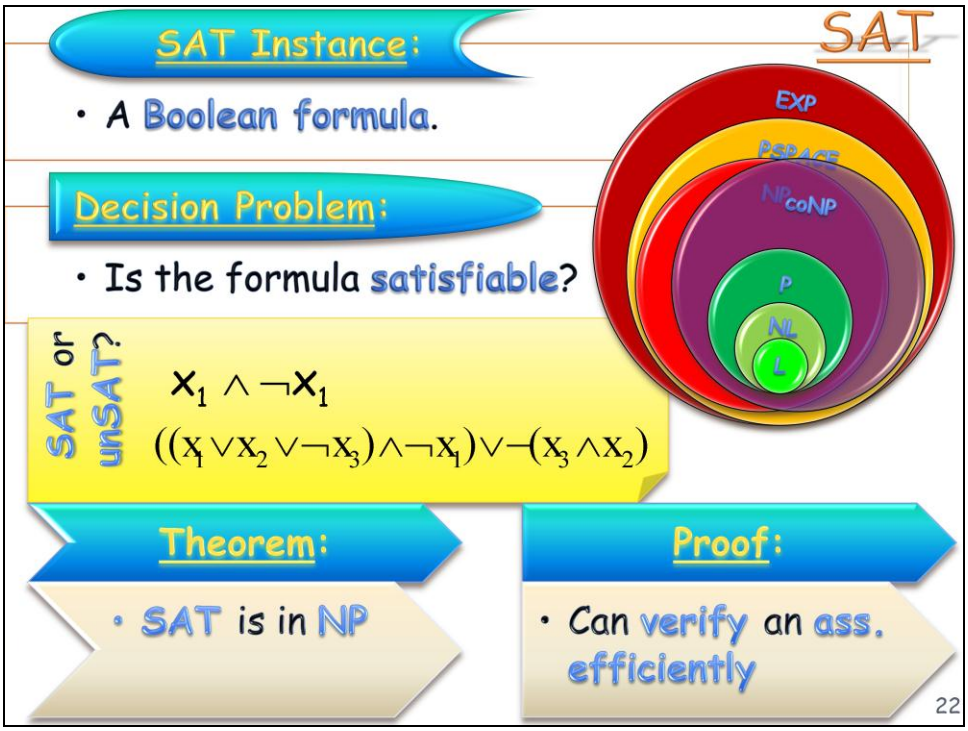


We're now going to prove one of the most basic Theorems of computer science --- proved by S. Cook and independently by L. Levin.

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

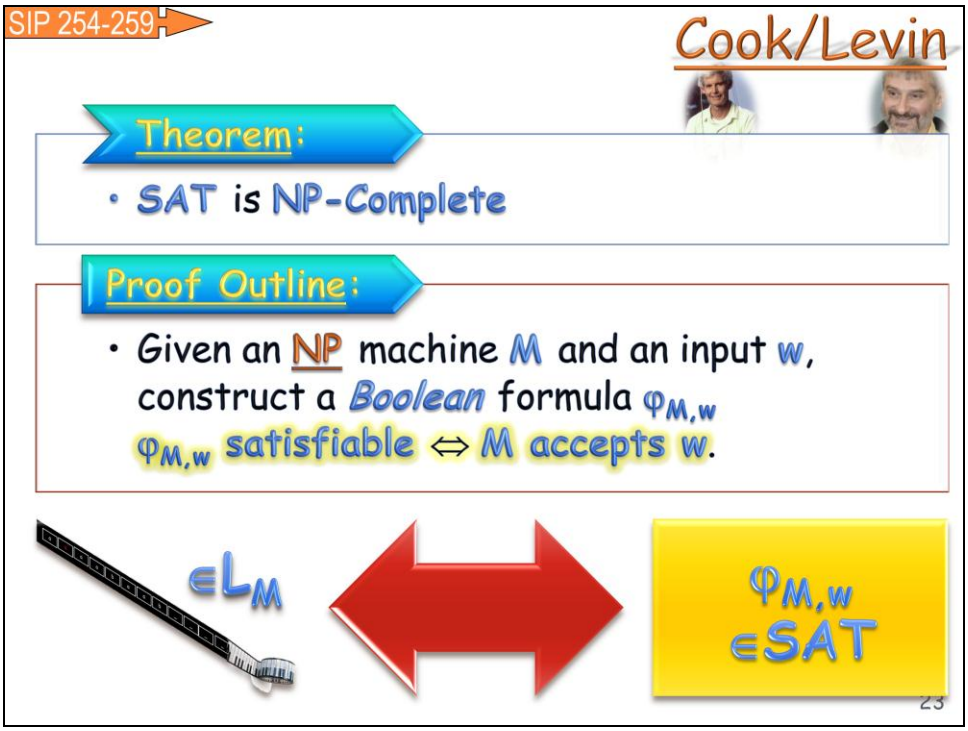


We'll define the SAT problem, and then proceed to prove that it is NP-complete.

