Lecture 22:
Introduction to NP-completeness (1997)

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Among n people, a celebrity is defined as someone who is known by everyone but does not know anyone. We seek to identify the celebrity (if one is present) by asking questions of the form "Hey, x, do you know person y?". Show how to find the celebrity using $O(n)$ questions.

Note that there are n^2 possible questions to ask, so we cannot ask them all.

What if we ask 1 if she knows 2, and 2 if she knows 1? If both know each other neither can be a celebrity. If neither know each other, neither can be a celebrity. If one of them knows the other, the former cannot be a celebrity.

Thus in two questions we can eliminate at least one person

from celebrity status. Thus in $2(n-1)$ questions, we have only one possible celebrity. It is now possible to check whether the survivor is really a celebrity using $n-1$ additional queries, by checking whether everyone else knows them.

An Eulerian cycle in a graph visits each edge exactly once. A graph contains an Eulerian cycle iff it is connected and the degree of each vertex is even. Give an $O(|E|)$ algorithm to find an Eulerian cycle if one exists.

Observe that an cycle of edges defines a graph where each vertex is of degree 2. Thus deleting a cycle from an Eulerian graph leaves each vertex with even degree, although the graph may not be connected.

We can use depth-first search to decompose the edges of a graph into cycles. If the graph was connected, these cycles must link together. Splicing them together gives an Eulerian cycle. For example, the cycle $(1, 2, 3, 1)$ and $(4, 5, 6, 1, 4)$ can be spliced together as $(4, 5, 6, 1, 2, 3, 1, 4)$.

Although Eulerian cycle has an efficient algorithm, the Hamiltonian cycle problem (visit each vertex exactly once) is NP-complete.

The Theory of NP-Completeness

Several times this semester we have encountered problems for which we couldn't find efficient algorithms, such as the traveling salesman problem. We also couldn't prove an exponential time lower bound for the problem.

By the early 1970s, literally hundreds of problems were stuck in this limbo. The theory of NP-Compleness, developed by Stephen Cook and Richard Karp, provided the tools to show that all of these problems were really the same problem.

Polynomial vs. Exponential Time

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f(n) = n!$
10	0.01 μs	0.1 μs	1 μs	3.63 ms
20	0.02 μs	0.4 μs	1 ms	77.1 years
30	0.03 μs	0.9 μs	1 sec	8.4×10^{15} years
40	0.04 μs	1.6 μs	18.3 min	
50	0.05 μs	2.5 μs	13 days	
100	0.1 μs	10 μs	4×10^{13} years	
1,000	1.00 μs	1 ms		

The Main Idea

Suppose I gave you the following algorithm to solve the *bandersnatch* problem:

Bandersnatch(G)

> Convert G to an instance of the Bo-billy problem Y.
> Call the subroutine Bo-billy on Y to solve this instance.
> Return the answer of Bo-billy(Y) as the answer to G.

Such a translation from instances of one type of problem to instances of another type such that answers are preserved is called a *reduction*.

Now suppose my reduction translates G to Y in $O(P(n))$:

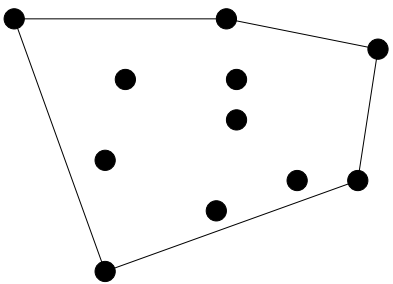
1. If my Bo-billy subroutine ran in $O(P'(n))$ I can solve the Bandersnatch problem in $O(P(n) + P'(n))$

2. If I know that $\Omega(P'(n))$ is a lower-bound to compute Bandersnatch, then $\Omega(P'(n) - P(n))$ must be a lower-bound to compute Bo-billy.

