1 Definition of Reduction

Problem A is *reducible*, or more technically *Turing reducible*, to problem B, denoted $A \leq B$ if there a main program M to solve problem A that lacks only a procedure to solve problem B. The program M is called the *reduction* from A to B. (Note that we use the book's notation $\propto_T$ and $\leq$ interchangeably.) A is *polynomial-time reducible* to B if M runs in polynomial time. A is *linear-time reducible* to B if M runs in linear time and makes at most a constant number of calls to the procedure for B.

Assume that the reduction runs in time $A(n)$, exclusive of $R(n)$ recursive calls to B on an input size of $S(n)$. Then if one plugs in an algorithm for B that runs in time $B(n)$, one gets an algorithm for A that runs in time $A(n) + R(n) \cdot B(S(n))$.

A *decision problem* is a problem where the output is 0/1. Let A and B be decision problems. We say A is *many-to-one reducible* to B if there is reduction M from A to B of following special form:

- M contains exactly one call to the procedure for B, and this call is the last step in M.
- M returns what the procedure for B returns.

Equivalently, A is *many-to-one reducible* to B if there exists a computable function f from instances of A to instances of B such that for all instances I of A it is the case that I has output 1 in problem A if and only if $f(I)$ has output 1 in problem B.

Reductions can be used to both find efficient algorithms for problems, and to provide evidence that finding particularly efficient algorithms for some problems will likely be difficult. We will mainly be concerned with the later use.

2 Using Reductions to Develop Algorithms

Assume that A is some new problem that you would like to develop an algorithm for, and that B is some problem that you already know an algorithm for. Then showing $A \leq B$ will give you an algorithm for problem A.

Consider the following example. Let A be the problem of determining whether n numbers are distinct. Let B be the sorting problem. One way to

solve the problem A is to first sort the numbers (using a call to a black box sorting routine), and then in linear time to check whether any two consecutive numbers in the sorted order are equal. This give an algorithm for problem A with running time $O(n)$ plus the time for sorting. If one uses an $O(n \log n)$ time algorithm for sorting, like mergesort, then one obtains an $O(n \log n)$ time algorithm for the element uniqueness problem.

3 Using Reductions to Show that a Problem is Hard

4 Matrix Squaring is as Hard as Matrix Multiplication

You want to find an $O(n^2)$ time algorithm to square an n by n matrix. The obvious algorithm runs in time $\Theta(n^3)$. You know that lots of smart computer scientists have tried to find an $O(n^2)$ time algorithm for multiply two matrices, but have not been successful. But it is at least conceivable that squaring a matrix (multiplying identical matrices) might be an easier problem than multiply two arbitrary matrices. To show that in fact that matrix squaring is not easier than matrix multiplication we linear-time reduce matrix multiplication to matrix squaring.

Theorem: If there is an $O(n^2)$ time algorithm to square an n by n matrix then there is an $O(n^2)$ time algorithm to multiply two arbitrary n by n matrices

Proof: We show that Matrix multiplication is linear time reducible to Matrix Squaring. We exhibit the following reduction (program for matrix multiplication):

- Read X, and Y, the two matrices we wish to multiply.
- Let
$$I = \begin{bmatrix} 0 & Y \\ X & 0 \end{bmatrix}$$
- Call procedure to compute
$$I^2 = \begin{bmatrix} XY & 0 \\ 0 & XY \end{bmatrix}$$

- Read XY of from the top left quarter of I^2.

If you plug in an $O(B(n)$ time algorithm for squaring, the running time of the resulting algorithm for squaring is $O(n^2 + B(2n))$. Thus an $O(n^2)$ time algorithm for squaring yields an $O(n^2)$ time algorithm for matrix multiplication.
End Proof:

So the final conclusion is: Since lots of smart computer scientists have tried to find an $O(n^2)$ time algorithm for multiply two matrices, but have not been successful, you probably shouldn't waste a lot of time looking for an $O(n^2)$ time algorithm to square a matrix.

5 Convex Hull is as Hard as Sorting

The convex hull problem is defined below:
INPUT: n points $p_1, \ldots, p_n$ in the Euclidean plane.
OUTPUT: The smallest (either area or perimeter, doesn't matter) convex polygon that contain all n points. The polygon is specified by a linear ordering of its vertices.

