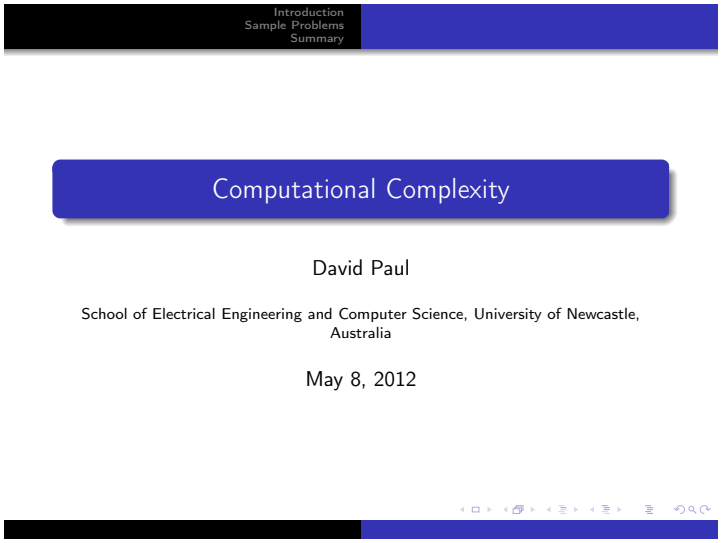
Computational Complexity

David Paul

May 8, 2012



1 Introduction

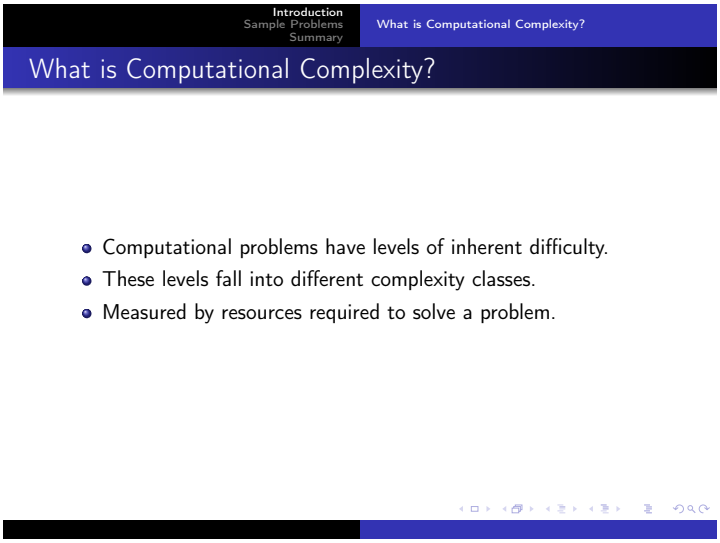
Before we begin, I want you to think about a problem that you've probably solved many times, but maybe haven't thought about too much. It's a problem that comes up very frequently in computer science, so it has been studied quite extensively. That problem is sorting.

Imagine that you have a deck of n cards, labeled from 1 to n, in some random order and you want them arranged from smallest to largest. How can we do that? The most naive way would be to randomly mix up the cards and see if they're in order. But there's a problem with this method: it's slow. There are $n!$ possible arrangements of the cards. According to physicists, the universe is around 13.5 billion years old, and contains something like 10^{80} atoms. If every atom in the universe was able to do one of the "shuffle and check" operations in a second, and each had been doing so since the big bang, with none ever repeating an ordering that had already been tested, then we'd have been guaranteed to have sorted just over 80 cards by now.

Obviously, there are better ways. Rather than just randomly shuffling, we could remove the first card and put it in a new pile, then remove the next card and put it in order in the new pile. Continuing in this way, we'd need fewer than n^2 steps, which is much better than $n!$. But n^2 is still pretty bad for a sorting algorithm. Typically, we think of something in the order of $n \log n$ as being a good level of performance. And, really, that's what computational complexity is all about - looking at how hard it is to actually solve various problems.

In this presentation we'll take a more formal look at exactly what computational complexity is, then look at a couple of "easy" problems, a few "hard" problems, and a couple of problems where we're not actually sure just how difficult they are. Along the way, we'll use a few techniques to help us determine just how difficult particular problems are, and be introduced to the $\mathbf{P} = \mathbf{NP}$ problem, one of the Clay Mathematics Institute's million-dollar Millennium Prize Problems, and probably the most important unsolved problem in computer science.