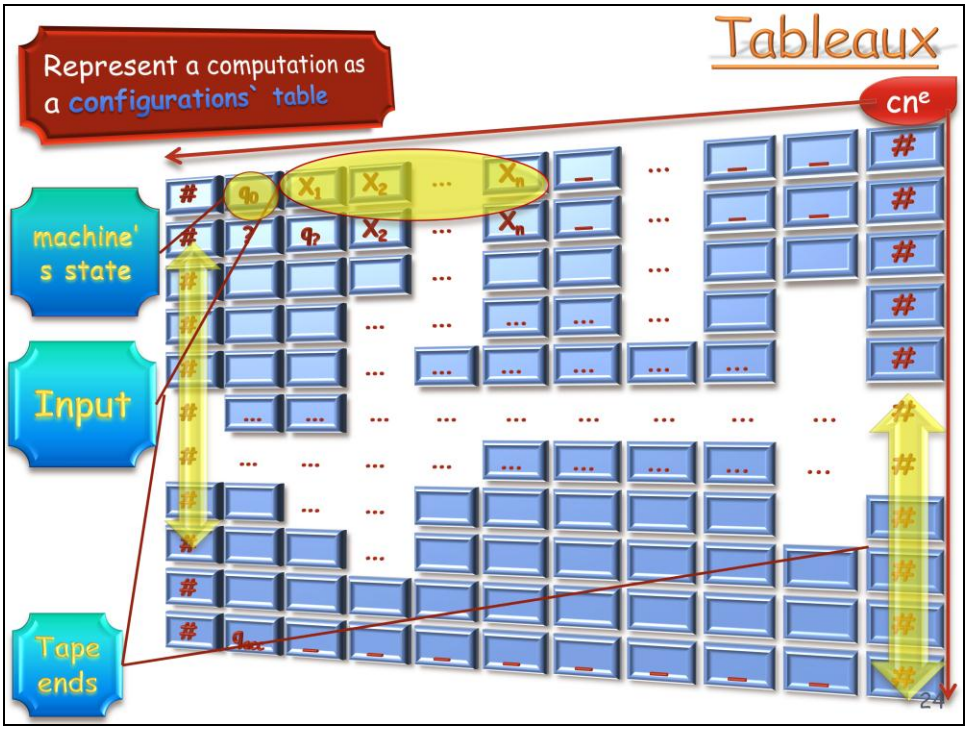


A SAT formula is a Boolean formula over Boolean variables.

The decision problem corresponding to it is whether there exists an assignment to the variables that causes the formula to evaluate to true.
SAT is clearly in NP.



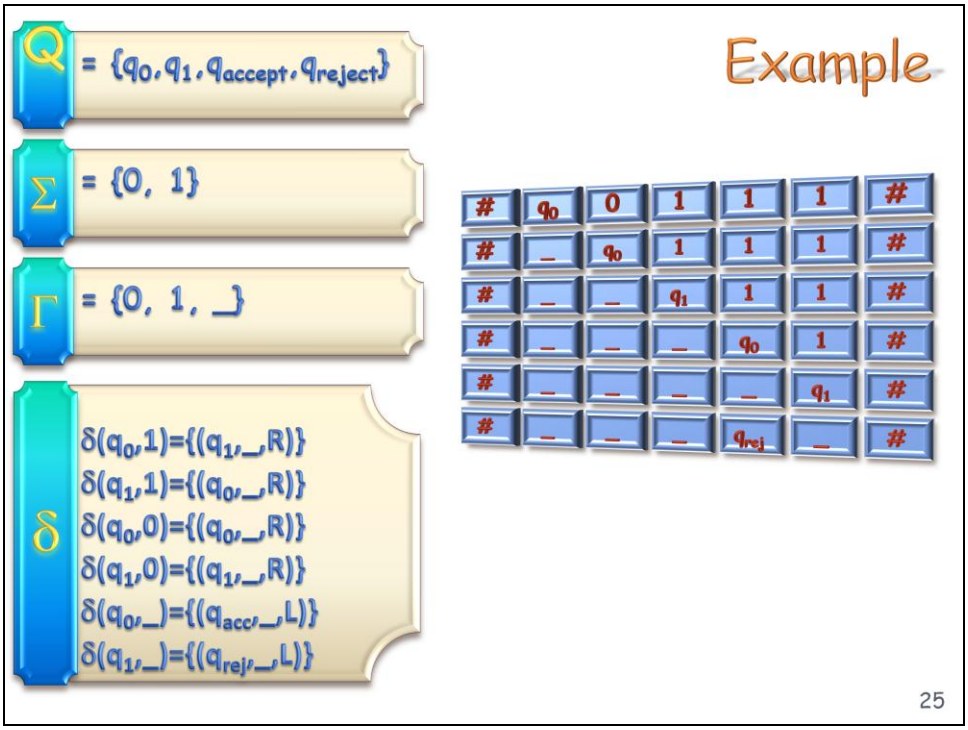
SAT is, moreover, NP hard, which is a much more fundamental statement. It being both in NP and NP hard, makes it NP-complete. The proof proceeds by, given a TM M and any input string W, constructing a SAT formula which is satisfied if and only if the TM M accepts the string W.



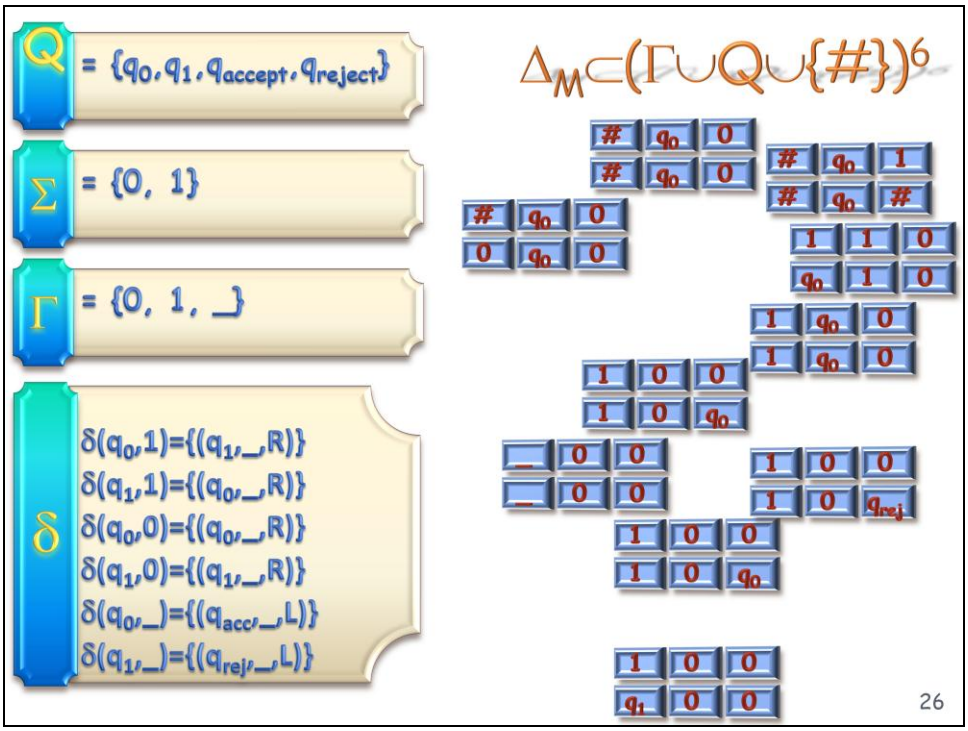
A computation of a (non deterministic) TM can be described in a table, where the i'th row corresponds to the configuration of the machine after i steps.