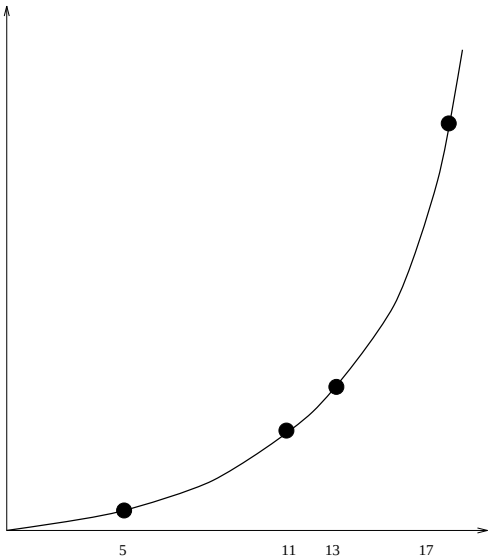
The second argument is the idea we use to prove problems hard!

Convex Hull and Sorting

A nice example of a reduction goes from sorting numbers to the convex hull problem:



We must translate each number to a point. We can map x to (x, x^2).



Why? That means each integer is mapped to a point on the parabola $y = x^2$.

Since this parabola is convex, every point is on the convex hull. Further since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by x-coordinate, ie. the original numbers.

Sort(S)

For each $i \in S$, create point (i, i^2).
Call subroutine convex-hull on this point set.
From the leftmost point in the hull,
read off the points from left to right.

Creating and reading off the points takes $O(n)$ time.
What does this mean? Recall the sorting lower bound of $\Omega(n \lg n)$. If we could do convex hull in better than $n \lg n$, we could sort faster than $\Omega(n \lg n)$ – which violates our lower

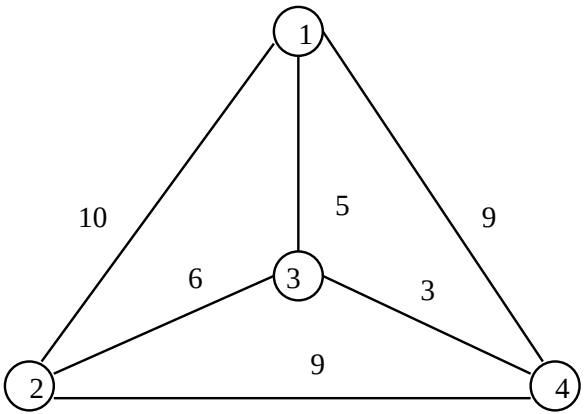
bound.

Thus convex hull must take $\Omega(n \lg n)$ *as well!!!*

Observe that any $O(n \lg n)$ convex hull algorithm also gives us a complicated but correct $O(n \lg n)$ sorting algorithm as well.

What is a problem?

A *problem* is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution.
An instance is a problem with the input parameters specified.
Example: The Traveling Salesman
Problem: Given a weighted graph G, what tour $\{v_1, v_2, ..., v_n\}$ minimizes $\Sigma_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$.
Instance: $d[v_1, d_2] = 10$, $d[v_1, d_3] = 5$, $d[v_1, d_4] = 9$, $d[v_2, d_3] = 6$, $d[v_2, d_4] = 9$, $d[v_3, d_4] = 3$



Solution: $\{v_1, v_2, v_3, v_4\}$ cost= 27

A problem with answers restricted to yes and no is called a *decision problem*. Most interesting optimization problems can be phrased as decision problems which capture the essence of the computation.

Example: The Traveling Salesman Decision Problem.

Given a weighted graph G and integer k, does there exist a

traveling salesman tour with cost $\leq k$?
Using binary search and the decision version of the problem we can find the optimal TSP solution.
For convenience, from now on we will talk *only* about decision problems.
Note that there are many possible ways to encode the input graph: adjacency matrices, edge lists, etc. All reasonable encodings will be within polynomial size of each other.
The fact that we can ignore minor differences in encoding is important. We are concerned with the difference between algorithms which are polynomial and exponential in the size of the input.