1.1 What is Computational Complexity?



So, what is computational complexity? As we saw with the sorting example, it's looking at how difficult problems are. Importantly, it's not looking at how well a particular solution works - if it was, then the "shuffle and check" algorithm for sorting cards would lead us to think that sorting was a very difficult task - but the inherent difficulty of the problem - how difficult is the problem using the best possible algorithm to solve it?

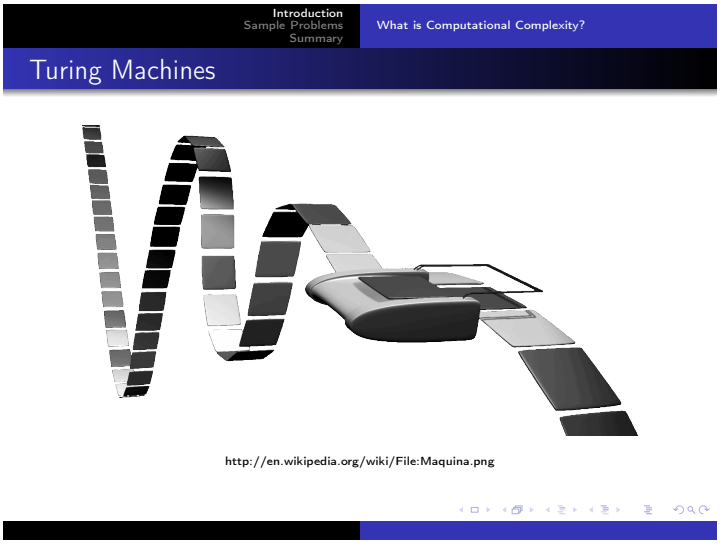
But how do we know if we have the best possible solution to a problem? It could be that there's a much better solution, but we just can't think of it. Well, there are ways that we can reason about problems to prove that they need at least a certain amount of resources to be solved correctly.

Using those tools, we can define a partial ordering on all problems, and use the partial ordering to place problems in different complexity classes. The problems can then be grouped in such a way that we can call some problems "harder" than other problems, or consider different problems to be essentially equivalent.

We'll assume that the resource being measured is the worst-case time required to solve the problem for an input of a particular size. We could just as easily study best- or average-case time, but considering the worst case gives us better bounds on just how difficult a problem is.

There's a problem though. We're measuring time, but what device do we use to solve the problem? If we use a desktop computer, we're obviously going to be slower than if we use a supercomputer, even if we're going through the exact same process, so it's not really fair to compare the two. Instead, we can use a formal model of computing that will guarantee that all comparisons are fair.

1.1.1 Turing Machines

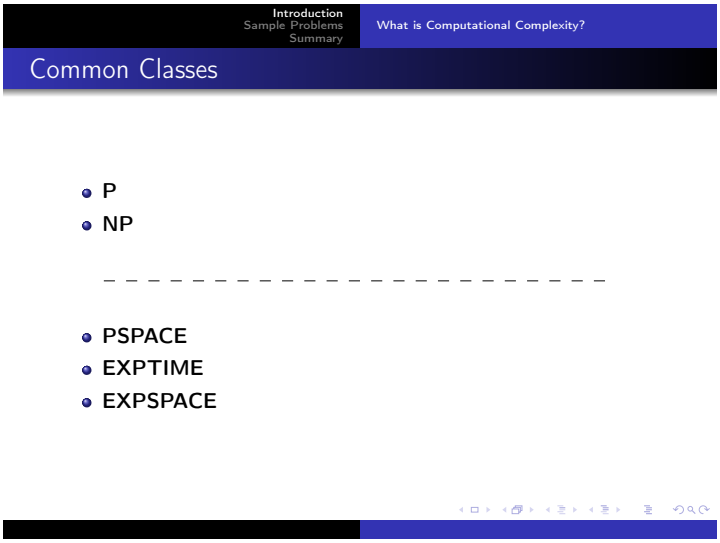


And the model we'll discuss here is Turing machines. A Turing machine consists of a tape reader that has an internal state, and can read from, write to, or move left and right along an infinite tape. The machine works by reading the symbol currently under the tape head and then looking up a table to determine what to do when it reads the input symbol in its current state. The machine then performs whichever operation the table defines, and possibly changes its internal state.

