1 NP Completeness

1.1 Encodings

An encoding is a mapping from abstract objects to character strings over some finite alphabet.

A **concrete problem** is a problem whose instances are sets of binary strings.

An algorithm **solves** a concrete problem in time $O(T(n))$ if, given an input of length n, produces the (encoding of the) solution in at most $O(T(n))$ time.

1.2 Complexity Classes

Polynomial time solvability: $O(n^k)$ some k.

Complexity class P: decision problems solvable in polynomial time.

This definition is independent of encoding as long as one only considers encodings that are **polynomially related**.

Unary-binary as non-polynomially related encodings.

1.3 Formal language framework

- Alphabets (finite)
- Languages over an alphabet Σ
- A decision problem as the set of strings with a yes answer
- An algorithm accepts (rejects) a string (or loops)
- A language is *decided* by an algorithm
- Acceptance in polynomial time
- Decidability in polynomial time
- Definition of complexity class

1.4 Polynomial time verification

Checking (verifying) versus solving

Example: Existence of a path between vertices s and t

Verification algorithm $A(x, y)$

Certificates y

Example – a path between two vertices verifies that they are connected. Thus x could be the connectedness question and y could be a path that verifies that two vertices are connected.

Language verified by A

1.5 The class NP

NP is the set of languages L such that there is a polynomial time verification algorithm A such that

$$x \in L \text{ iff there exists "short" } y : A(x, y) = 1$$

We say A verifies L in polynomial time.

Example – a coloring y verifies that a graph x can be colored with 4 colors. But it may be hard to find such a coloring y.

1.6 The P = NP question

P $\subseteq$ NP: A can ignore y

Closure under complementation is also unknown – would imply that there is always a verification of non-solutions i.e. verification of non-4 colorability for problems in NP

1.7 NP-Completeness

NP-complete problems: If any one of them has a polynomial time solution then all do, and P = NP. But no one has found such a solution, nor has anyone proven that it does not exist. Many problems of practical importance are NP-complete.

These are in a sense the hardest problems in NP.

1.8 Examples

- Graph isomorphism is in NP (but maybe not NP complete); a verification is an isomorphism
- The hamiltonian cycle problem is in NP (and NP complete); a verification is a cycle
- The tautology problem is in co-NP.

In practice when trying to find an algorithm for a problem, one either finds a polynomial time algorithm, or shows the problem to be NP complete; sometimes neither is possible. It is important to be able to show a problem NP complete because it means that one is very unlikely to find a polynomial time algorithm for it.

1.9 Reducibility

A language L_1 is **polynomial-time reducible** to L_2 ($L_1 \leq_P L_2$) if there is a polynomial time computable function f from languages to languages such that for all x, $x \in L_1$ iff $f(x) \in L_2$.

If L_1 is polynomial time reducible to L_2, then an algorithm to solve L_2 can be used to obtain an algorithm to solve L_1. Also, the time required will be the same to within a polynomial. That is, L_1 cannot be much harder than L_2.

Lemma 36.3 If L_1 and L_2 are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Proof.

1.10 NP-completeness

A language L is **NP-compete** if

1. $L \in NP$, and

2. $L' \leq_P L$ for every $L' \in NP$

Thus every other language in NP is polynomial reducible to L. We say L is **NP-hard** if it satisfies property 2.

Theorem 36.4 If any NP complete problem is in P, then P = NP. If any NP complete problem is not in P, then no NP complete problem is in P, and P $\neq$ NP.

Thus the whole P = NP question boils down to the complexity of a single NP-complete problem (any one of them).

1.11 Cooks's Proof

Cook [1971] found the first NP-complete problem and became famous for it. He showed that the satisfiability problem was NP complete. The book shows a slightly different problem to be NP-complete, that is, **circuit satisfiability**.

A circuit has **AND**, **OR**, **NOT**, and possibly other gates, with outputs of some gates connected to inputs of others in a directed acyclic graph (no cycles). The circuit satisfiability problem is to determine whether there is an input to the circuit leading to an output of 1 (true). The book shows this to be NP-complete using a slightly informal argument.

Review the argument.

Another way to do this is to construct a Boolean formula which is satisfiable iff a nondeterministic Turing machine accepts an input in polynomial time.

For this the variables are: $C(i, j, t)$, $1 \leq i \leq p(n)$, $1 \leq j \leq m$, $0 \leq t \leq p(n)$.

The interpretation is that $C(i, j, t) = 1$ (true) iff the contents of ith tape cell of M is X_j at time t.

$S(k, t)$, $1 \leq k \leq s$, $0 \leq t \leq p(n)$.