Satisfiability

Consider the following logic problem:
Instance: A set V of variables and a set of clauses C over V.
Question: Does there exist a satisfying truth assignment for C?
Example 1: $V = v_1, v_2$ and $C = \{\{v_1, \overline{v}_2\}, \{\overline{v}_1, v_2\}\}$
A clause is satisfied when at least one literal in it is TRUE. C is satisfied when $v_1 = v_2 =$TRUE.
Example 2: $V = v_1, v_2$,

$$C = \{\{v_1, v_2\}, \{v_1, \overline{v}_2\}, \{\overline{v}_1\}\}$$

Although you try, and you try, and you try and you try, you can get no satisfaction.

There is no satisfying assigment since v_1 must be FALSE (third clause), so v_2 must be FALSE (second clause), but then the first clause is unsatisfiable!

For various reasons, it is known that satisfiability is a hard problem. Every top-notch algorithm expert in the world (and countless other, lesser lights) have tried to come up with a fast algorithm to test whether a given set of clauses is satisfiable, but all have failed. Further, many strange and impossible-to-believe things have been shown to be true if someone in fact did find a fast satisfiability algorithm.

Clearly, Satisfiability is in NP, since we can guess an assignment of TRUE, FALSE to the literals and check it in polynomial time.

P versus NP

The precise distinction between whether a problem is in P or NP is somewhat technical, requiring formal language theory and Turing machines to state correctly.

However, intuitively a problem is in P, (ie. polynomial) if it can be solved in time polynomial in the size of the input.

A problem is in *NP* if, given the answer, it is possible to verify that the answer is correct within time polynomial in the size of the input.

Example P – Is there a path from s to t in G of length less than k.

Example NP – Is there a TSP tour in G of length less than k. Given the tour, it is easy to add up the costs and convince me

it is correct.

Example *not NP* – How many TSP tours are there in G of length less than k. Since there can be an exponential number of them, we cannot count them all in polynomial time.

Don't let this issue confuse you – the important idea here is of reductions as a way of proving hardness.

3-Satisfiability

Instance: A collection of clause C where each clause contains exactly 3 literals, boolean variable v.
Question: Is there a truth assignment to v so that each clause is satisfied?
Note that this is a more restricted problem than SAT. If 3-SAT is NP-complete, it implies SAT is NP-complete but not visa-versa, perhaps long clauses are what makes SAT difficult?! After all, 1-Sat is trivial!
Theorem: 3-SAT is NP-Complete
Proof: 3-SAT is NP – given an assignment, just check that each clause is covered. To prove it is complete, a reduction from $Sat \propto 3 - Sat$ must be provided. We will transform

each clause independantly based on its *length*.
Suppose the clause C_i contains k literals.

- If $k = 1$, meaning $C_i = \{z_1\}$, create two new variables v_1, v_2 and four new 3-literal clauses:

  $\{v_1, v_2, z_1\}$, $\{v_1, \overline{v}_2, z_1\}$, $\{\overline{v}_1, v_2, z_1\}$, $\{\overline{v}_1, \overline{v}_2, z_1\}$.

  Note that the only way all four of these can be satisfied is if z is TRUE.

- If $k = 2$, meaning $\{z_1, z_2\}$, create one new variable v_1 and two new clauses: $\{v_1, z_1, z_2\}$, $\{\overline{v}_1, z_1, z_2\}$

- If $k = 3$, meaning $\{z_1, z_2, z_3\}$, copy into the 3-SAT instance as it is.

- If $k > 3$, meaning $\{z_1, z_2, ..., z_n\}$, create $n - 3$ new

variables and $n-2$ new clauses in a chain: $\{v_i, z_i, \overline{v}_i\}$, ...

If none of the original variables in a clause are TRUE, there is no way to satisfy all of them using the additional variable:

$$(F, F, T), (F, F, T), \ldots, (F, F, F)$$

But if any literal is TRUE, we have $n-3$ free variables and $n-3$ remaining 3-clauses, so we can satisfy each of them.
$(F, F, T), (F, F, T), \ldots, (\mathbf{F}, \mathbf{T}, \mathbf{F}), \ldots, (T, F, F), (T, F, F)$
Since any SAT solution will also satisfy the 3-SAT instance and any 3-SAT solution sets variables giving a SAT solution – the problems are equivallent. If there were n clauses and m total literals in the SAT instance, this transform takes $O(m)$ time, so SAT and 3-SAT.