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.



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.

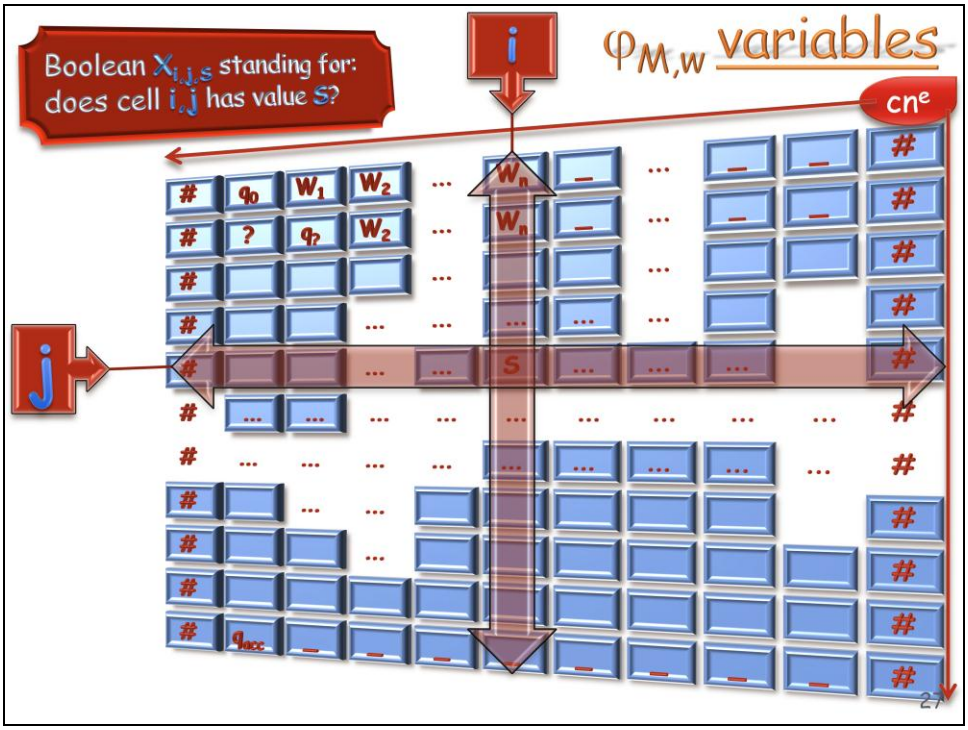


Let us concentrate for a moment on a 3 by 2 window of the configurations' table.

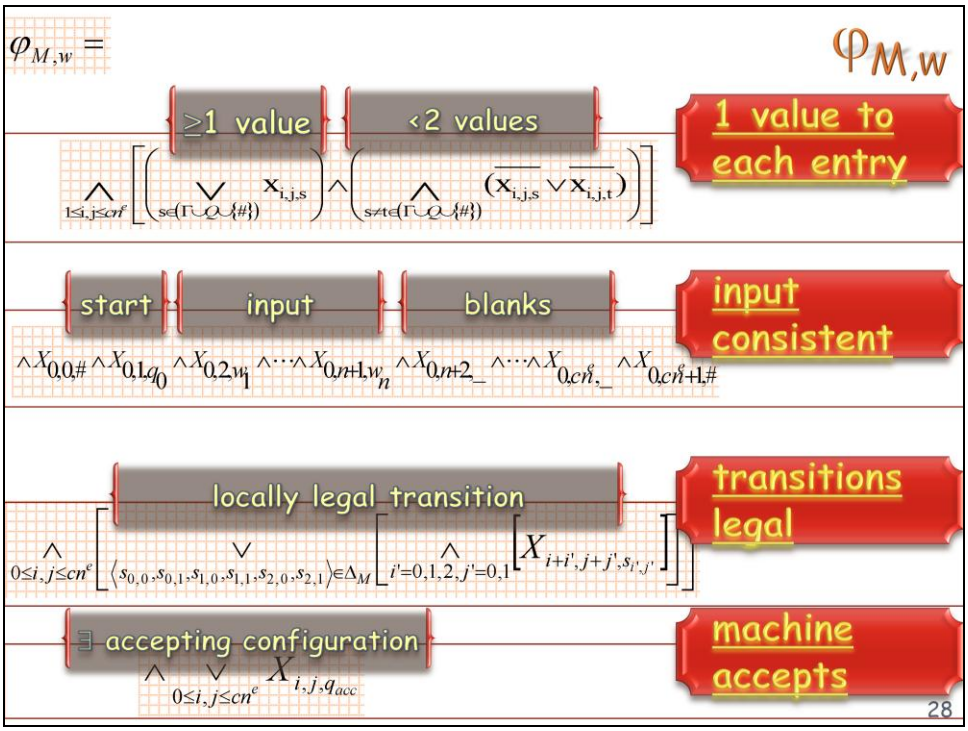
Which of the listed examples is legal?

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.



As to the Boolean variables of the formula constructed in the reduction: 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.

$$\varphi_{M,w} = \bigwedge_{1\le i,j\le cn^e}\left[\left(\bigvee_{s\in(\Gamma\cup Q\cup\{\#\})} x_{i,j,s}\right)\wedge\left(\bigwedge_{s\neq t\in(\Gamma\cup Q\cup\{\#\})}(\overline{x_{i,j,s}}\vee\overline{x_{i,j,t}})\right)\right]$$

$$\wedge X_{0,0,\#}\wedge X_{0,1,q_0}\wedge X_{0,2,w_1}\wedge\cdots\wedge X_{0,n+1,w_n}\wedge X_{0,n+2,\_}\wedge\cdots\wedge X_{0,cn^e,\_}\wedge X_{0,cn^e+1,\#}$$

$$\bigwedge_{0\le i,j\le cn^e}\left[\bigvee_{\langle s_{0,0},s_{0,1},s_{1,0},s_{1,1},s_{2,0},s_{2,1}\rangle\in\Delta_M}\left[\bigwedge_{i'=0,1,2,j'=0,1}\left[X_{i+i',j+j',s_{i',j'}}\right]\right]\right]$$

$$\wedge\bigvee_{0\le i,j\le cn^e} X_{i,j,q_{acc}}$$

Q.E.D.

