ICS 311 #24: NP-Completeness

Outline

1. Don't Let This Happen to You
2. Problem Classes
3. P and NP Classes
4. Encoding Problems and Polynomial Time Verification
5. NP Completeness
6. Constructing NPC
7. NP Complete Problems

Readings and Screencasts

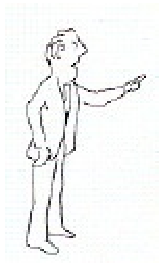
- CLRS 3rd Ed. Chapter 34 (read for the main concepts).
- Screencasts 24 A Introduction to Concepts, 24 B An Initial NPC Problem, 24 C More NPC Problems.

Don't Let This Happen to You

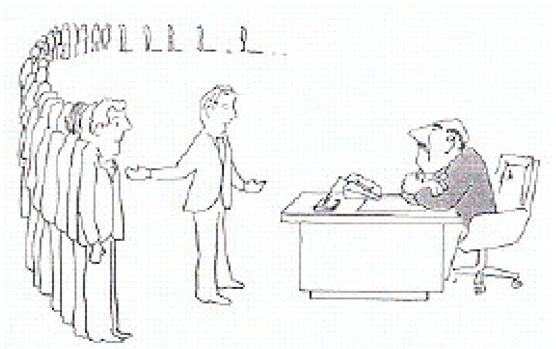
Suppose you worked in industry, and your boss calls you to his office and says they want to enter the emerging "iThingy" market. He wants to offer customized iThingies, and it's important to have an algorithm that, given a specification, can efficiently construct a design that meets the maximum number of requirements in the lowest cost way.

You try for weeks but just can't seem to find an efficient solution. Every solution that you come up with amounts to trying all combinations of components, in exponential time. More efficient algorithms fail to find optimal solutions.

You don't want to have to go your boss and say "I'm sorry I can't find an efficient solution. I guess I'm just too dumb."



You would like to be able to confidently stride into his office and say "I can't find an efficient solution because no such solution exists!" Unfortunately, proving that the problem is inherently intractable is also very hard: in fact, no one has succeeded in doing so.



Today we introduce a class of problems that no one has been able to prove is intractable, but nor has anyone been able to find an efficient solution to any of them.

If you can show that the iThingy configuration problem is one of these problems, you can go to your boss and say, "I can't find an efficient solution, but neither can all these smart and famous computer scientists!"

(In Topic 25 we'll cover what you would say next. Thanks to Garey & Johnson (1979) for the story and images.)

Problem Classes

For most of this semester, we abstracted away from the study of particular ***implementations*** to study the computational complexity of ***algorithms*** independently of their implementation. More precisely, we made universally quantified statements to the effect that all possible implementations of an algorithm would exhibit certain asymptotic growth within constant factors of each other.

Now we abstract further, to study the computational complexity of ***problems***, independently of the algorithms used to solve them. We will be trying to make universally quantified statements about the computational complexity of all possible algorithms for a problem. (We won't always succeed in doing this.)

For example, when we showed that any comparison-based sorting algorithm is O(n lg n) in Topic 10, we were making a claim about a class of algorithms for a problem.

Tractability and Decidability

Broadly, we are concerned with three classes of problems:

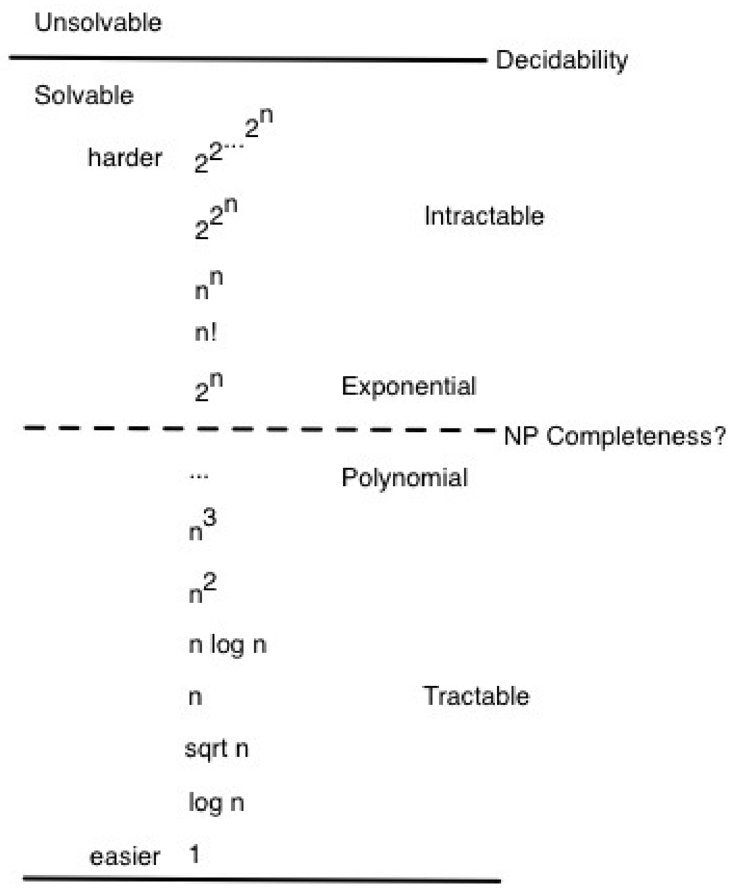
- **Tractable problems**, generally considered to be those solvable in polynomial time (most of the problems considered so far in this course).
  - *Examples:* Any problem studied this semester for which we have solutions bounded by n^k for a fixed constant k.
  - Some solutions are not strictly polynomial but are bounded above by a polynomial (e.g., n lg n for sorting is bounded by n^2), so sorting is considered polynomial.

- **Intractable problems** are those solvable in super-polynomial but not polynomial time. Today we will be concerned with the question of which problems are in this class.
  - *Examples:* Enumerate all binary strings of length n; Compute the power set (set of all sets) of n items. The solutions consist of 2^n binary strings, or 2^n sets, respectively, so even if the solution were available instantly, exponential time would be required just to output the result!
  - *Unknown:* Integer linear programming, finding longest simple paths in a graph, finding the optimal way to schedule tasks on factory machines under scheduling and deadline constraints, determining whether certain boolean formulas can be simultaneously satisfied, optimizing circuit layouts under certain constraints, finding the cheapest way to visit all of a set of cities without repeating any and returning to your starting city, finding the optimal partition of graphs under a modularity metric, and many many more ...

- **Unsolvable problems** for which no algorithm can be given guaranteed to solve all instances of the problem. These problems are also known as ***Undecidable***.
  - Example: the ***halting problem*** (given a description of a program and arbitrary input, decide whether the program will halt on that input), demonstrated by Alan Turing.

Hierarchy of Problem Classes

The hierarchy of problem classes was illustrated at the beginning of the semester with this diagram:



We have spent most of our time on problems in the lower half of this diagram. Now we consider whether there are problems that are ***intrisically*** in the upper half.