This seems a very simple model, but, in fact, such a machine is more powerful than any computer that has ever been built. The Turing machine can simulate anything we can do on a conventional computer, and removes some limits that any physical machine cannot. The reason for this is that any real-world system will have a finite storage limit, whereas a Turing machine has infinite storage space on its tape.

So, a Turing machine starts with some input on its tape, and follows its rules table to determine what to do. There are numerous variations of Turing machines, all of which are able to compute the same problems, but perform slightly differently. Here we'll consider two such variations. The first is a deterministic Turing machine, which has exactly one rule in its rule table for each state and input symbol. This means that at any step there is no decision to make - only one action is possible. In contrast, a non-deterministic Turing machine can have more than one rule for each internal-state/input-symbol combination. At each point the machine has a decision to make about which rule to follow. We assume that the machine always chooses the best rule to complete the given computation. This can be simulated by, at each decision point, making copies of the machine and having each copy run one of the different possible rules at that point. The fact that such a simulation can be modelled with a deterministic Turing machine shows that the two are computationally equivalent, though the non-deterministic machine's ability to "guess" the right rule to follow means that it will be much more efficient than the deterministic Turing machine, which actually has to try out each possibility.

1.1.2 Common Classes



Ok, so now we have a formal model of computation that ensures that we're able to fairly compare different solutions to problems. We still have the problem where we don't know if our particular solution is the best possible for the problem we're trying to solve, but we can at least now define some complexity classes and determine whether problems belong in particular classes.

Two of the most important complexity classes are the classes of **P** and **NP**. **P** is the class of problems for which any input of size n can be solved on a deterministic Turning machine in a number of steps that is less than some polynomial bound. So, sorting, for example, belongs in this class. The "shuffle and check" method isn't bound by a polynomial, but our other method of removing a card and inserting it into a new sorted pile in the correct position is bound by a polynomial of order n^2. We don't really care about what the actual polynomial is, since we don't know that our solution is the best possible - it certainly isn't in this case - but that result is enough to prove that sorting is in P.

The class NP, on the other hand, is the set of problems that can be solved in polynomial time on a non-deterministic Turing machine. We can think of this as the problems that, when we're given a possible solution, we can verify that the solution is correct in polynomial time on a deterministic Turing machine. As we said before, a non-deterministic Turing machine guesses the right rule to choose whenever it has a decision. By having an actual solution to look at, a deterministic Turing machine is essentially being told which rule to follow to get to the given solution, and that key allows the machine to solve the problem instance in polynomial time.

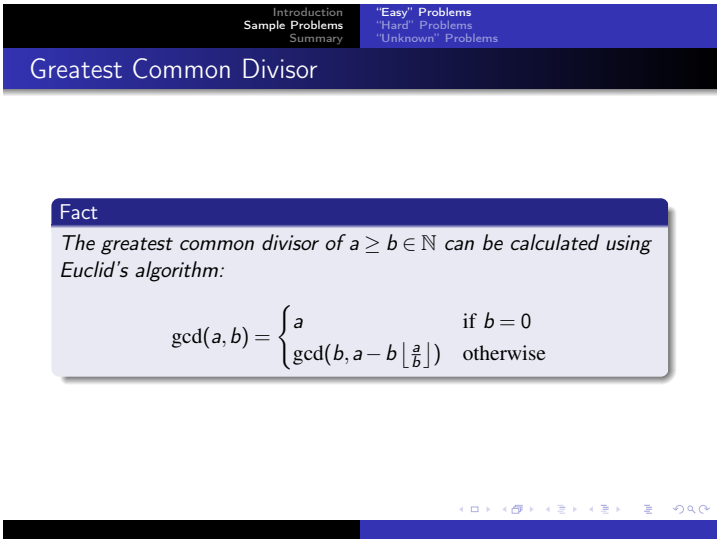
So, we can see that problems that are in **P** are definitely in **NP**. We can certainly determine whether a pack of cards is sorted correctly in polynomial time without making any guesses, for example. So, $\mathbf{P} \subseteq \mathbf{NP}$. Probably the greatest open question in computer science is whether or not $\mathbf{NP} \subseteq \mathbf{P}$, i.e. whether $\mathbf{P} = \mathbf{NP}$. General opinion is that they are different, but nobody's been able to prove it yet. If you do prove it, however, the Clay Mathematical Institute will give you a million dollars, since it's one of the unsolved Millennium prize problems.