Claim:

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

Corollary:

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

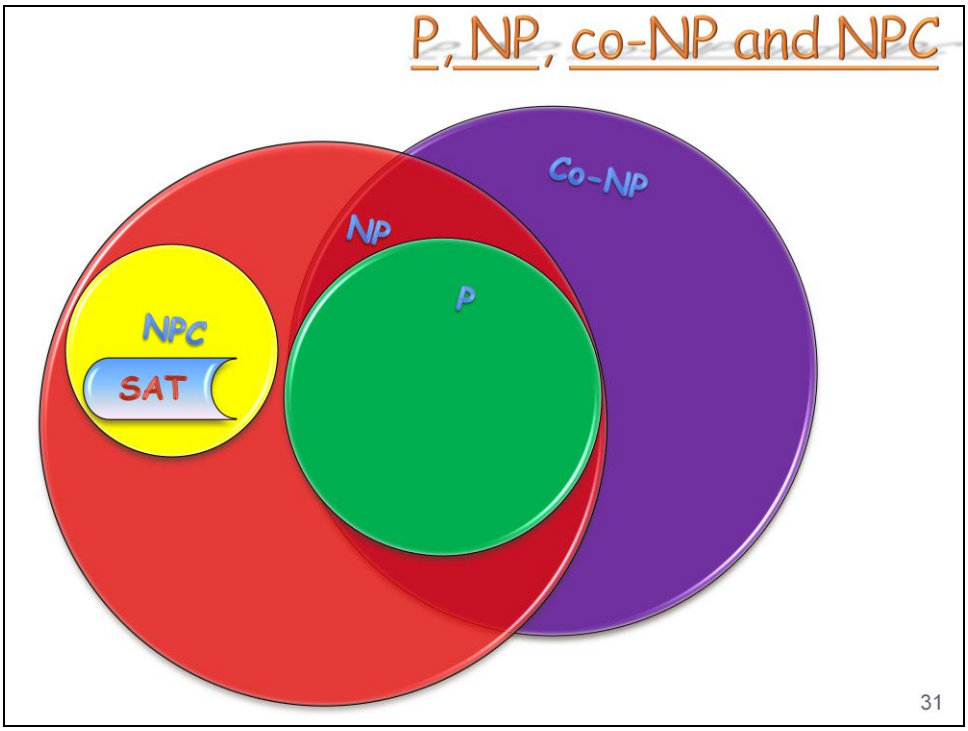
Claim:

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

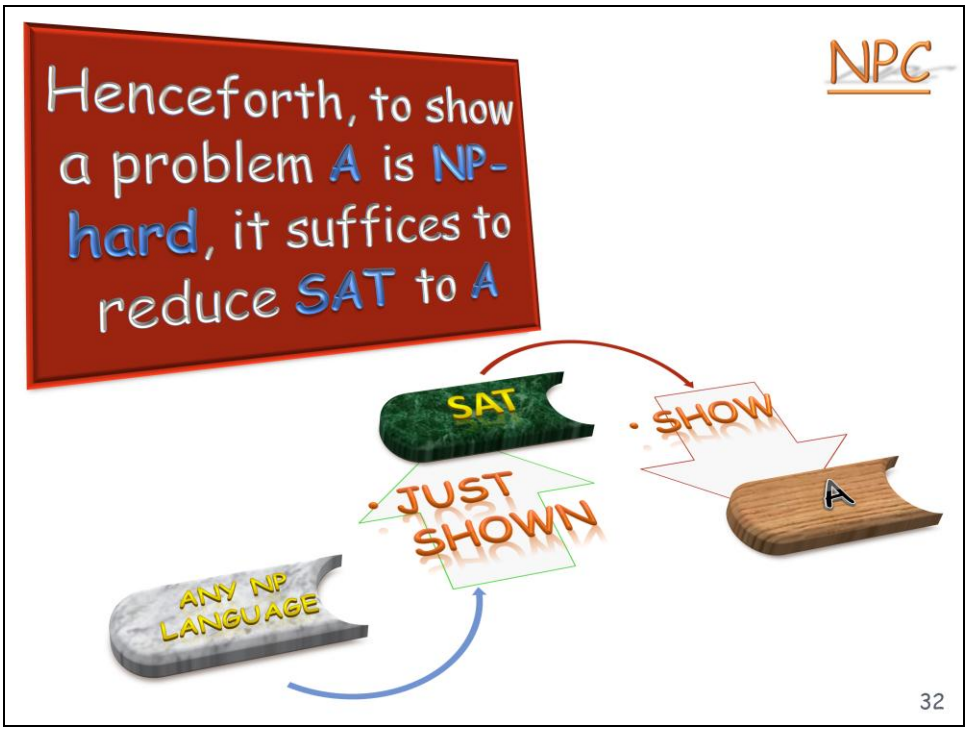
29

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.