P and NP Classes

P denotes the class of problems solvable in polynomial time, such as most of the problems we have considered this semester.

Nondeterministic Polynomial

NP denotes the class of problems for which solutions are ***verifiable*** in polynomial time: given a description of the problem x and a "certificate" y describing a solution (or providing enough information to show that a solution exists) that is polynomial in the size of x, we can check the solution (or verify that a solution exists) in polynomial time as a function of x and y.

Problems in NP are **decision problems**: we answer "yes" or "no" to whether the certificate is a solution. The polynomial size requirement on y rules out problems that take exponential time just to output their results, such as the enumeration of all binary strings of length n. (One could check the solution in time polynomial in y, but y would be exponential in x, so overall the problem is not tractable.)

These problems are called "nondeterministic polynomial" because one way of defining them is to suppose we have a **nondeterministic machine** that, whenever faced with a choice, can guess the correct alternative, producing a solution in polynomial time based on this guess. (Amazingly, no one has been able to prove that such a fanciful machine would help!)

Another way to think of this is that the machine can copy itself at each choice point, essentially being multithreaded on an infinite number of processors and returning the solution as soon as the solution is found down a span of polynomial depth (see Topic on Multithreading).

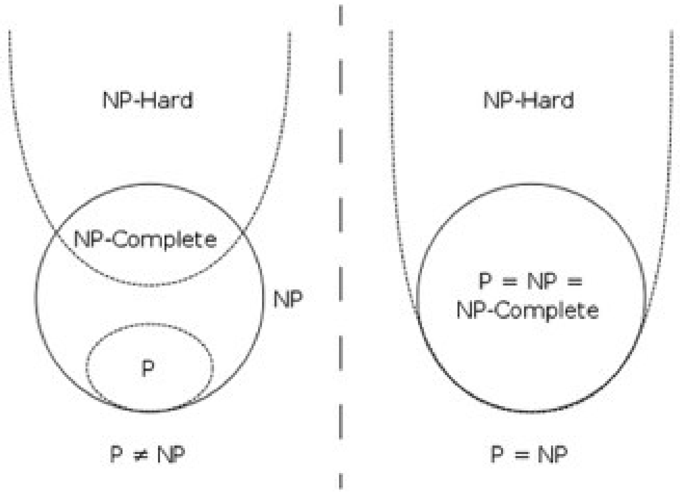
(Your text does not take either of these approaches, preferring the "certificate" definition. We elaborate on this later, but not in depth.)

Relationship of P to NP

Clearly, P is a subset of NP.

The million dollar question (literally, see the Clay Mathematics Institute Millenium Prize for the P vs NP Problem), is whether P is a strict subset of NP (i.e., whether there are problems in NP that are not in P, so are inherently exponential).

NP Completeness



Theorists *have* been able to show that there are problems that are just as hard as any problem in NP, in that if any of these **NP-Hard** problems are solved then any problem in NP can be solved by reduction to (translating them into instances of) the NP-Hard problem.

Those NP-Hard problems that are also in NP are said to be **NP Complete**, denoted **NPC**, in that if *any* NPC problem can be solved in polynomial time then *all* problems in NP can also be solved in polynomial time.

The study of NP Completeness is important: the *most cited reference in all of Computer Science* is Garey & Johnson's (1979) book Computers and Intractability: A Guide to the Theory of NP-Completeness. (Your textbook is the second most cited reference in Computer Science!)

In 1979 Garey & Johnson wrote, "The question of whether or not the NP-complete problems are intractable is now considered to be one of the foremost open questions of contemporary mathematics and computer science."

Over 30 years later, in spite of a million dollar prize offer and intensive study by many of the best minds in computer science, this is still true: No one has been able to either

- Prove that there are problems in NP that cannot be solved in polynomial time (which would mean P ≠ NP), or
- Find a polynomial time solution for a single NP Complete Problem (which would mean P=NP).

Although either alternative is possible, most computer scientists believe that P ≠ NP.

Discriminating P and NP problems

Problems that look very similar to each other may belong to different complexity classes (if P ≠ NP), for example:

- **Linear programming** can be solved in polynomial time (the simplex algorithm), but if we require that the assigned values be integers we have the **integer linear programming problem**, which is

NP-Hard.

- **Shortest paths** in a graph can be found in O(VE) or better, but the problem of finding **longest simple paths** is NP-Hard (deciding whether there is a path longer than length k is NP-Complete).

- An **Euler tour** that traverses each edge of a graph once can be found in O(E) time, but finding a **Hamiltonian cycle** that traverses each vertex exactly once via a simple cycle is NP-Hard.

- We can determine whether a boolean formula in **2-Conjunctive Normal Form** is satisfiable in polynomial time, but doing the same for a formula in **3-Conjunctive Normal Form** is NP-Complete.

Clearly, it is important that we be able to recognize such problems when we encounter them and not share the fate of the iThingy algorithm designer. (In the next topic we'll discuss approximation algorithms for dealing with them in a practical way.)

Encoding Problems and Polynomial Time Verification

This section discusses some concepts used in the study of complexity classes. We are not delving into formal proofs such as those provided in the text. The main purpose of this section is to give you familiarity with the terminology and to show why the remaining notes talk about "languages".

Abstract Problems

An **abstract problem** Q is a binary relation mapping problem **instances** I to problem **solutions** S.

NP Completeness is concerned with **decision problems**: those having a yes/no answer, or in which Q maps I to $\{0, 1\}$.

Many problems are **optimization problems**, which require that some value be minimized (e.g., finding shortest paths) or maximized (e.g., finding longest paths).

We can easily convert an optimization problem to a decision problem by asking: is there a solution that has value lower than (or higher than) a given value?

Encodings and Concrete Problems

To specify whether an abstract problem is solvable in polynomial time, we need to carefully specify the size of the input.

An **encoding** of a problem maps problem instances to binary strings. We consider only "reasonable" encodings

- For example, we do not represent numbers in unary format. If we did, the input to the problem of enumerating all binary strings of length n would itself be of length 2^n, so an "O(n)" solution would be possible!

- A "reasonable" encoding an integer is polynomial related to its representation, and a set of objects polynomially related to its encoding. Standard codings such as ASCII meet this requirement.

Once problems and their solutions are encoded, we are dealing with **concrete problem instances**. Then a problem Q can be thought of as a function $Q : \{0, 1\}^* \rightarrow \{0, 1\}^*$, or if it is a decision problem, $Q : \{0, 1\}^*$

$\rightarrow \{0, 1\}$. ($Q(x) = 0$ for any string $x \in \Sigma^*$ that is not a legal encoding.)

Accepting and Deciding Formal Langauges

By casting computational problems as decision problems, theorists can use concepts from formal language theory in their proofs. We are not doing these proofs but you should be aware of the basic concepts and distinctions:

A **language** L over an alphabet Σ is a set of strings made up of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, ...\}$ is the language of binary representations of prime numbers.