We'll concentrate on **P** and **NP** in this presentation, but there are many other complexity classes too. **EXPTIME**, for example, are problems that can be solved in a number of steps bounded by an exponential function, while **PSPACE** and **EXPSPACE** are problems that can be solved using a polynomial or exponentially bound amount of storage space. From top to bottom of this slide, each complexity class is a subset of the classes below it, and for most of them we don't yet know whether it's a proper subset or not.

2 Sample Problems

2.1 "Easy" Problems

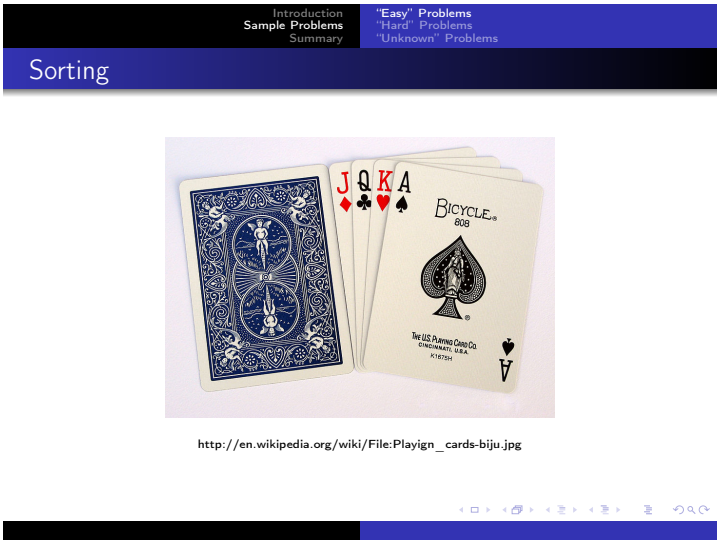
2.1.1 Greatest Common Divisor



Now we know that problems that are in **P** can be considered "easy" problems, so lets have a look at a couple. The first is one that should be a familiar algorithm for you: finding the greatest common divisor. Euclid came up with an algorithm to solve it, and this algorithm proves that the problem in in **P**.

It can actually be shown by induction that if this algorithm requires n steps for a pair of natural numbers $a > b$, then the smallest values for a and b are the Fibonacci numbers F_{n+2} and F_{n+1} respectively. Using this result, and facts about the Fibonacci numbers (in particular, how they relate to the golden ratio), it can be shown that the number of steps required for any a and b can never be more than 5 times the number of its digits, giving a polynomial bound on the size of the input.

2.1.2 Sorting

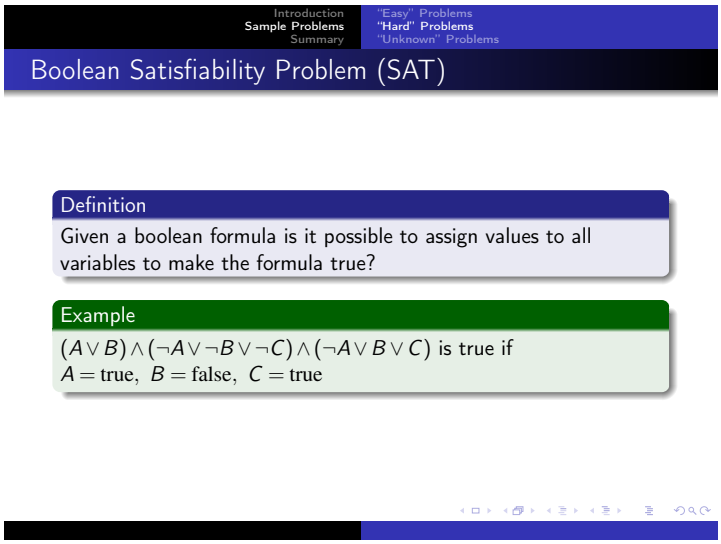


Sorting is another problem that we've already seen to be in **P**. There are many others: things like multiplication and division, finding a maximum matching in a graph, and even determining whether or not a number is prime.

But problems in *P* really are "easy". While many of them are very important, once we have a good algorithm for them, they're not really all that interesting from a computational complexity point-of-view. So let's move on to some "hard" problems.