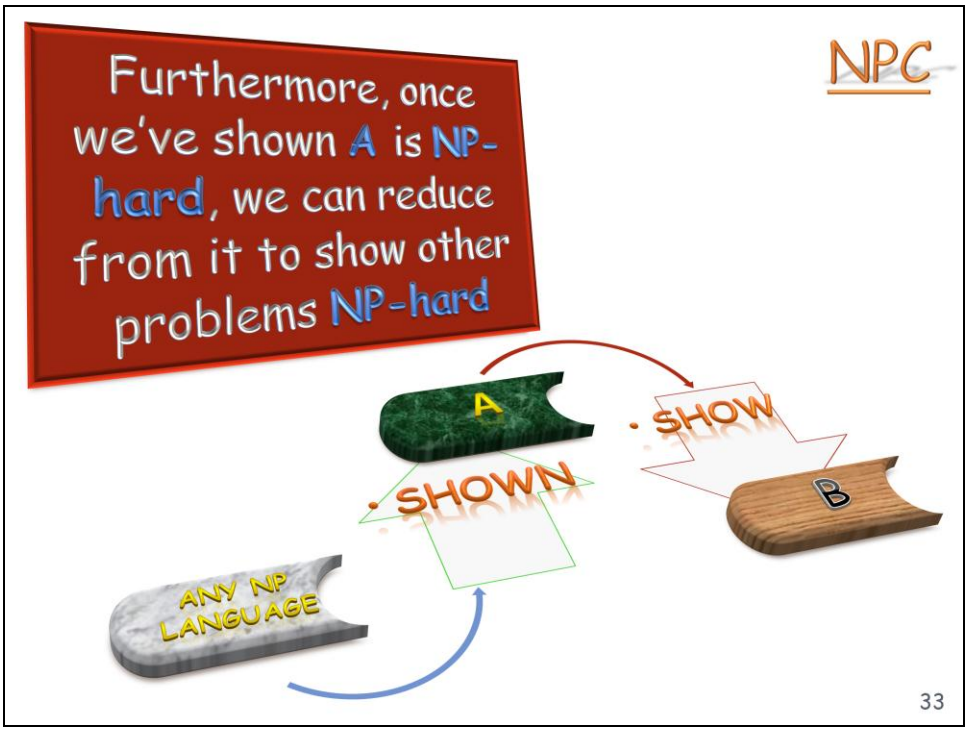
We have just shown that any language in NP can be efficiently reduced to SAT. This implies SAT is NP hard. Since we have already shown SAT is in NP, we conclude that SAT is NP-complete.



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.

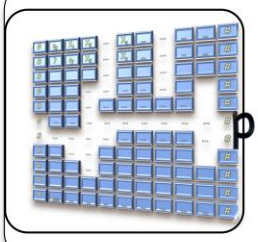


Now that we have shown SAT is NP hard, to show other problems are NP hard, we may reduce SAT to them. We don't have to repeat this proof.



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

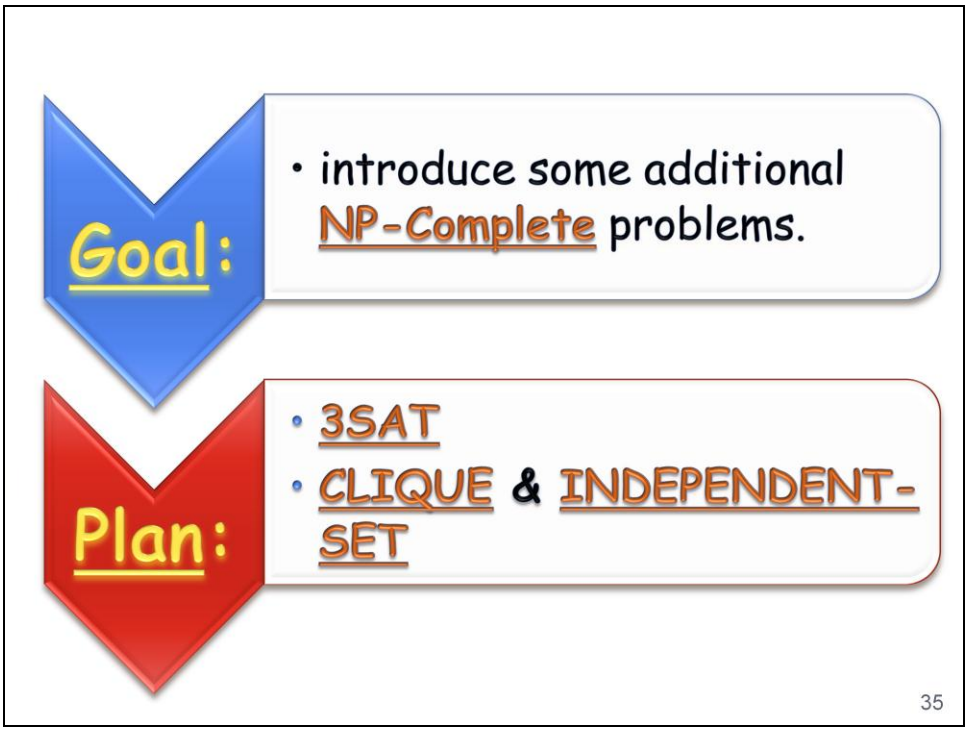
Summary



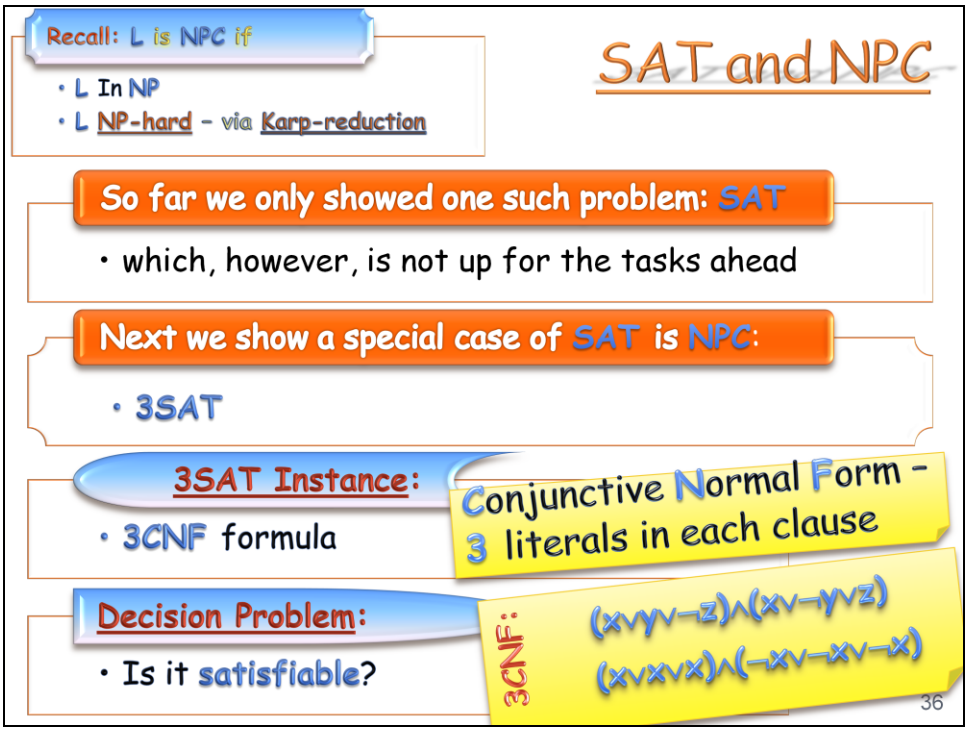
proved **SAT** is **NP-Complete**

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

34



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.