The language that contains all strings over Σ is denoted Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$, where ε denotes an empty string.

An algorithm A **accepts** a string $x \in \{0, 1\}^*$ if given x the output of A is 1.

The **language L accepted by A** is the set of strings $L = \{x \in \{0, 1\}^* : A(x) = 1\}$.

But A need not halt on strings not in L. It could just never return an answer. (The existence of problems like the Halting Problem necessitate considering this possibility.)

A language is **decided** by an algorithm A if it accepts precisely those strings in L *and* rejects those not in L (i.e., A is guaranteed to halt with result 1 or 0).

A language is **accepted in polynomial time** by A if it is accepted by A in time $O(n^k)$ for every encoded input of length n and some constant k. Similarly, a language is **decided in polynomial time** by A if it is decided by A in time $O(n^k)$.

Polynomial Time Verification

A **complexity class** is a set of languages for which membership is determined by a complexity measure. (Presently we are interested in the running time required of any algorithm that decides L, but complexity classes can also be defined by other measures, such as space required.) For example, we can now define P more formally as:

$$\mathbf{P} = \{L \subseteq \{0, 1\}^* : \exists \text{ algorithm } A \text{ that decides } L \text{ in polynomial time}\}.$$

A **verification algorithm $A(x, y)$** takes two arguments: an **encoding x** of a problem and a **certificate y** for a solution. A returns 1 if the solution is valid for the problem. (A need not solve the problem; only verify the proposed solution.)

The **language verified** by a verification algorithm A is

$$L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$

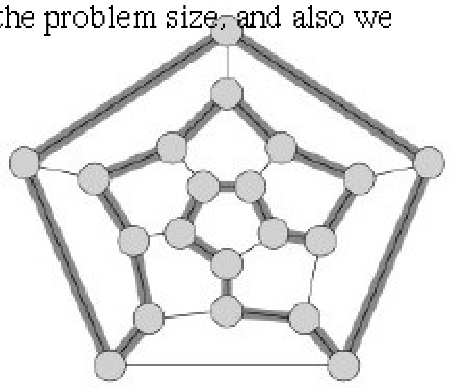
We can now define the complexity class **NP** as the class of languages that can be verified by a polynomial time algorithm, or formally:

$L \in$ NP iff $\exists$ polynomial time algorithm $A(x, y)$ and constant c such that:

$$L = \{x \in \{0,1\}^* : \exists \text{ certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}.$$

The constant c ensures that the size of the certificate y is polynomial in the problem size, and also we require that A runs in time polynomial in its input, which therefore must be polynomial in both $|x|$ and $|y|$.

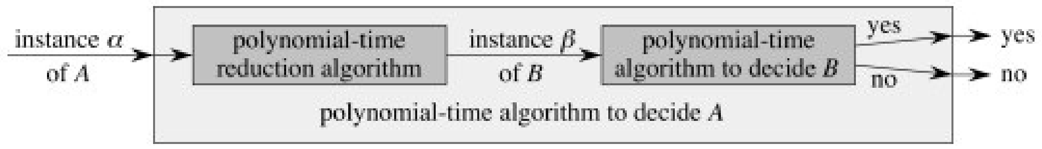
For example, although only exponential algorithms are known for the Hamiltonian Cycle problem, a proposed solution can be encoded as a sequence of vertices and verified in polynomial time.



NP Completeness

The NP-Complete problems are the "hardest" problems in NP, in that if one of them can be solved in polynomial time then every problem in NP can be solved in polynomial time. This relies on the concept of *reducibility*.

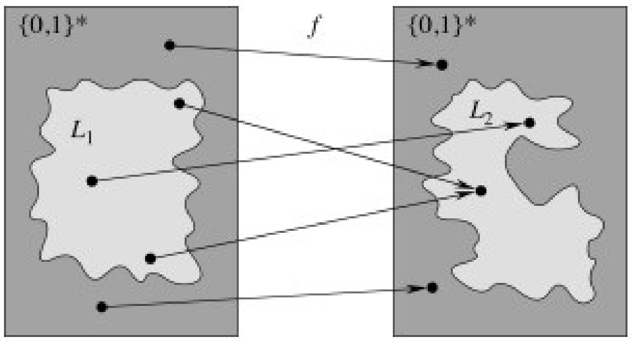
Reducibility


instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

A problem A can be polynomially reduced to a problem B if there exists a polynomial-time computable function f that converts an instance α of A into an instance β of B.

Stated in terms of formal languages, L_1 is reducible to L_2 if there exists a polynomial-time computable function f : $\{0, 1\}^* \rightarrow \{0, 1\}^*$ such that:

$$x \in L_1 \text{ iff } f(x) \in L_2, \forall x \in \{0, 1\}^*.$$



As shown in the figure, exactly those instances that are in L_1 are in L_2.

NP Completeness Defined

A language $L \subseteq \{0, 1\}^*$ is **NP-Complete (in NPC) if**

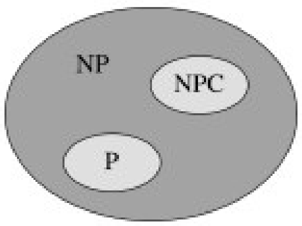
1. $L \in$ NP, and
2. **Every $L' \in$ NP is polynomial reducible to L.**

Languages satisfying 2 but not 1 are said to be **NP-Hard**. (This includes optimization problems that can be converted to decision problems in NP.)

The major Theorem of this lecture is:

> **If any NP-Complete problem is polynomial-time solvable, then P = NP.**
>
> Equivalently, **if any problem in NP is not polynomial-time solvable, then no NP-Complete problem is polynomial time solvable.**

The expected situation (but by no means proven) corresponds to the second statement of the theorem, as depicted to the right.



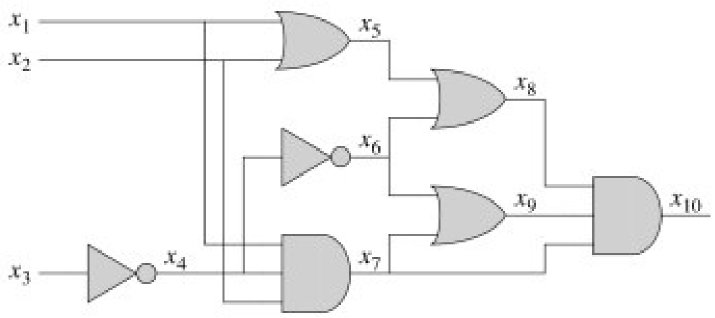
Constructing NPC

In order to construct the class NPC, we need to have one problem known to be in NPC. Then we can show that other problems are in NPC by reducibility proofs (reducing the other candidates to this known problem).

Circuit Satisfiability: An Initial Problem

In 1971, Cook defined the class NPC and proved that it is nonempty by proving that the **Circuit Satisfiability *(CIRCUIT-SAT)*** problem is NP-Complete.