Note that a slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete. However, it breaks down when we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses. It turns out that resolution gives a polynomial time algorithm for 2-SAT. Having at least 3-literals per clause is what makes the problem difficult. Now that we have shown 3-SAT is NP-complete, we may use it for further reductions. Since the set of 3-SAT instances is smaller and more regular than the SAT instances, it will be easier to use 3-SAT for future reductions. Remember the direction to reduction!

$$Sat \propto 3 - Sat \propto X$$

A Perpetual Point of Confusion

Note carefully the direction of the reduction.
We must transform *every* instance of a known NP-complete problem to an instance of the problem we are interested in. If we do the reduction the other way, all we get is a slow way to solve x, by using a subroutine which probably will take exponential time.
This always is confusing at first - it seems bass-ackwards. Make sure you understand the direction of reduction now - and think back to this when you get confused.

Integer Programming

Instance: A set v of integer variables, a set of inequalities over these variables, a function $f(v)$ to maximize, and integer B.

Question: Does there exist an assignment of integers to v such that all inequalities are true and $f(v) \geq B$?

Example:

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 3$$

A solution to this is $v_1 = 1$, $v_2 = 2$.

Example:

$$v_1 \geq 1, \quad v_2 \geq 0$$

$$v_1 + v_2 \leq 3$$

$$f(v) : 2v_2, \quad B = 5$$

Since the maximum value of $f(v)$ given the constraints is $2 \times 2 = 4$, there is no solution.

Theorem: Integer Programming is NP-Hard

Proof: By reduction from Satisfiability

Any set instance has boolean variables and clauses. Our Integer programming problem will have twice as many variables as the SAT instance, one for each variable and its compliment, as well as the following inequalities:

For each variable v_i in the set problem, we will add the following constraints:

- $1 \leq V_i \leq 0$ and $1 \leq \overline{V}_i \leq 0$

Both IP variables are restricted to values of 0 or 1, which makes them equivalent to boolean variables restricted to true/false.

- $1 \leq V_i + \overline{V}_i \leq 1$

  Exactly one of the IP variables associated with a given sat variable is 1. This means that exactly one of V_i and $\overline{V}_i$ are true!

- for each clause $C_i = \{v_1, \overline{v}_2, \overline{v}_3 \ldots v_n\}$ in the sat instance, construct a constraint:

$$v_1 + \overline{v}_2 + \overline{v}_3 + \ldots v_n \geq 1$$

  Thus at least one IP variable must be one in each clause! Thus satisfying the constraint is equivalent to satisfying

the clause!

Our maximization function and bound are relatively unimportant: $f(v) = V_1\ B = 0$.
Clearly this reduction can be done in polynomial time.

We must show:

1. Any SAT solution gives a solution to the IP problem.

   In any SAT solution, a TRUE literal corresponds to a 1 in the IP, since if the expression is SATISFIED, at least one literal per clause in TRUE, so the sum in the inequality is ≥ 1.

2. Any IP solution gives a SAT solution.

   Given a solution to this IP instance, all variables will be 0 or 1. Set the literals correspondly to 1 variable TRUE and the 0 to FALSE. No boolean variable and its complement will both be true, so it is a legal assignment with also must satisfy the clauses.

Neat, sweet, and NP-complete!

Things to Notice

1. The reduction preserved the structure of the problem. Note that the reduction did not *solve* the problem – it just put it in a different format.
2. The possible IP instances which result are a small subset of the possible IP instances, but since some of them are hard, the problem in general must be hard.
3. The transformation captures the essence of why IP is hard - it has nothing to do with big coefficients or big ranges on variables; for restricting to $0/1$ is enough. A careful study of what properties we do need for our reduction tells us a lot about the problem.

4. It is not obvious that IP $\leq$ NP, since the numbers assigned to the variables may be too large to write in polynomial time - don’t be too hasty!