You would like to find an efficient algorithm for the convex hull problem. You know that lots of smart computer scientists have tried to find an a linear time algorithm for sorting, but have not been successful. We want use that fact to show that it will be hard to find a linear time algorithm for the convex hull problem.
Theorem: If there is an $O(n)$ time algorithm for convex hull then there is an $O(n)$ time algorithm for sorting.
Proof: We need to show that sorting $\leq$ convex hull via a linear time reduction. Consider the following reduction (algorithm for sorting).

- Read $x_1, \ldots, x_n$
- Let $p_i = (x_i, x_i^2)$ and $p_{n+1} = (0, 0)$.
- Call procedure to compute the convex hull C of the points $p_1, \ldots, p_{n+1}$.
- In linear time, read the sorted order of the first coordinates off of C by traversing C counter-clockwise.

If you plug in an $O(B(n)$ time algorithm for the convex hull problem, the running time of the resulting algorithm for sorting is $O(n + B(n + 1))$. Thus an $O(n)$ time algorithm for the convex hull problem yields an $O(n)$ time algorithm for matrix multiplication.
End Proof:

So the final conclusion is: Since lots of smart computer scientists have tried to find an $O(n)$ time algorithm for sorting, but have not been successful, you probably shouldn't waste a lot of time looking for an $O(n)$ time algorithm to solve the convex hull problem.

6 3-Colinear is as Hard as 3-Sum

Consider the following 3-Sum problem:

INPUT: n numbers $z_1, \ldots, z_n$
OUTPUT 1 if there exists an i, j and k such that $z_i + z_j + z_k = 0$, and 0 otherwise.

The most obvious algorithm for 3-Sum runs in time $\Theta(n^3)$. By using sorting and binary search this can be relatively easily be reduced to $\Theta(n^2 \log n)$. But no one knows of a linear time algorithm, or even a $O(n^{1.99})$ time algorithm. So 3-Sum is our known hard problem.

Consider the following "new" problem, 3-Colinear:

INPUT: n points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ in the plane
OUTPUT: 1 if there exists three points that are colinear, which means that they lie on a common line, and 0 otherwise.

We are interested in giving evidence that finding a $o(n^{1.99})$ time algorithm for 3-Colinear is a challenging problem. In particular we want to show that 3-Colinear is at least as hard as 3-Sum. To accomplish this we reduce 3-Sum to 3-Colinear with a linear time many-to-one reduction as follows:

- 3-SumAlgorithm$(z_1, \ldots, z_n)$.
- $p_i = (z_i, z_i^3)$ for $1 \leq i \leq n$.
- return 3-ColinearAlgorithm$(p_1, \ldots, p_n)$

The correctness of this algorithm follows from the algebraic fact that $(a, a^3), (b, b^3), (c, c^3)$ are colinear if and only if $a + b + c = 0$. Assume for

the moment that $a < b < c$. The the line segment between (a, a^3) and (b, b^3) has slope $(b^3 - a^3)/(b - a)$. The the line segment between (b, b^3) and (c, c^3) has slope $(c^3 - b^3)/(c - b)$. Thus these three points will colinear iff $(b^3 - a^3)/(b - a) = (c^3 - b^3)/(c - b)$. A bit of algebra reveals that this is equivalent to $a + b + c = 0$.

7 NP-complete/NP-equivalent Problems

There are a class of problem called NP-complete problems. For our purposes we use NP-complete and NP-equivalent interchangeably, although there is a technical difference that not really relevant to us. The following facts about these NP-complete are relevant:

1. If any NP-complete has a polynomial time algorithm then they all do. Another way to think of this is that all pairs of NP-complete problems are reducible to each other via a reduction that runs in polynomial time.

2. There are thousands of known natural NP-complete problems from every possible area of application that people would like to find polynomial time algorithms for.

3. No one knows of a polynomial time algorithm for any NP-complete problem.

4. No one knows of a proof that no NP-complete problem can have a polynomial time algorithm.

5. Almost all computer scientists believe that NP-complete problems do not have polynomial time algorithms.

A problem L is *NP-hard* if a polynomial time algorithm for L yields a polynomial time algorithm for any/all NP-complete problem(s), or equivalently, if there is an NP-complete problem C that polynomial time reduces to L. A problem L is *NP-easy* if a polynomial time algorithm for any NP-complete problem yields a polynomial time algorithm for L, or equivalently, if there is an NP-complete problem C such that L is polynomial time reducible to C. A problem is *NP-equivalent*, or for us NP-complete if it is both NP-hard and NP-easy.