This problem asks: given an acyclic boolean combinatorial circuit composed of AND, OR and NOT gates, does there exist an assignment of values to the input gates that produces a "1" at a designated output wire? Such a circuit is said to be **satisfiable**.



The first part of the proof, that CIRCUIT-SAT is in NP, is straightforward.

- Given as certifiate an assignment of boolean values to each of the wires in a circuit, one can check that the assignment meets the logic conditions of each component, which clearly can be done in polynomial time in the size of the description of the circuit.
- Alternatively, one can give as certificate only the input values, and verify by simulating the circuit, propagating values through the gates and to the designated output wire, in polynomial time.

The second part of the proof, that CIRCUIT-SAT is NP-Hard, was complex. The gist was as follows.

- Given any language L corresponding to a problem in NP, there must exist an algorithm A that verifies L in polynomial time $T(n)$, where n is the input size..

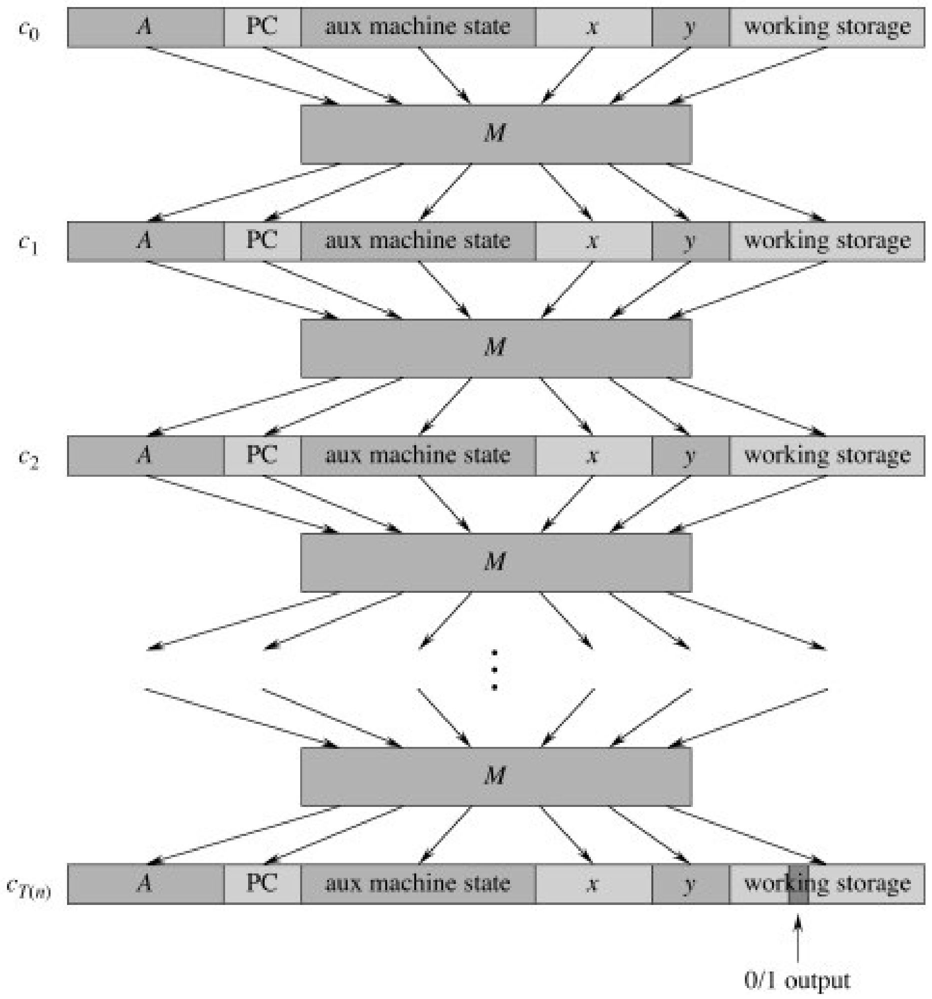
- We can represent the computation of A as a sequence of configurations or states of a machine M on which the algorithm runs. (A configuration represents the state of the computer, including program, storage and program counters.)

- First, a boolean circuit corresponding to M is constructed.

- Then the reduction makes a copy of the boolean circuit for each possible state (computational configuration) of M, feeding the output of one configuration into the input of another configuration in such a manner that computer memory is no longer needed: all state is encoded in the circuit wiring (c_i in the figure). The result is a single combinatorial circuit.

- Crucially, this transformation can be done in polynomial time. A verifies L in polynomial time, meaning that the number of state changes $T(n)$ is bounded by a polynomial function of the input size n, so we only need a polynomial number of copies of M with input and output wires for $c_0 \dots c_{T(n)}$ as shown in the figure. The size of each copy is also polynomially related to n. Thus the whole construction is polynomial in n.

- Then an algorithm that solves CIRCUIT-SAT can be used to simulate A, deciding L (we read the the circuit wire corresponding to the 0/1 decision in A's working memory).



NP Completeness Proofs by Reduction

Polynomial reduction is transitive:

> If L is a language such that L' reduces polynomially to L for some $L' \in$ NPC, then L is NP-Hard.
>
> Furthermore, if $L \in$ NP, then $L \in$ NPC.

Transitivity follows from the definitions and that the sum of two polynomials is itself polynomial.

This means that we can prove that other problems are in NPC without having to reduce every possible problem to them. The general procedure for proving that L is in NPC is:

1. **Prove $L \in$ NP (show one can check solutions in polynomial time).**
2. **Prove L is NP-Hard:**
   1. **Select a known language L' in NPC**
   2. **Describe an algorithm A that computes function f mapping *every* instance $x \in \{0, 1\}^*$ of L' to *some appropriately constructed* instance $f(x)$ of L.**
   3. **Prove that $x \in L'$ iff $f(x) \in L, \forall x \in \{0, 1\}^*$.**
   4. **Prove that A runs in polynomial time.**

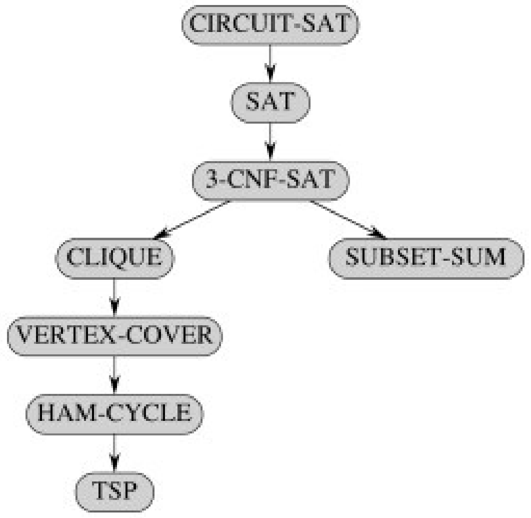
Important: Why doesn't mapping *every* instance of L to *some* instances of L' work?