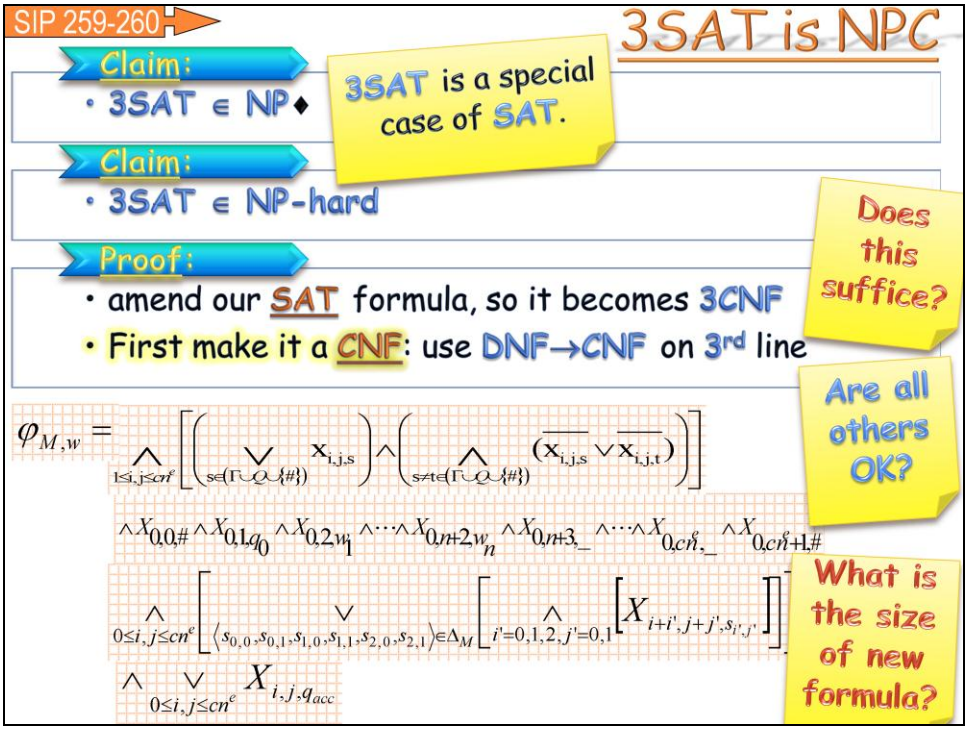


Recall that a language is NP-complete if it is in NP and is also NP-hard.

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 clause.

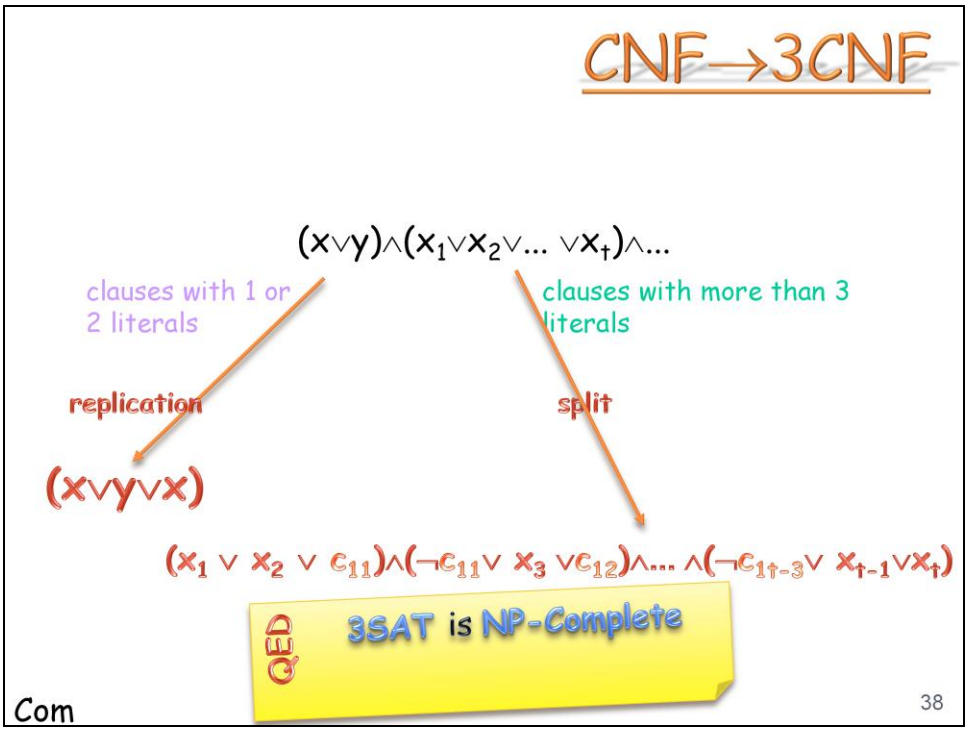
The language 3SAT consists of all such formulas that have an assignment that satisfies them.



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).