Fact: The decision version of the CNF-SAT problem (determining if a satisfying assignment exists) is NP-complete

8 Self-Reducibility

An optimization problem O is *polynomial-time self-reducible* to a decision problem D if there is a polynomial time reduction from O to D. Almost all natural optimization problems are self-reducible. Hence, there is no great loss of generality by considering only decision problems. As an example, we now that the CNF-SAT problem is self-reducible.

Theorem: If there is a polynomial time algorithm for the decision version of CNF-SAT, then there is a polynomial time algorithm to find a satisfying assignment.

Proof: Consider the following polynomial time reduction from the problem of finding a satisfying assignment to the decision version of the CNF-SAT problem.

We consider a formula F. One call to the procedure for the decision version of CNF-SAT will tell whether F is satisfiable or not. Assuming that F is satisfiable, the following procedure will produce a satisfying assignment.

Pick an arbitrary variable x that appears in F. Create a formula F' by removing clauses in F that contain the literal x, and removing all occurrences of the literal $\bar{x}$. Call a procedure to see if F' is satisfiable. If F' is satisfiable, make x equal to true, and recurse on F'. If F' is not satisfiable then create a formula F'' by removing clauses in F that contain the literal $\bar{x}$, and removing all occurrences of the literal x. Make x false, and recurse on F''.

End Proof

9 The Equivalence of Independent Set, Clique and Vertex Cover

Theorem: If one of Independent Set, Clique, and Vertex Cover has a polynomial-time algorithm then all of these problems have a polynomial time algorithm.

10 3SAT is NP-hard

The 3SAT problem is defined as follows:

INPUT: A Boolean Formula F in CNF with exactly 3 literals per clause

OUTPUT: 1 if F is satisfiable, 0 otherwise.

The CNF-SAT problem is defined as follows:

INPUT: A Boolean Formula F in conjunctive normal form

OUTPUT: 1 if F is satisfiable, 0 otherwise.

Assume CNF-SAT is known to be NP-complete.

Theorem: 3SAT is NP-hard.

Proof: We exhibit the following polynomial time many-to-one reduction from CNF-SAT to 3SAT.

Main Program for CNF-SAT: Read the formula F. Create a new formula G by replacing each clause C in F by a collection $g(C)$ of clauses. We get several cases depending on the number of literals in C.

1. If C contain one literal x then $g(C)$ contains the clauses

$$x \vee a_C \vee b_C$$

$$x \vee a_C \vee \bar{b}_C$$

$$x \vee \bar{a}_C \vee b_C$$

and

$$x \vee \bar{a}_C \vee \bar{b}_C$$

2. If C contains two literals x and y then $g(C)$ contains the clauses

$$x \vee y \vee a_C$$

and

$$x \vee y \vee \bar{a}_C$$

3. If C contains 3 literals then $g(C) = C$.

4. If $C = (x_1 \vee \ldots \vee x_k)$ contains $k \geq 4$ literals then $g(C)$ contains the clauses

$$x_1 \vee x_2 \vee a_{C,1}$$

$$x_3 \vee \bar{a}_{C,1} \vee a_{C,2}$$

$$x_4 \vee \bar{a}_{C,2} \vee a_{C,3}$$

$$\cdots$$

$$x_{k-2} \vee \bar{a}_{C,k-4} \vee a_{C,k-3}$$

and

$$x_{k-1} \vee x_k \vee \bar{a}_{C,k-3}$$

Note that in each case a_C, b_C, $a_{C,1}, \ldots, a_{C,k-3}$ are new variables that appear nowhere else outside of $g(C)$.

Clearly G can be computed in polynomial time. G has exactly 3 literals per clause. Furthermore, F is satisfiable if and only if G is satisfiable.
End Proof

This type of reduction, in which each part of the instance is replaced or modified independently, is called *local replacement*.