The interpretation is that $S(k, t) = 1$ (true) iff M is in state q_k at time t.

$H(i, t)$, $1 \leq i \leq p(n)$, $0 \leq t \leq p(n)$.

The interpretation is that $H(i, t) = 1$ (true) iff the tape head points to cell i at time t.

The Boolean formula expresses the constraints on a nondeterministic Turing machine computation and also states that the computation does not reject the input. This formula is satisfiable iff there is a sequence of configurations leading to acceptance in polynomial time.

1.12 Showing Other Problems NP Complete

Once we have one NP-complete problem we can obtain more using the following lemma:

Lemma 36.8 If L' is NP-complete and $L' \leq_P L$, then L is NP-hard. If L is also in NP, then L is NP-complete.

Proof.

Five-step recipe for showing NP-completeness of L:

1. Prove $L \in NP$

2. Select NP-complete L'

3. Find a reduction f from L' to L

4. Prove $x \in L'$ iff $f(x) \in L$

5. Prove that f is polynomial time computable

Other problems shown to be NP-complete

- Formula satisfiability

  Example: A formula is $(p \supset q) \supset ((\neg q) \supset (\neg p))$ This is a tautology; whether you replace p and q by $true$ or $false$ the formula evaluates to true. The tautology problem is to determine if a formula is a tautology. The tautology problem is actually co-NP complete.

  Example: A formula is $(p \wedge q) \vee p$. This formula is satisfiable because if you replace p by $true$ and q by anything, the formula evaluates to $true$. The formula satisfiability problem is to determine if a Boolean formula is satisfiable. This problem is NP-complete.

- 3-CNF satisfiability

  A formula is in 3-CNF if it is a disjunction of conjunctions of three literals, where a literal is a Boolean variable or its negation. So an example is the formula $(p \vee q \vee \neg r) \wedge (\neg p \vee \neg r \vee s)$. The 3-CNF satisfiability problem is to determine whether a formula in 3-CNF is satisfiable. This problem is NP-complete.

- The clique problem
- The vertex cover problem (minimize size of cover)
- The subset sum problem
- The hamiltonian cycle problem
- The traveling salesman problem
- Graph coloring

and many, many more ...

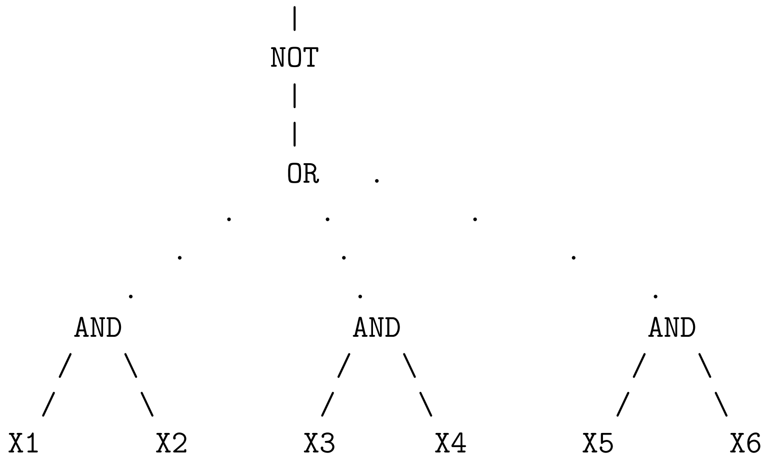
In fact, some journals no longer accept NP completeness proofs, since there are so many of them and they are now becoming routine and uninteresting – of course, it still helps a lot to know how hard a problem is!

Many of the reductions are uninteresting, too.

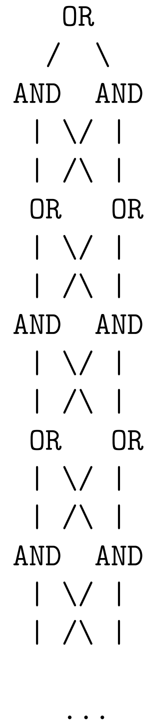
Proof that formula satisfiability is NP complete (follow the 5 step recipe). Reduce from circuit satisfiability.