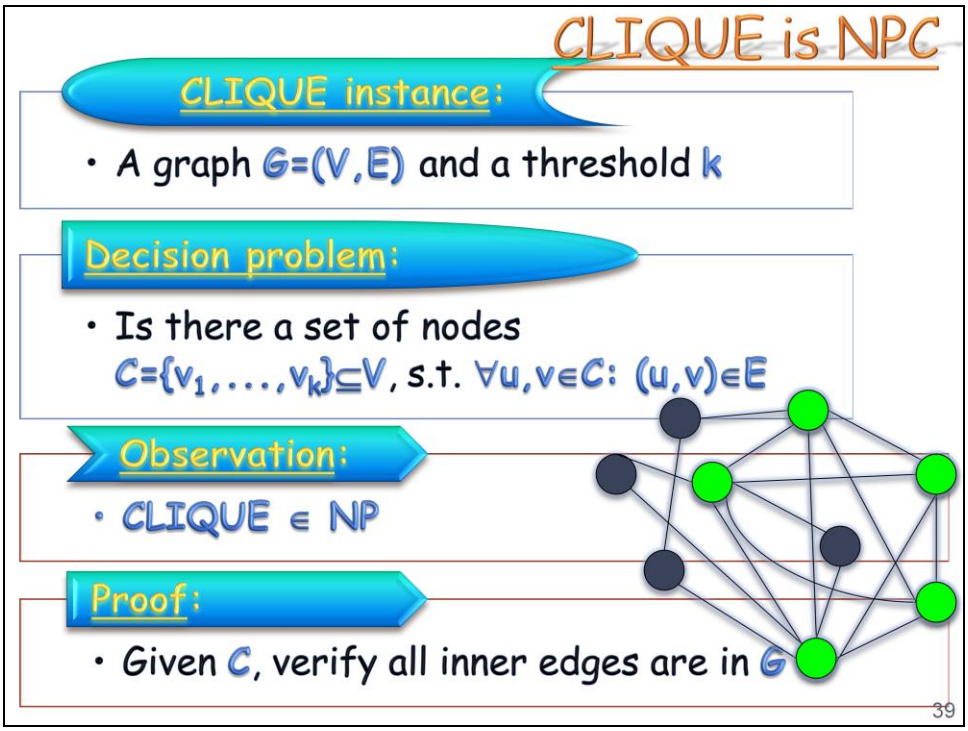
To prove that 3SAT is NP-hard, it suffices to efficiently alter the SAT formula we had obtained previously into the proper form.

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.



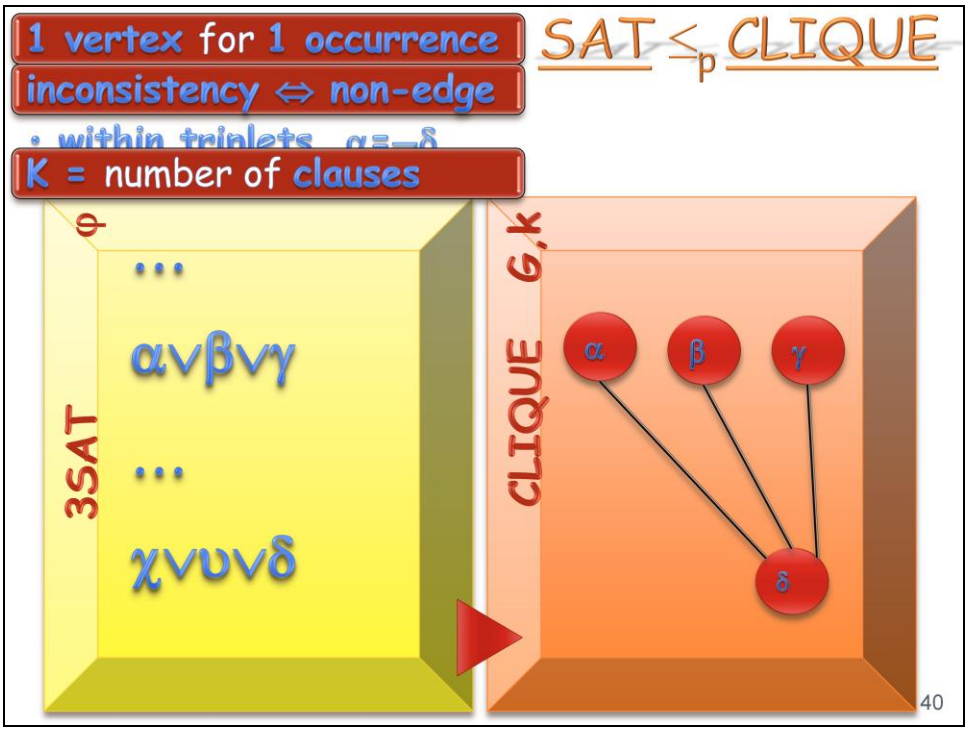
For the purpose of translating general CNF to 3CNF, one needs to replace each clause with a simple set of 3-wide clauses utilizing some extra variables, while maintaining satisfiability of the original clause.

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



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.



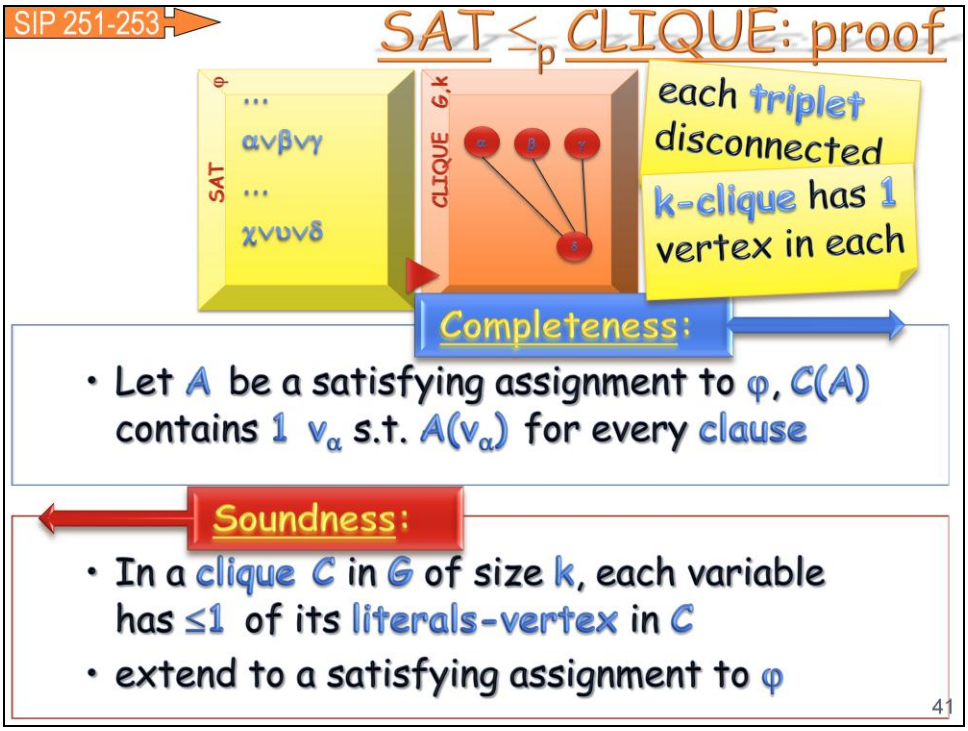
To show the CLIQUE problem is NP-hard, we'll reduce 3SAT to it.