2.2 "Hard" Problems

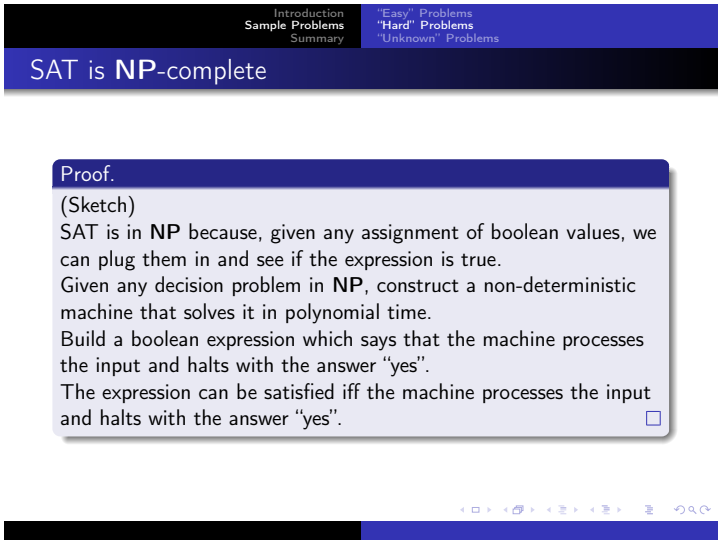
2.2.1 Boolean Satisfiability Problem (SAT)



The first "hard" problem we'll look at is boolean satisfiability, or SAT. Given some variables that can take the values "true" or "false", and some expression involving those variables, and logical and ($\wedge$), or ($\vee$), and negation ($\neg$) operators, can we assign values to the variables to make the expression true?

So, for example, given the expression $(A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee C)$ is true if $A = \text{true},\ B = \text{false},\ C = \text{true}$. Given that solution, we can easily check this in polynomial time - just plug in the values and move along the expression - so this problem is definitely in **NP**. But why do we consider this problem "hard"? It's entirely possible that this problem's in **P**, but it turns out that that's just as unlikely as $\mathbf{P} = \mathbf{NP}$, and I'll show you why.

2.2.2 SAT is NP-complete

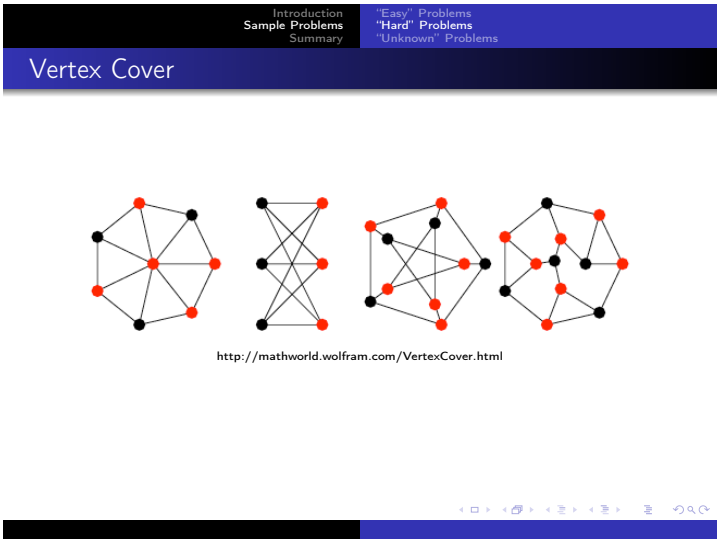


So, we've already seen that SAT is in **NP**. So let's assume we have some other decision problem that's in **NP** (a decision problem is just a problem that has an answer of either "yes" or "no" - this proof works on more general problems, but we'll just keep it to decision problems here). So, we have a problem in **NP**; that means we can construct a non-deterministic Turing machine that solves the problem in polynomial time.

But remember the concept of a universal Turing machine: that's a machine that takes as input a description of a Turing machine and some input for that machine, and simulates the machine that was input. Well, using something like that, we can build a boolean expression that says that the machine processes the given input and halts with the answer "yes". Further, this expression can be satisfied if and only if the machine can actually process the input and halt with a "yes" answer.

In other words, we can take any problem in **NP** and convert it to an instance of SAT. This means that SAT is at least as difficult as any other problem in **NP** - there's no problem in **NP** that's harder than SAT. Since SAT is also in **NP**, we say that SAT is **NP**-complete; if we have a good solution to SAT, then we have a good solution to every problem in **NP**, since we could just convert any other **NP** problem to a SAT instance in polynomial time and then solve the SAT problem. In particular, if we find a polynomial-time solution for SAT, then we show that SAT$\in$ **P**, which would mean that **P** = **NP**. So, the greatest problem in computer science is equivalent to determining if SAT has a polynomial-time solution.