11 1-in-3-SAT is NP-hard

The 1-in-3-SAT problem is defined as follows:
INPUT: A collection of clauses with exactly three distinct literals per clause. Each literal is a Boolean variable or the negation of a Boolean variable.
OUTPUT: 1 if there is an assignment to the variables that makes exactly one literal per clause true, and 0 otherwise.

Theorem: 1-in-3-SAT is NP-hard.

Begin proof:

We show 3SAT is polynomial time reducible to 1-in-3-SAT.

Local replacement: Replace each clause $\{x \vee y \vee z\}$ in 3SAT instance F by the following three clauses in the 1-in-3-SAT instance G:

$$\{\bar{x}, a, b\}$$

$$\{y, b, c\}$$

$$\{\bar{z}, c, d\}$$

Note that x, y and z are literals, not necessarily variables. Now the claim is that F is satisfiable iff one literal per clause can be assigned true in G.

Assume F is satisfiable, and let A be a satisfying assignment. We construct an assignment B of the variables for G. The literals x, y and z have the same value in G as they have in F. Then if y is 1 in A then b and c are 0 in B. Further a in G is the opposite of $\bar{x}$, and d in G is the opposite of $\bar{z}$. If y and x are 0 in A, then a, and b are 0 in B, and c and d are 1 in B. If y and z are 0 in A, then a, and b are 1 in B, and c and d are 0 in B. One can check that in every case B assigns exactly one literal per clause true in G.

Now assume B is an assignment that satisfies one literal per clause in G. We construct an assignment A for F where the value of x, y and z in A is identical to the assignment in B. To show that A is a satisfying assignment, we just need to show that its not possible that all of x, y and z are 0 in B. To see this note that if y is 0 in B then exactly 1 of b or c would be 1 in B. If b is 1 in B then the first clause would have two variables equal to true, contradicting the fact that B assigns exactly 1 literal per clause to be true. The same problem occurs with clause 3 if c is 1 in B.

End proof:

12 Vertex cover is NP-hard

The Vertex Cover Problem is defined as follows:
INPUT: Graph G and integer k
Output: 1 if G have a vertex cover of size k or less.

A *vertex cover* is a collection S of vertices such that every edge in G is incident to at least one vertex in S.

Theorem: Vertex cover is NP-hard

Proof: We show 3SAT is polynomial-time many-to-one reducible to Vertex Cover. Given the formula F in 3SAT form we create a G and a k as follows. For each variable x inF we create two vertices x and $\bar{x}$ and connect them with an edge. Then for each clause C we create three vertices C_1, C_2 and C_3, connected by three edges. We then put an edge between C_i, $1 \leq i \leq 3$, and the vertex corresponding to the ith literal in C. Let $k =$ the number of variables in F plus twice the number of clauses in F.

We now claim that G has a vertex cover of size k if and only if F is satisfiable. To see this note that a triangle corresponding to a clause requires at least 2 vertices to cover it and and edge between a variable and its negation requires at least 1 vertex.

13 Subset Sum is NP-hard

Theorem: Subset Sum is NP-hard

Proof: We show that 3SAT is polynomial-time many-to-one reducible to Subset Sum. This is by example. Consider the formula (x or y or not z) and (not x or y or z) and (x or not y or z) We create one number of each variable and two numbers for each clause as follows. Further the target L is set as shown.

Numbers name:	Base 10 representation					
				clause 1	clause 2	clause 3
x	1	0	0	1	0	1
not x	1	0	0	0	1	0
y	0	1	0	1	1	0
not y	0	1	0	0	0	1
z	0	0	1	0	1	1
not z	0	0	1	1	0	0
clause 1	0	0	0	1	0	0
slop	0	0	0	1	0	0
clause 2	0	0	0	0	1	0
slop	0	0	0	0	1	0
clause 1	0	0	0	0	0	1
slop	0	0	0	0	0	1
L	1	1	1	3	3	3

14 Why the Dynamic Programming Algorithm for Subset Sum is not a Polynomial Time Algorithm

Recall that the input size is the number of bits required to write down the input. And a polynomial time algorithm should run in time bounded by a polynomial in the input size.

The input is n integers $x_1, \ldots, x_n$ and an integer L. The inputs size is then

$$\log_2 L + \sum_{i=1}^{n} \log_2 x_i$$

which is at least $n + \log_2 L$.