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.

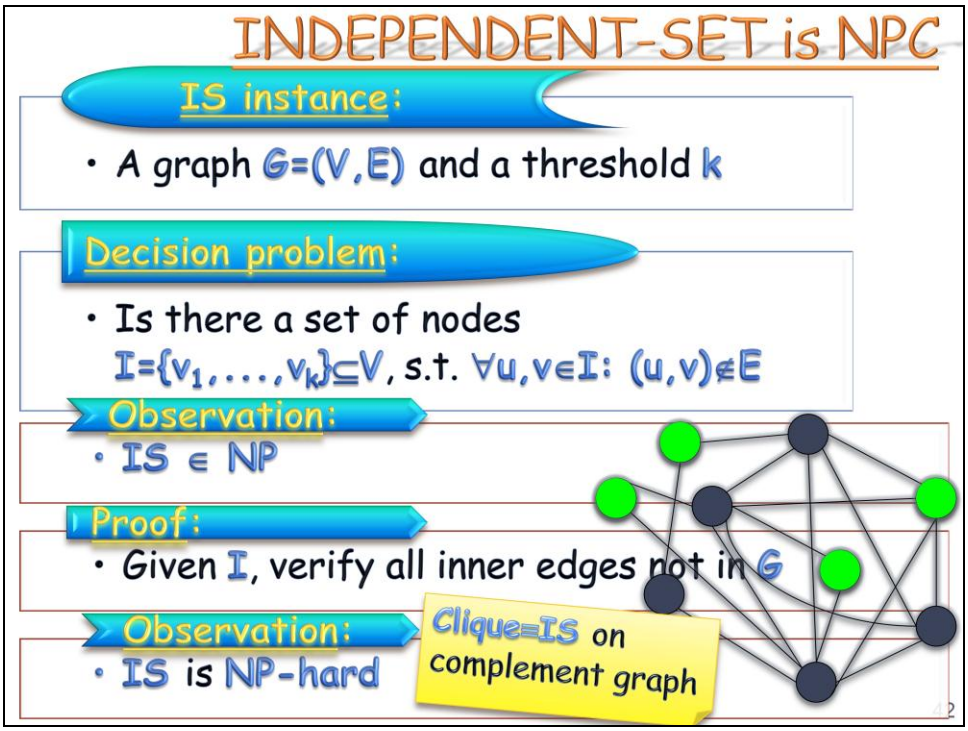


Since there are no edges between a triplet, a CLIQUE of size k must comprise one vertex in each triplet.

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.



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.

Reductions

Polynomial Time Reductions

Completeness

Hamiltonian Path

Log Space Reductions

Completeness

Complexity Classes

NP

co-NP

P

L

NL

EXPTIME

PSPACE

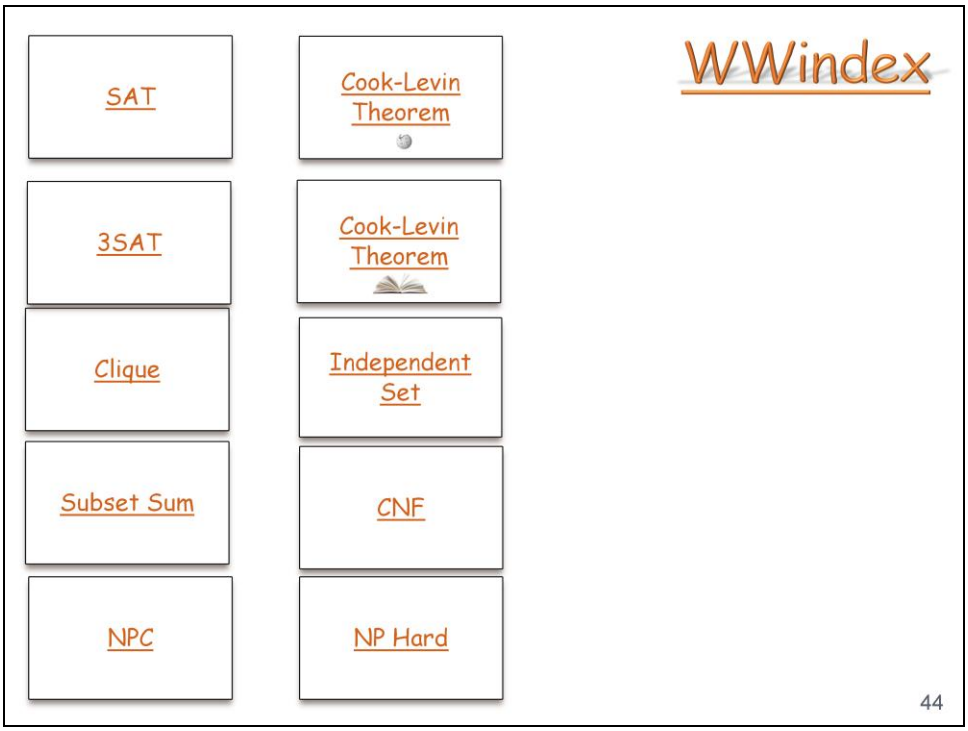
WWindex


Hamilton, William Rowan


Karp, Richard


Cook, Stephen Arthur


Levin, Leonid


WWindex
SAT
Cook-Levin Theorem
3SAT
Cook-Levin Theorem
Clique
Independent Set
Subset Sum
CNF
NPC
NP Hard
44