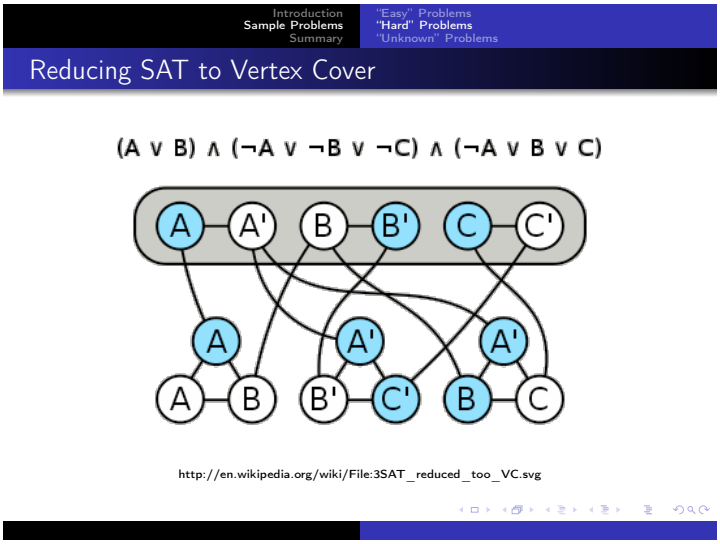
2.2.3 Vertex Cover



Now let's consider another problem that's in **NP**: vertex cover. Imagine you had an art gallery with priceless art along each corridor. You want to hire guards so you have a guard looking down each corridor, but guards are expensive, so you want to hire the minimum number required. That's the vertex cover problem: given a graph and a natural number k, is there a set K of no more than k vertices such that for each edge in the graph, as least one of the vertices attached to the edge is in K?

This problem is obviously in **P** - looking at the red vertices in these graphs, we can easily check to see if they form a vertex cover, and given k we can also determine whether there are no more than k included in the cover.

2.2.4 Reducing SAT to Vertex Cover



So, vertex cover is in **NP**, but I claim that it's also **NP**-complete. to prove this, I'll first show that every instance of SAT can be written as a set of conjunctive clauses where each has size exactly three.

Consider the expressions A and $A \vee B$. These are equivalent to $A \vee A \vee A$ and $A \vee A \vee B$, respectively.

Now assume that there's a clause with more than three variables, say $A_1 \vee A_2 \vee A_3 \ldots A_n$. This is equivalent to $(A_1 \vee A_2 \vee B) \wedge (\neg B \vee A_3 \ldots A_n)$, where B is a new variable that doesn't appear in the clause otherwise. Repeating this process, we can get all expressions to have terms with exactly three variables being "or"ed all being "and"ed together. Such an expression is called an instance of 3SAT.

Now, we can see how each 3SAT expression can be converted into a vertex cover problem that only has a solution if the 3SAT problem has a solution.

For each variable A, we create two nodes A and $\neg A$, connected by an edge. To cover that edge, it means that either A or $\neg A$ must be in the cover. Then, for each clause, we create a triangle with each node labelled after one of the terms in each 3SAT clause. Two of these must be included to cover all of the edges in the triangle. We then connect all the the nodes labelled A in each triangle to the node labelled A in the set of pairs, and do similarly for $\neg A$. If there are n variables and m clauses, then the 3SAT instance has a "yes" solution iff our vertex cover instance has a cover of size $2m+n$.

Thus, we've shown that any instance of 3SAT can be converted into vertex cover problem, meaning that vertex cover is at least as difficult as SAT. But SAT is **NP**-complete, meaning that SAT is at least as difficult as vertex cover. This means that the two are computationally equivalent, meaning that vertex cover is also **NP**-complete.

This shows that once we have an **NP**-complete problem, it's enough to reduce it to another problem in **NP** to show that the other problem is also **NP**-complete. We'll use that to our advantage when examining the next problem.