Idea for avoiding duplication of subformulas

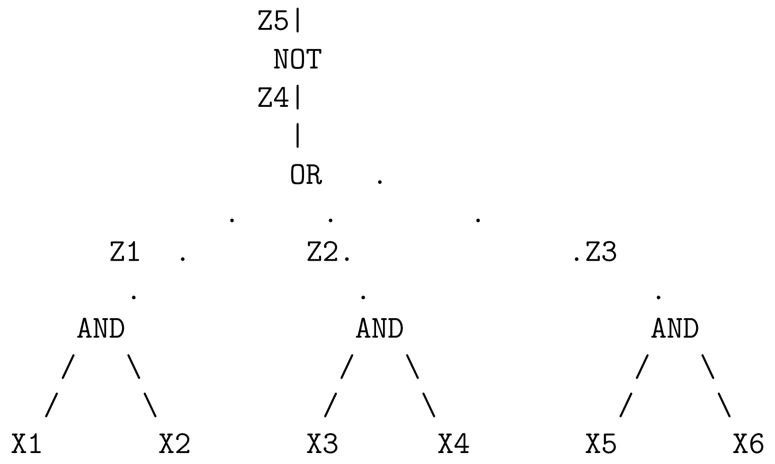
Suppose we have the circuit



The reduction to NOT(OR(AND(X1, X2), AND(X3, X4), AND(X5, X6))) won't always work because we can have exponential size increase due to duplicated subformulas, making the reduction non polynomial time, as in this example:



So we have to add new variables for the wires, as follows:



Then we create formulas relating the values of the variables, as follows:

$$Z1 \equiv AND(X1, X2)$$

$$Z2 \equiv AND(X3, X4)$$

$$Z3 \equiv AND(X5, X6)$$

$$Z4 \equiv OR(Z1, Z2, Z3)$$

$$Z5 \equiv NOT(Z4)$$

We then concatenate them all together, with Z5, as follows:
$(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6)) \wedge (Z4 \equiv OR(Z1, Z2, Z3)) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$

Note that Z5 is added on, too. In this way, a circuit $\mathcal{C}$ is mapped on to a formula $\mathcal{F}$. This is the reduction. We need to show it is polynomial time (easy), and preserves solvability. (Also not difficult.)

We now show that 3 CNF is NP complete by reducing from formula satisfiability. An example of a formula in 3 CNF is the following:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_2 \vee x_1)$$

The general steps in the reduction are as follows:

- All Boolean connectives as binary
- New variables for subformulas
- Reduce to conjunctive normal form
- Add variables to clauses that are too small

We illustrate these steps on the formula obtained above: $(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6)) \wedge (Z4 \equiv OR(Z1, Z2, Z3)) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$

First we make all connectives binary:
$(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6))$
$\wedge$
$(Z4 \equiv OR(Z1, OR(Z2, Z3))) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$

Then we add new variables for subformulas so that the number of variables in each conjunct is 3 or less. For our formula that is already true for each conjunct except one, so we obtain this formula:

$(Z1 \equiv AND(X1, X2)) \wedge (Z2 \equiv AND(X3, X4)) \wedge (Z3 \equiv AND(X5, X6)) \wedge (Z4 \equiv OR(Z1, Y1)) \wedge (Y1 \equiv OR(Z2, Z3)) \wedge (Z5 \equiv NOT(Z4)) \wedge Z5$

Next we put each subformula in conjunctive normal form, obtaining:

$(Z1 \supset AND(X1, X2)) \wedge (Z1 \subset AND(X1, X2)) \wedge (Z2 \supset AND(X3, X4)) \wedge (Z2 \subset AND(X3, X4)) \wedge (Z3 \supset AND(X5, X6)) \wedge (Z3 \subset AND(X5, X6)) \wedge (Z4 \supset OR(Z1, Y1)) \wedge (Z4 \subset OR(Z1, Y1)) \wedge (Y1 \supset OR(Z2, Z3)) \wedge (Y1 \subset OR(Z2, Z3)) \wedge (Z5 \supset NOT(Z4)) \wedge (Z5 \subset NOT(Z4)) \wedge Z5$

Continuing, we obtain:

$(\neg Z1 \vee X1) \wedge (\neg Z1 \vee X2) \wedge (\neg X1 \vee \neg X2 \vee Z1) \wedge (\neg Z2 \vee X3) \wedge (\neg Z2 \vee X4) \wedge (\neg X3 \vee \neg X4 \vee Z2) \wedge \ldots \wedge Z5$

The problem now is that some of the disjunctions have too few variables. So we add extra variables to them; for example, we replace $(\neg Z1 \vee X1)$ by $(\neg Z1 \vee X1 \vee W) \wedge (\neg Z1 \vee X1 \vee \neg W)$ for a new variable W. This has to be done twice if we start with just one variable.

After all these steps, we obtain a 3 CNF formula that is satisfiable iff the original formula was satisfiable. Also, the transformation is polynomial time.

