The Theory of NP-Completeness

Several times this semester we have encountered problems for which we couldn't find efficient algorithms, such as the travelling 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 $O(P(n))$:

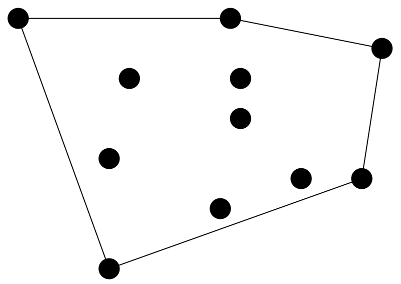
1. If my Bo-billy subroutine ran in $O(P'(n))$ I can solve the Bandersnatch problem in $O(P(n)+P'(n))$

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.

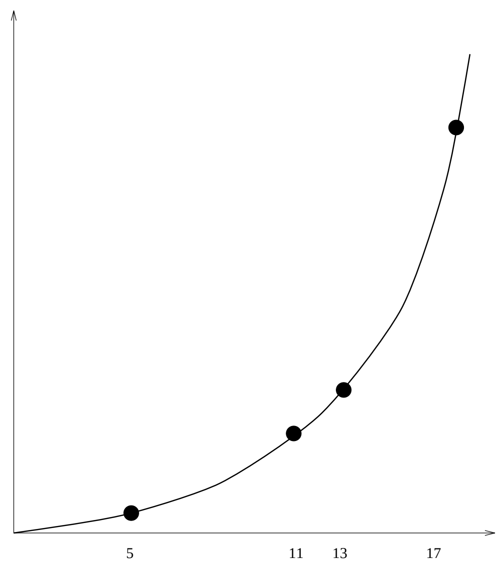
The second argument is the idea we use to prove problems hard!

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?

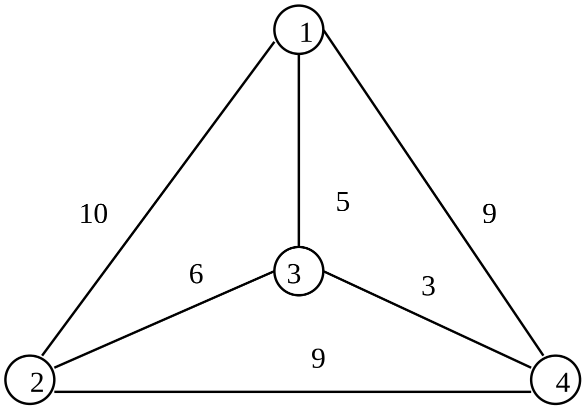
A *problem* is a general question, with parameters for the input and conditions on what is a satisfactory answer or solution.

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 $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$.

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 travelling 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 clause 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!

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