The running time of the algorithm is $O(nL)$. Note that this time estimate assumed that all arithmetic operations could be performed in constant, which may be a bit optimistic for some inputs. Never the less, if n is constant. Then the input size $I = \Theta(\log L)$ and the running time is $L = \Theta(2^I)$ is exponential in the input size.

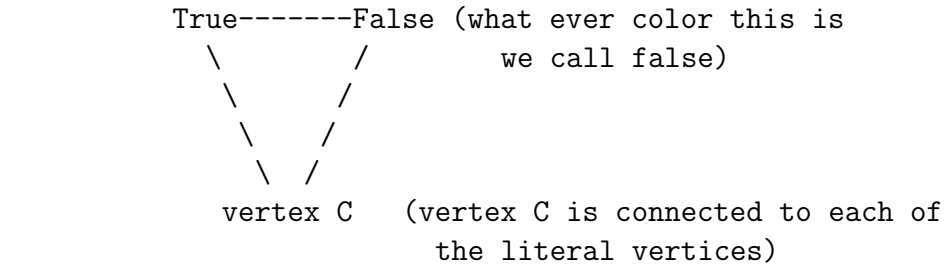
A most concrete way to see this is to consider an instance with $n = 3$ and $L = 2^{1000}$. The input size is about 1002 bits, or 120 bytes. But any algorithm that requires 2^{1000} steps is not going to finish in your life time.

15 3-coloring is NP-hard

The 3-Coloring Problem is defined as follows:
INPUT: Graph G
OUTPUT: 1 if the vertices of G can be colored with 3 colors so that no pair of adjacent vertices are coloring the same, and 0 otherwise.
Theorem: 3-Coloring is NP-hard
Proof: We show that 3SAT is polynomial-time many-to-one reducible to 3-Coloring.

Once again we do this by example. Consider formula F = (x or y or not z) and (not x or y or z) and (x or not y or z) We create the graph G as follows. For each variable x in F we create two vertices x and not x and connect them. We then create a triangle, with three new vertices True, False, and C, and connect C to each literal vertex.

So for the example above we would get something like:



x-----not x y ------ not y z------- not z

We then create the gadget/subgraph shown in figure 1 for each clause.

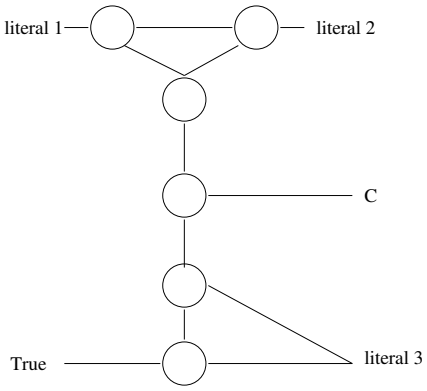


Figure 1: The Clause Gadget

We claim that this clause gadget can be 3 colored if and only if at least one of literal 1, literal 2 literal 3 is colored true. Hence, it follows that G is 3-colorable if and only if F is satisfiable. The final graph G for the above formula is shown below.

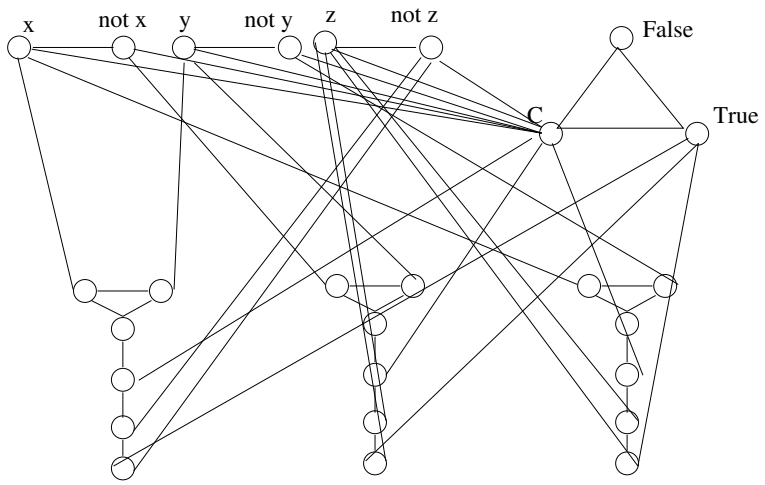


Figure 2: The final graph