Proof that the clique problem is NP complete:

Reduce from 3 CNF; suppose there are k clauses $C_1, \ldots, C_k$. Create a graph G for the clique problem, as follows:

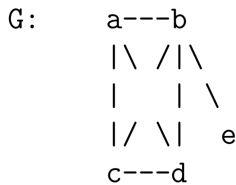
- Three vertices for each clause, one for each literal
- Edges between vertices in different clauses if the literals are not complementary
- A k clique exists iff S is satisfiable

Proof that the vertex cover problem is NP complete.

Reduce from the clique problem, using graph complement.

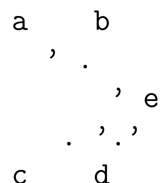
Suppose G has a clique of size k. Let $\overline{G}$ be the complementary graph, with an edge between (u, v) iff G does not have this edge. Then G has a clique of size k iff $\overline{G}$ has a vertex cover of size $V - k$.

Example of vertex cover reduction:



G has a clique of size 4

G complement:



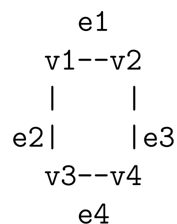
G complement has a vertex cover of size 1

Proof that the subset sum problem is NP complete:

Reduce from the vertex cover problem.

Create base 4 numbers, with a digit position for each edge in the graph. Each vertex has ones for digits of its incident edges. Each edge has a one in its own position. Then we want to sum so that there is a 2 in every position. Thus every position has to appear twice. This cannot come only from the edge digits, so there must be an incident vertex. But if there is only one incident vertex, then the edge has to be included, too; otherwise it is not. Also, the total number of vertices is controlled by a significant digit.

Example of the subset sum construction:



This has a vertex cover of {v1,v4}. The subset sum instance:

```
 4^4    4^2    4^0
    4^3    4^1
```

		e4	e3	e2	e1	Value
v1:	1	0	0	1	1	4^4 + 4^1 + 4^0
v2:	1	0	1	0	1	4^4 + 4^2 + 4^0

```
v3:  1  1   0   1   0    4^4 + 4^3 + 4^1
v4:  1  1   1   0   0    4^4 + 4^3 + 4^2
e1:  0  0   0   0   1    4^0
e2:  0  0   0   1   0    4^1
e3:  0  0   1   0   0    4^2
e4:  0  1   0   0   0    4^3
---------------------
    (2) 2   2   2   2    2*4^4 + 2*4^3
                         + 2*4^2 + 2*4^1
                         + 2*4^0
```

So we obtain the following subset sum problem: Is there a subset of the integers $\{4^4+4^1+4^0, 4^4+4^2+4^0, 4^4+4^3+4^1, 4^4+4^3+4^2, 4^0, 4^1, 4^2, 4^3\}$ that adds to $2*4^4+2*4^3+2*4^2+2*4^1+2*4^0$? Or, is there a subset of the integers $\{261, 273, 324, 336, 1, 4, 16, 64\}$ that sums to 682? And the answer is yes, the subset corresponding to $\{v1, v4, e4, e3, e2, e1\}$, that is, $\{261, 336, 1, 4, 16, 64\}$, and one can verify that $261+336+1+4+16+64=682$. We add enough edges (one or two in each column) to insure that each column has two ones altogether.

NP completeness of the Hamiltonian circuit problem:

Very involved.

The idea is that one constructs A-widgets, that permit exactly one of two edges to be traversed, and B widgets, that permit any proper subset of three edges (but not all of them) to be traversed in a Hamiltonian cycle. Using these, we can convert a 3 satisfiability problem into a graph that has a Hamiltonian cycle iff the set of clauses is satisfiable.

NP completeness of the traveling salesman problem:

Reduce from the Hamiltonian circuit problem. Convert a graph G to a graph G' with edge costs $c(i,j)$. G' is a complete graph. Let the cost be zero if edge (i,j) is in G, one otherwise. Then it is easily seen that G' has a traveling salesman tour of cost 0 iff G has a Hamiltonian circuit.

1.13 The Turing Machine Approach

The class NP is often defined in terms of nondeterministic Turing machines instead of polynomial time verifiers, so you should be familiar with this approach, too. A Turing machine has an infinite tape on which the input is

written. It has a finite set of states, and can read, write, and move to the left or right, while changing state. It can also be nondeterministic.

The class NP is defined as the class of problems that can be accepted by nondeterministic Turing machines in polynomial time. That is, there is *some* sequence of transitions that leads to an accepting state. One can show that this is equivalent to the verifier definition given in the text.

1.14 Quantum Computation

This permits an exponential number of states and all states are processed in parallel. For example, if there are 10 electrons and each are in 2 states simultaneously, then the whole system has 2^{10} states. It is known that integers can be factored in polynomial time on quantum computers (which have never been built), but this is not known for conventional computers. It could even be that problems in NP are solvable in polynomial time on quantum computers, though this hasn't been shown. If this were true, it might make the P = NP question of secondary importance.