Now we can populate the class NPC, first by reducing some problems to CIRCUIT-SAT, and then other problems to these new problems. Literally thousands of problems have been proven to be in NPC by this means. Let's look at a few.

NP Complete Problems

The text steps through reduction of problems as shown in the figure. We do not have time to go through the proofs, and it is more important that you are aware of the diversity of problems in NPC than that you are able to do NP-Completeness proofs. So we just list them briefly.

Satisfiability (SAT)


$$\begin{tikzcd} & \text{CIRCUIT-SAT} \arrow[d] & \\ & \text{SAT} \arrow[d] & \\ & \text{3-CNF-SAT} \arrow[dl] \arrow[dr] & \\ \text{CLIQUE} \arrow[d] & & \text{SUBSET-SUM} \\ \text{VERTEX-COVER} \arrow[d] & & \\ \text{HAM-CYCLE} \arrow[d] & & \\ \text{TSP} & & \end{tikzcd}$$

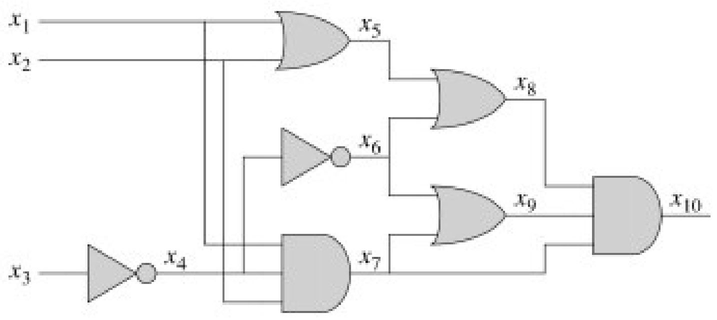
An instance is a boolean formula φ composed of

- n boolean variables $x_1 \ldots x_n$,
- m boolean connectives of two inputs and one output: $\wedge$ (logical **and**), $\vee$ (logical **or**), $\neg$ (logical **not**), $\rightarrow$ (logical **implication**), $\leftrightarrow$ (logical **equivalence**), and
- parentheses.

A **truth assignment** is a set of values for the variables of φ and a **satisfying assignment** is a truth assignment that evaluates to 1 (true).

SAT = $\{\langle\varphi\rangle : \varphi$ is a satisfiable boolean formula$\}$

There are 2^n possible assignments, but a given assignment can be checked in polynomial time.



CIRCUIT-SAT is reduced to SAT in polynomial time through a construction that turns CIRCUIT-SAT gates into small logical formulas for SAT:

1. Given any instance of CIRCUIT-SAT, define a variable x_i for each of the wires.
2. Write a formula φ that is the conjunction of the variable for the output wire plus a logical expression for each of the logic gates.

The resulting boolean formula is satisfied just when the circuit is satisfied. (You can verify that the formula shown is equivalent to the circuit.)

$$x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

This shows that we can reduce an arbitrary instance of CIRCUIT-SAT to a specialized instance of SAT in polynomial time. That means if we can solve SAT (e.g., in polynomial time) we can solve any instance of CIRCUIT-SAT in polynomially related time, *and* since we know that CIRCUIT-SAT is NPC, transitively we can use SAT to solve any instance of *any problem* in NP: **SAT ← CIRCUIT-SAT** and **CIRCUIT-SAT ← Problems in NP** implies **SAT ← Problems in NP**. Furthermore, only a polynomial cost is incurred in the translation, so the time required to solve SAT is polynomially related to that of the problems in NP.

Mapping an arbitrary instance of SAT to a specialized instance of CIRCUIT-SAT would not work. Such a reduction would go in the wrong direction to give logical transitivity: CIRCUIT-SAT ← SAT and CIRCUIT-SAT ← Problems in NP does not let us infer SAT ← Problems in NP).

3-Conjunctive Normal Form Satisfiability (3-CNF-SAT)

Reduction proofs require that we handle any possible case of a known NPC problem. It would be complicated to handle all the possible forms of SAT formulas, so it is useful to have a more restricted logical form for the target for reduction proofs. 3-CNF serves this purpose.

A **literal** in a boolean formula is an occurrence of a variable or its negation.

A boolean formula is in **conjunctive normal form (CNF)** if it is a conjunction of **clauses**, each of which is the disjunction of one or more literals.

A boolean formula is in **3-conjunctive normal form (3-CNF)** if each clause has exactly three distinct literals. For example:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

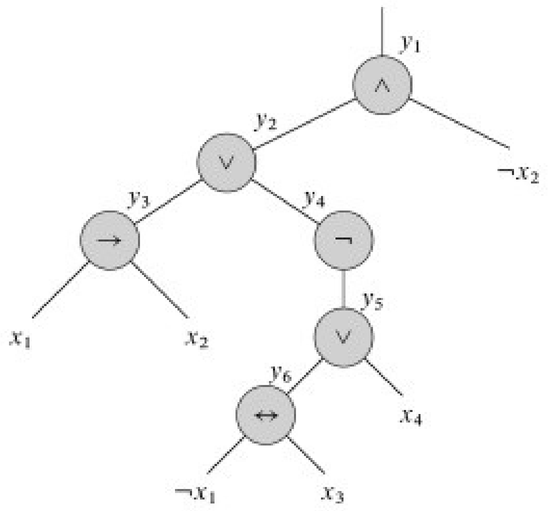
3-CNF-SAT asks whether a boolean formula is satisfiable by an assignment of truth values to the variables.

There are an exponential possible number of variable assignments, but a given one can be checked in polynomial time merely by substituting and evaluating the expression.

SAT can be reduced to 3-CNF-SAT through a polynomial-time process of:

1. parsing the SAT expression into a binary tree with literals as leaves and connectives as internal nodes;
2. introducing a variable y_i for the output of each internal node;
3. rewriting as the conjunction of the root variable and a clause for each node of the binary tree ($y_i \leftrightarrow$ the literal for its child nodes);
4. converting each clause to conjunctive normal form (see text), first by converting to disjunctive normal form and then applying DeMorgan's laws to convert to CNF; and then
5. supplying dummy variables as needed to convert clauses of 1 or 2 variables into 3-CNF.

For $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$, the tree is shown to the right and the expression resulting from the tree is shown below.



$$\begin{aligned} y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee x_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \end{aligned}$$

The remainder of the conversion uses DeMorgan's laws (see the text for the step by step description):

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$
$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$

resulting in:

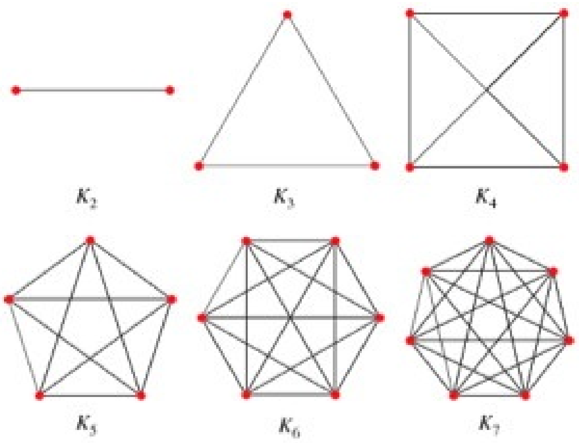
$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2).$$

CLIQUE

A **clique** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$, each pair of which is connected by an edge in E (a complete subgraph of G). (Examples of cliques of sizes between 2 and 7 are shown on the right.)

