Lecture 21:
Other Reductions

Steven Skiena

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

http://www.cs.sunysb.edu/∼skiena

Problem of the Day

Show that the *Dense subgraph* problem is NP-complete:
Input: A graph G, and integers k and y.
Question: Does G contain a subgraph with exactly k vertices and at least y edges?

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*.

What Does this Imply?

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!

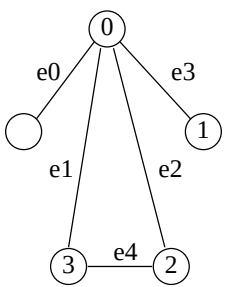
Integer Partition (Subset Sum)

Instance: A set of integers S and a target integer t.
Problem: Is there a subset of S which adds up exactly to t?
Example: $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ and $T = 3754$
Answer: $1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = T$
Observe that integer partition is a number problem, as opposed to the graph and logic problems we have seen to date.

Integer Partition is *NP*-complete

To prove completeness, we show that vertex cover $\propto$ integer partition. We use a data structure called an incidence matrix to represent the graph G.

	e4	e3	e2	e1	e0
v0	0	1	1	1	1
v1	0	1	0	0	0
v2	1	0	1	0	0
v3	1	0	0	1	0
v4	0	0	0	0	1



How many $1'$s are there in each column? Exactly two.
How many $1'$s in a particular row? Depends on the vertex degree.

Using the Incidence Matrix

The reduction from vertex cover will create $n + m$ numbers from G.

The numbers from the vertices will be a base-4 realization of rows from the incidence matrix, plus a high order digit:

$x_i = 4^{|E|} + \Sigma_{j=0}^{|E|-1} b[i,j] \times 4^j$

ie. $V_2 = 10100$ becomes $4^5 + (4^4 + 4^2)$.

The numbers from the edges will be $y_i = 4^j$.

The target integer will be

$$t = k \times 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \times 4^j$$

Why?

Each column (digit) represents an edge. We want a subset of vertices which covers each edge. We can only use k x vertex/numbers, because of the high order digit of the target.
$x_0 = 100101 = 1041$ $x_2 = 111000 = 1344$ $y_1 = 000010 = 4$
We might get only one instance of each edge in a cover - but we are free to take extra edge/numbers to grab an extra 1 per column.

VC in $G \rightarrow$ Integer Partition in S

Given k vertices covering G, pick the k cooresponding vertex/numbers. Each edge in G is incident on one or two cover vertices. If it is one, includes the cooresponding edge/number to give two per column.

Integer Partition in $S \rightarrow VC$ in G

Any solution to S must contain *exactly* k vertex/numbers. Why? It cannot be more because the target in that digit is k and it cannot be less because, with at most 3 $1'$s per edge/digit-column, no sum of these can carry over into the next column. (This is why base-4 number were chosen).

This subset of k vertex/numbers must contain at least one edge-list per column, since if not there is no way to account for the two in each column of the target integer, given that we can pick up at most one edge-list using the edge number. (Again, the prevention of carrys across digits prevents any other possibilites).

Neat, sweet, and *NP*-complete!

Notice that this reduction could not be performed in polynomial time if the number were written in unary $5 = 11111$. Big numbers is what makes integer partition hard!

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.

Integer Programming is NP-Hard

We use a reduction from Satisfiability
Any SAT 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 IP variable associated with a given SAT variable is 1. Thus 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 $= 1$ in each clause! Satisfying the constraint equals 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.

Why?

Any SAT solution gives a solution to the IP problem – A TRUE literal in SAT corresponds to a 1 in the IP. If the expression is satisfied, at least one literal per clause must be TRUE, so the sum in the inequality is ≥ 1.

Any IP solution gives a SAT solution – All variables of any IP solution are 0 or 1. Set the literals corresponding to 1 to be TRUE and those of 0 to FALSE. No boolean variable and its complement will both be true, so it is a legal assignment which satisfies the clauses.

Neat, sweet, and NP-complete!

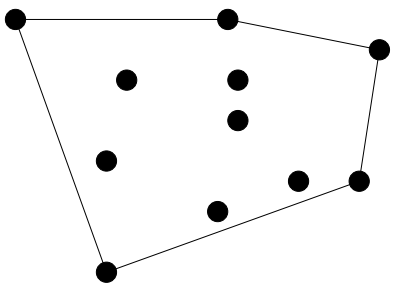
Things to Notice

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.

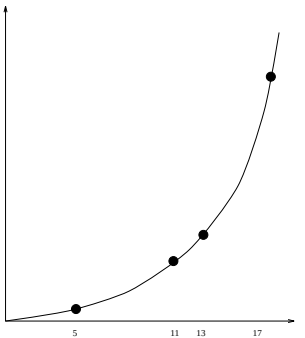
4. It is not obvious that IP $\leq$ NP, since the numbers assigned to the variables may be too large to write in polynomial time - don’t be too hasty!

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).



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.

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.

P versus NP

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

Classifiying Example Problems

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.