1.15 Approximation Algorithms

Even if a problem is NP complete, we can find solutions that are close to optimal in polynomial time in many cases. Sometimes one can prove that even this is impossible if P $\neq$ NP.

One can bound the **ratio** of the cost of a solution to the cost of an optimal solution, or the **relative error**, as a function of n, the size of the input. One looks for an approximation algorithm that runs in polynomial time and has a specified ratio, for example.

A **polynomial-time approximation scheme** is an algorithm that accepts as an input a problem instance and a relative error ϵ and runs in polynomial time and produces a solution within ϵ of optimal. (May not exist.)

A **fully polynomial-time approximation scheme** has a running time that is polynomial both in n and in $1/\epsilon$. This guarantees that one can improve the accuracy of the approximation without too much extra work. (May not exist, too.)

The Vertex Cover Problem

For this we can obtain a ratio of two to an optimal solution in polynomial time by a very simple algorithm.

```
APPROX-VERTEX-COVER(G)
    C <- 0
    E' <- E[G]
    while E' .ne. 0
         do let (u,v) be an arbitrary edge of E'
            C' <- C U {u,v}
            remove from E' every edge incident
                   on either u or v
    return C
```

Example.

Proof of approximation.

The Traveling Salesman Problem

We obtain a ratio of two to an optimal solution in polynomial time assuming the **triangle inequality**:

$$c(u, w) \leq c(u, v) + c(v, w).$$

The approximation algorithm makes use of minimum spanning trees.

```
APPROX-TSP-TOUR(G,c)
    select a vertex r in V[G] as a root
    grow a MST T for G from r using
        Prim's algorithm
    Let L be the list of vertices of T
        visited in preorder
    return the Hamiltonian cycle that
        visits the vertices in the order L
```

Example and proof of approximation.

Without the triangle inequality, no polynomial time approximation algorithm is possible unless P = NP:

The idea is to take a graph G and produce another graph G' in which the costs of the edges in G are 1 but the costs of edges not in G are very large. Then an approximation algorithm for the TSP G' can be used to test if G has a Hamiltonian circuit.

Example.

The Set Covering Problem

Here we obtain an approximation algorithm with a logarithmic bound.

An instance $(X, \mathcal{F})$ of the set covering problem consists of a finite set X and a family $\mathcal{F}$ of subsets of X such that every element of X belongs to at least one subset in $\mathcal{F}$. We want to find a minimum size subset $\{S_1, S_2, \ldots, S_k\}$ of elements of $\mathcal{F}$ such that $S_1 \cup S_2 \cup \ldots S_k = X$. That is, k should be as small as possible.

The approximation algorithm is greedy:

```
GREEDY-SET-COVER(X,F)
    U <- X
    C <- 0
    while U .ne. 0
        do select an S in F that
           maximizes |S /\ U|
           U <- U - S
           C <- C U {S}
    return(C)
```

The ratio obtained is $H(max\{|S| : S \in \mathcal{F}\})$ where $H(d) = 1 + 1/2 + 1/3 + \ldots + 1/d$.

Example.

Proof of ratio.

The Subset Sum Problem

This is interesting because a fully polynomial approximation scheme is known. For this we want to find a subset of a set S of integers which sums to t (or as close as possible to t), where S and t are given as input. However, it is not permitted for the sum to exceed t.

First we give an exact (exponential) algorithm:

```
EXACT-SUBSET-SUM(S,t)
    n <- |S|
    L_0 <- <0>
    for i <- 1 to n
        do L_i <-
               MERGE-LISTS(L_(i-1),L_(i-1)+x_i)
```

```
        remove from L_i all elements
                            larger than t
    return the largest element in L_n
```

This returns a sum as large as possible but not larger than t.

The idea of the approximation algorithm is to remove from the lists elements that are close to one another in value:

```
APPROX-SUBSET-SUM(S,t,epsilon)
    n <- |S|
    L_0 <- <0>
    for i <- 1 to n
        do L_i <-
               MERGE-LISTS(L_(i-1),L_(i-1)+x_i)
           L_i <- Trim(L_i,epsilon/n)
           remove from L_i all elements
                           larger than t
    return the largest element in L_n
```

Here we use TRIM defined as follows:

```
TRIM(L, delta)
    m <- |L|
    L' <- <y_1>
    last <- y_1
    for i <- 2 to m
         do if last < (1 - delta)y_i
               then append y_i onto
                       the end of L'
                    last <- y_i
    return L'
```

Example.

Proof of approximation.