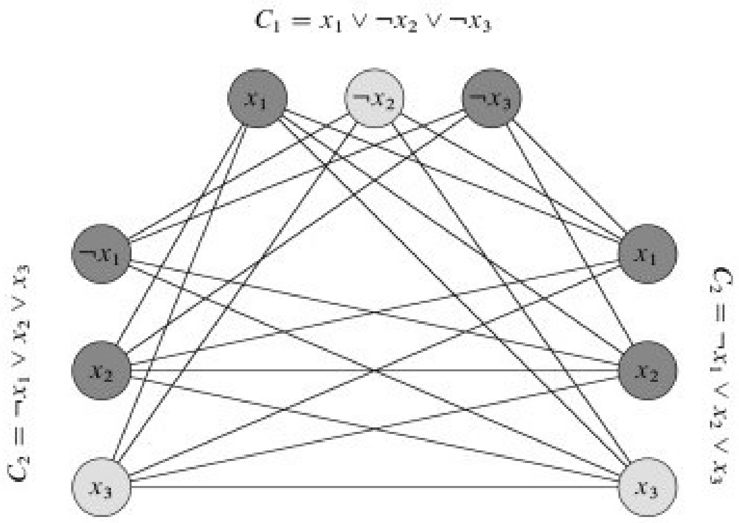
The **clique problem** is the problem of finding a clique of maximum size in G. This can be converted to a decision problem by asking whether a clique of a given size k exists in the graph:

CLIQUE = {⟨G, k⟩ : G is a graph containing a clique of size k}



One can check a solution in polynomial time. *(How?)*

3-


$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_2
x_3
x_1
x_2
x_3
$C_2 = \neg x_1 \vee x_2 \vee x_3$
$C_2 = \neg x_1 \vee x_2 \vee x_3$

CNF-SAT is reduced to CLIQUE by a clever reduction illustrated in the figure. Given an arbitrary formula φ in 3-conjunctive normal form with k clauses, we construct a graph G and ask if it has a clique of size k as follows:

- We create a vertex for every literal in φ
- For every pair of vertices, we create an edge between them if the corresponding literals are in different triples *and* the literals are consistent (i.e., one is not the negation of the other).

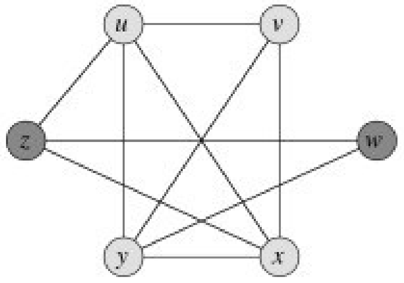
If there are k clauses in φ, we ask whether the graph has a k-clique. The claim is that such a clique exists in G if and only if there is a satisfying assignment for φ:

- **If** direction *(If clique exists in G, then there is a satisfying assignment in φ)*:
  - By definition of k-clique, existence of a k-clique implies there are k vertices in G that are all connected to each other.
  - By our construction, the fact that two vertices are connected to each other means that they can receive a consistent boolean assignment (we can assign 1 to all of them), *and* that they are in different clauses.
  - Since there are k vertices in the clique, then at least one literal in each of the k clauses can be assigned a 1, i.e., the formula φ can be satisfied.
- **Only if** direction *(If φ can be satisfied, then there is a clique in G)*:
  - If φ can be satisfied, then we can assign values to the literals, such that at least one literal in each clause is assigned value 1. I.e. they are consistent.
  - Consider the vertices corresponding to those literals. Since the literals are consistent and they are in different clauses, there is an edge between every pair of them.
  - Since there are k clauses in φ we have a subset of at least k vertices in the graph with edges between every pair of vertices, i.e., we have a k-clique in G.

Any arbitrary instance of 3-CNF-SAT can be converted to an instance of CLIQUE in polynomial time with this particular structure. That means if we can solve CLIQUE we can solve any instance of 3-CNF-SAT, and since we know that 3-CNF-SAT is NPC, transitively we can solve any instance in NP in polynomially related time. Be sure you understand why mapping an arbitrary instance of CLIQUE to a specialized instance of 3-CNF-SAT would not work.

Vertex Cover

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ or both.

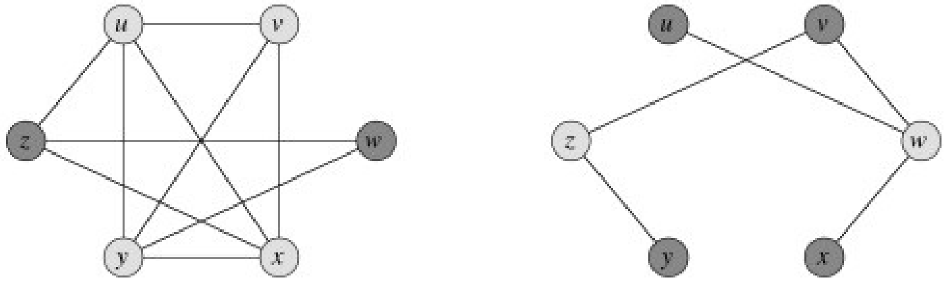


Each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. For example, in the graph on the right, $V' = \{v, z\}$ or $V' = \{v, w, y\}$ are vertex covers. Note $V' = V = \{u, v, w, x, y, z\}$ is also a vertex cover.

The **Vertex Cover Problem** is to find a vertex cover of minimum size in G. Phrased as a decision problem,

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has a vertex cover of size } k\}$$

There is a straightforward reduction of CLIQUE to VERTEX-COVER, illustrated in the figure. Given an instance $G=(V,E)$ of CLIQUE, one computes the complement of G, which we will call $G_C = (V,\bar{E})$, where $(u,v) \in \bar{E}$ iff $(u,v) \notin E$.



The graph G has a clique of size k iff the complement graph has a vertex cover of size $|V| - k$. (*Note* that this is an existence claim, not a minimization claim: a smaller cover may be possible.)

- **If** direction *(If G has a k-clique, then G_C has a vertex cover of size $|V| - k$)*:
  We show that none of the k vertices in the clique need to be in the cover.
  - They are all connected to each other in G, so none of them will be connected to each other in G_C.
  - Thus, every edge in G_C must involve at least one vertex not in the clique, so the clique vertices can be excluded from the cover: we can use vertices from the remaining $|V| - k$ vertices to cover all the edges in G_C.
  - The minimum vertex cover may be smaller than $|V| - k$, but we know that $|V| - k$ will work.