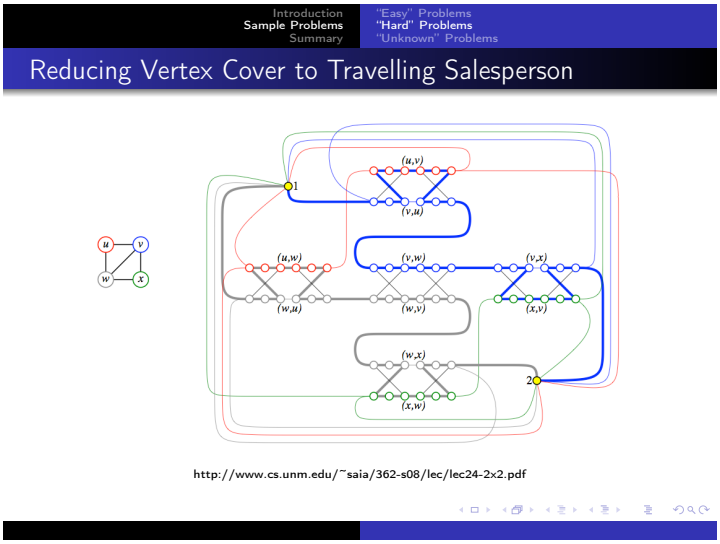
2.2.5 Travelling Salesperson

The next problem is the travelling salesperson problem. Say you have a heap of vacuum cleaners, and you want to go around Sweden to try and sell them. You want to visit each city in Sweden, but you want to travel the minimum distance possible. How do you do it? Well, this map here shows an optimal solution.

The travelling salesperson problem can more generally be described as: given a graph with edge weights and a number k, is it possible to construct a path that travels through each vertex only once such that the sum of the edge weights in the path is less than k.

Again, this is obviously in **NP** - given a solution, we check that the distance is less than k and that it visits each node exactly once. That's easily doable in polynomial time.

2.2.6 Reducing Vertex Cover to Travelling Salesperson



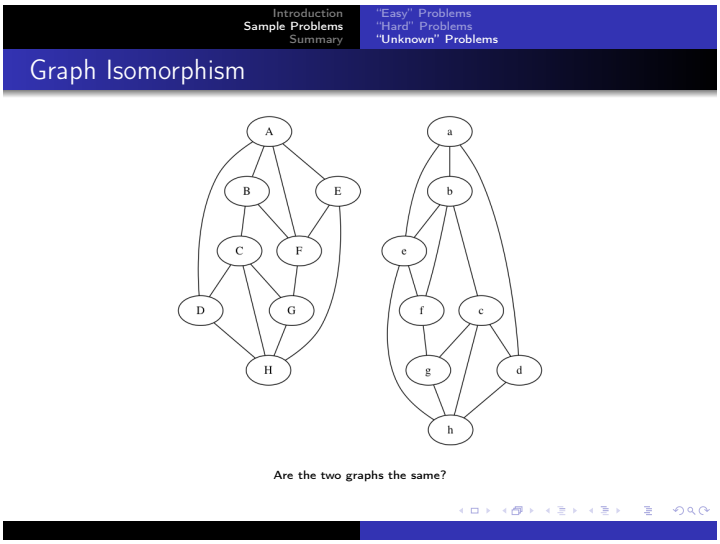
So, how do we show that travelling salesperson is also *NP*-complete? Well, we could reduce SAT to an instance of travelling salesperson, but since it's a graph problem, it might be best to go from vertex cover. So, given a general instance of vertex cover, can we convert that into an instance of travelling salesperson that has a "yes" answer only if the vertex cover instance has a "yes" answer?

Given an instance of vertex cover where we are looking for a cover of size k, we create k nodes labelled 1 to k. Then, for each edge (u,v) in the vertex cover instance, we create a set of twelve nodes, 6 connected in two lines with crosses between the first and third, and fourth and sixth nodes of each line. For each neighbour of a node, we then set up a vertex chain and connect the left- and right-most vertices of the chain to each of the vertices labelled 1 through k. If we set the edge weight to one for each edge in this graph, then there is a vertex cover solution only if there is a travelling salesperson solution.

So, our set of **NP**-complete problems keep growing. And each time we get a new one, we can use it to help show that other similar problems are also **NP**-complete. There are currently hundreds of problems that are known to be **NP**-complete, such as Tetris, minesweeper, and graph colouring. If we could find a polynomial solution for any of these problems, we'd prove that $\mathbf{P} = \mathbf{NP}$, but the fact that we haven't found such a solution for so many different problems is what leads most people too believe that there are problems in **NP** that are not in **P**.