- **Only if** direction *(If G_C has a vertex cover of size $|V| - k$, then G has a k-clique)*:
  We will use proof by contrapositive to show that there is no clique of size k in G, then there is no vertex cover of size $|V| - k$ in G_C.
  - Assume for the sake of contradiction that there is no k-clique in G, but there is a vertex cover V' in G_C of size $|V'| = |V| - k$.
  - The non-existence of a k-clique in G means that at least two vertices in <u>every</u> subset of k vertices are not connected in G.
  - Consider the subset $V \setminus V'$, i.e. a subset of k vertices that are not part of the presumed $(|V| - k)$-sized vertex cover of G_C. By the above, there exist at least two vertices in this subset (let's call them u and v), such that there is no edge (u,v) in G.
  - But if edge (u,v) is not in G, it must exist in G_C (the complement of G).
  - But if there is an edge (u,v) in G_C, and neither u nor v are in the vertex cover, then edge (u,v) is uncovered and V' is not a valid vertex cover.

  We reached a contradiction, therefore, if G does not contain a k-clique, G_C cannot contain a vertex cover of size $|V| - k$

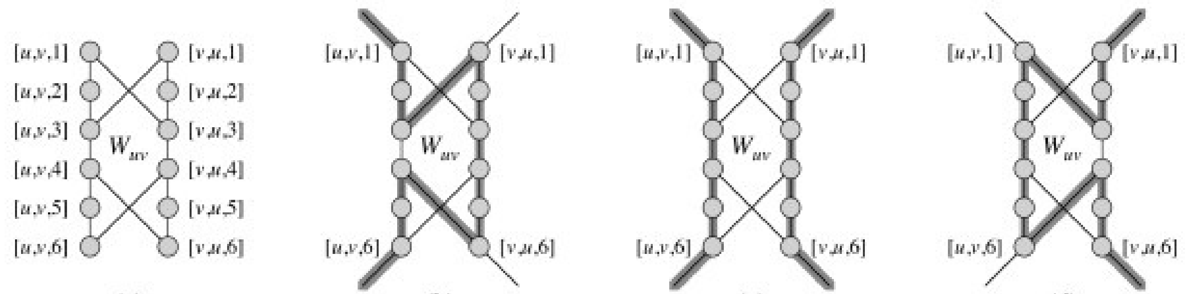
Hamiltonian Cycle (HAM-CYCLE)

A graph $G = (V,E)$ contains a ***Hamiltonian cycle*** if it contains a simple cycle C of size $|V|$. That is, G contains a cycle that visits every vertex exactly once.

HAM-CYCLE = $\{\langle G \rangle : G$ contains a Hamiltonian cycle$\}$

The Hamiltonian Cycle problem is shown to be in NPC by reduction of VERTEX-COVER to HAM-CYCLE.

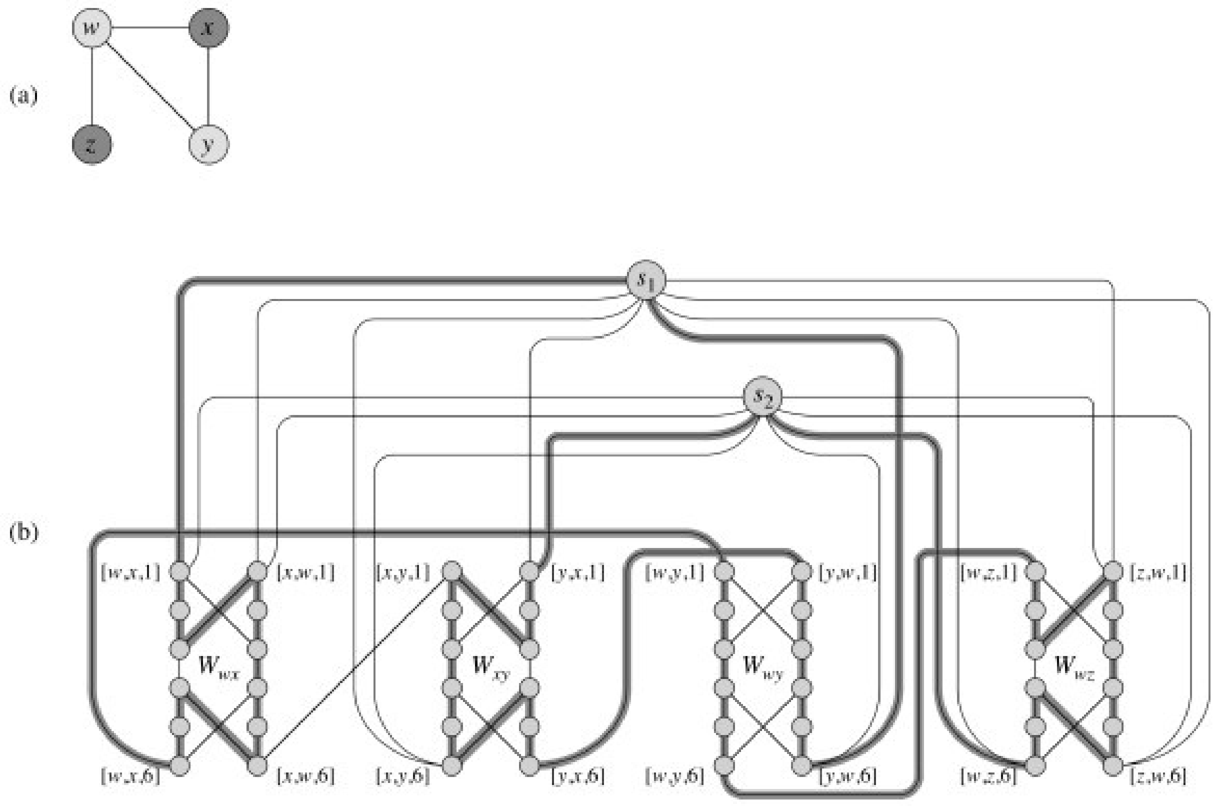
Given graph G — an instance of VERTEX-COVER — the construction converts edges of G into subgraph "widgets" shown in the figure (a) below. The Hamiltonian Cycle will go through every vertex of the widget associated with edge each (u,v) if and only if one or both of the vertices of the edge (u,v) are in the covering set. There is one such widget for every edge (u,v) in G.



(a) (b) (c) (d)

Any Hamiltonian cycle must include all the vertices in the widget (a), but there are only three ways to pass through each widget (b, c, and d in the figure). If only vertex u is included in the cover, we will use path (b); if only vertex v then path (d); otherwise path (c) to include both. (Not traversing the widget is not an option because at least one of the two vertices must be chosen to cover the edge.)

The widgets are then wired together in sequences that chain all the widgets that involve a given vertex, so if the vertex is selected all of the widgets corresponding to its edges will be reached.