2.3 “Unknown” Problems

2.3.1 Graph Isomorphism

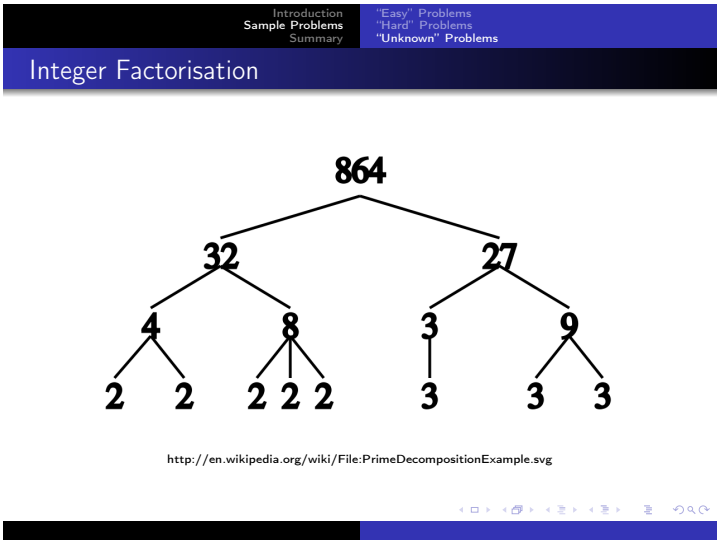


So, we’ve seen “easy” problems that are solvable in polynomial time, and we’ve seen the most difficult problems that , given a solution, can be verified in polynomial time, but is there anything in between? Actually, there’s remarkably few problems in **NP** that haven’t been proven to be either in **P** or **NP**-complete.

However, one such problem is graph isomorphism. Given two graphs, is it possible to map the labels of one graph’s vertices onto the labels of the other graph and have an identical structure?

This problem is obviously in **NP**, since we could go through all the vertices in a solution and make sure they have the right edge connections, but the problem hasn’t yet been shown to be either in **P** or to be **NP**-complete. It has been proven for certain subsets of graphs: tree isomorphisms for example are in **P**, but the more general graph problem is still unknown.

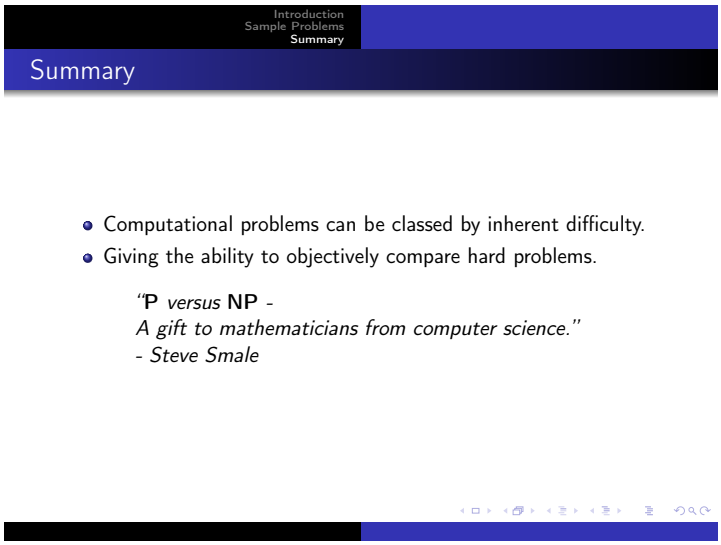
2.3.2 Integer Factorisation



Another unknown problem is integer factorisation: Given an integer n, does it have a factor less than k? This is interesting because the problem of determining whether an integer is prime or not is in P, but we still have no idea about actual factorisation.

A lot of the world relies on the assumption that factoring integers is hard to do. Modern cryptography is pretty much all based on that premise. If you visit your banks Website, then, behind the scenes, your computer and the bank's system are sharing integers with each other that only one or the other side know the factors of. The systems then use the fact that the other system knows those secret numbers to verify that they're talking to the correct person, and to encrypt the data so that only a person who knows those numbers can understand the message they've been sent.

3 Summary



So, in summary, computational complexity is reasoning about problems to find information about their inherent difficulty. Using this reasoning, we can place problems in complexity classes such as **P** and **NP**, and we can even often show that two problems are computationally equivalent.

When it comes to **P** and **NP**, one of the greatest unknowns is whether the two are equal, though with every new problem that is proven to be *NP*-complete without having a known polynomial-time solution, the evidence seems to point to their being different.

So, I'll leave with this quote from Fields medal winner Steve Smale: "**P** versus **NP** - A gift to mathematicians from computer science." - see, we're not really all that different at all.