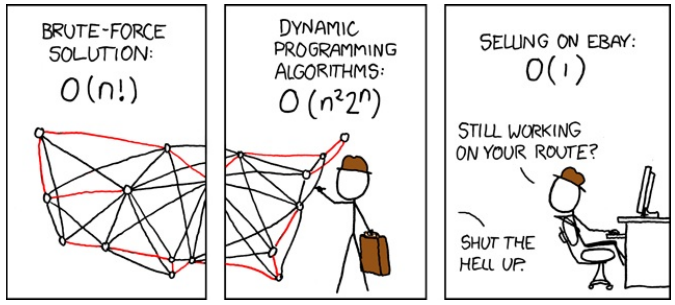


Finally, k selector vertices are added, and wired such that each will select the kth vertex in the cover of size k. I leave it to you to examine the discussion in the text, to see how clever these reductions can be!

Traveling Salesperson Problem (TSP)

One of the more famous NPC problems is TSP: Suppose you are a traveling salesperson, and you want to visit n cities exactly once in a Hamiltonian cycle, but choosing a **tour** with minimum cost.

TSP = $\{\langle G, c, k\rangle : G = (V, E)$ is a complete graph,
$c : V \times V \rightarrow \mathbb{N}$,
$k \in \mathbb{N}$, and
G has a traveling-salesperson tour with cost at most $k\}$



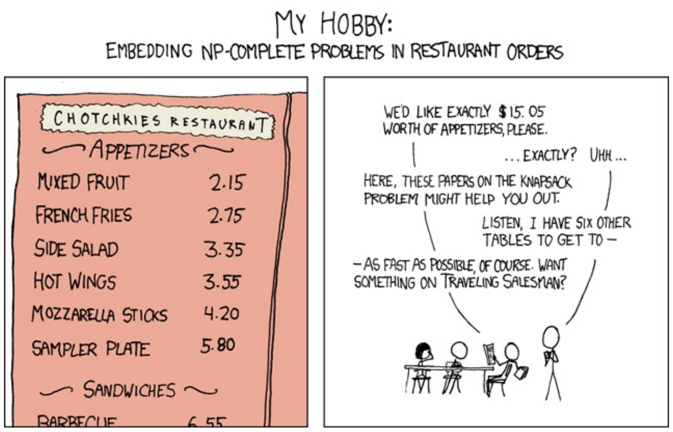
Only exponential solutions have been found to date, although it is easy to check a solution in polynomial time.

The reduction represents a HAM-CYCLE problem as a TSP problem on a complete graph, but with the cost of the edges in TSP being 0 if the edge is in the HAM-CYCLE problem, or 1 if not.

Subset-Sum Problem (SUBSET-SUM)

Many NP-Complete problems are of a numerical nature. We already mentioned integer linear programming. Another example is the subset-sum problem: given a finite set S of positive integers and an integer target $t > 0$, does there exist a subset of S that sums to t?

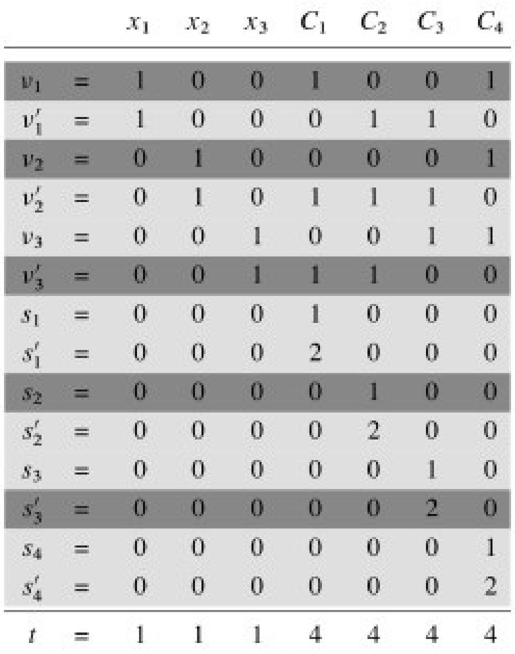
SUBSET-SUM = $\{\langle S, t : \exists$ subset $S' \subseteq S$ such that $t = \Sigma_{s \in S'} s\}$


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

The proof reduces 3-CNF-SAT to SUBSET-SUM. Please see the text for the details of yet another clever reduction!

Briefly:

- It involves



		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

  constructing two numbers for each variable x_i (one for the varaible and one for its negation), and two numbers for each clause C_j (these will hold "slack variables" needed to meet the target sum).
- The digits of the numbers are arranged in columns.
- The numbers for literals (variables and their negations) have a "1" in the column that indicates which variable it corresponds to, and also a "1" in the columns for the clauses in which that literal occurs.
- The numbers for clauses have either a 1 or a 2 in the column corresponding to that clause.
- The target value t has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.
- The only way one can achieve this target sum is to select those numbers corresponding to the literals that are satisfied in each clause, plus the required slack variables to reach 4.

For example, see how this clause maps to the table shown:

$$(x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3).$$

Other Problems

One can find large catalogs of other problems online, starting with those identified in the book Garey & Johnson (1979), Computers and Intractability. See for example Wikipedia. Following Garey & Johnson's outline they list problems in:

1. Graph theory
    1. Covering and partitioning
    2. Subgraphs and supergraphs
    3. Vertex ordering

   4. Iso- and other morphisms
   5. Miscellaneous
2. Network design
   1. Spanning trees
   2. Cuts and connectivity
   3. Routing problems
   4. Flow problems
   5. Miscellaneous
   6. Graph Drawing
3. Sets and partitions
   1. Covering, hitting, and splitting
   2. Weighted set problems
   3. Set partitions
4. Storage and retrieval
   1. Data storage
   2. Compression and representation
   3. Database problems
5. Sequencing and scheduling
   1. Sequencing on one processor
   2. Multiprocessor scheduling
   3. Shop scheduling
   4. Miscellaneous
6. Mathematical programming
7. Algebra and number theory
   1. Divisibility problems
   2. Solvability of equations
   3. Miscellaneous
8. Games and puzzles
9. Logic
   1. Propositional logic
   2. Miscellaneous
10. Automata and language theory
    1. 1 Automata theory
    2. 2 Formal languages
11. Computational geometry
12. Program optimization
    1. 1 Code generation
    2. 2 Programs and schemes
13. Miscellaneous

There are also problems for which their status is as of yet unknown. Much work to do!

If you want to learn more about NP-completeness and how to come up with NP-completeness proofs yourself, various complexity classes, and the limits of what computers are able to compute, you should take ICS 441.

Dan Suthers
Last modified: Sun Apr 26 10:50:59 EDT 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition; Garey & Johnson (1979), Computers and Intractability; and Weisstein, Eric W. "Complete Graph." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/CompleteGraph.html, xkcd.com, and possibly other sites whom I thank for not suing me.