NP-Completeness

P: is the set of decision problems (or languages) that are solvable in polynomial time.

NP: is the set of decision problems (or languages) that can be verified in polynomial time.

Polynomial reduction: $L1 \prec_P L2$ means that there is a polynomial time computable function f such that $x \in L1$ if and only if $f(x) \in L2$. A more intuitive to think about this, is that if we had a subroutine to solve $L2$ in polynomial time, then we could use it to solve $L1$ in polynomial time

$$\text{Lemma: If } L1 \prec_P L2 \text{ and } L2 \in \mathrm{P} \text{ then } L1 \in \mathrm{P}.$$
$$\text{Lemma: If } L1 \prec_P L2 \text{ and } L1 \notin \mathrm{P} \text{ then } L2 \notin \mathrm{P}.$$

• Polynomial reductions are transitive, that is, if $L1 \prec_P L2$ and $L2 \prec_P L3$, then $L1 \prec_P L3$.

NPHard: L is **NPhard** if for all $L' \in$ **NP**, $L' \prec_P L$. Thus, if we could solve L in polynomial time, we could solve all **NP** problems in polynomial time.

NPComplete: L is **NPcomplete** if (1) $L \in$ **NP** and (2) L is **NPhard**.

• The importance of **NPcomplete** problems should now be clear. If any **NPcomplete** problem (and generally any **NPhard** problem) is solvable in polynomial time, then every **NPcomplete** problem (and in fact every problem in **NP**) is also solvable in polynomial time.

• Conversely, if we can prove that any **NPcomplete** problem cannot be solved in polynomial time, then every **NPcomplete** problem (and generally every **NPhard** problem) cannot be solved in polynomial time.

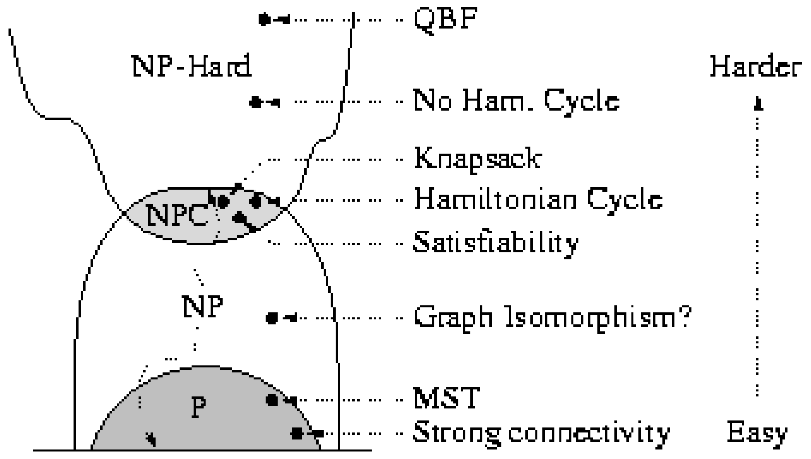
• Thus all **NPcomplete** problems are equivalent to one another (in that they are either all solvable in polynomial time, or none are).

NP-Completeness (cont.)

• The figure below illustrates one way that the sets **P**, **NP**, **NPhard**, and **NP-complete** (**NPC**) *might* look. We say *might* because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time.

• One is **Graph Isomorphism**, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in **NP**, but it is not known to be in **P**.

• The other is **QBF**, which stands for **Quantified Boolean Formulas**. In this problem you are given a Boolean formula with quantifiers ($\exists$ and $\forall$) and you want to know whether the formula is true or false.



One way that things 'might' be.

Structure of P, NP, and related complexity classes.

• An alternative way to show that a problem is **NPC** is to use transitivity of π_P .

Lemma: L is **NPcomplete** if (1) $L \in$ **NP** and (2) $L' \prec_P L$ for some **NPcomplete** language L'.

Note: The *known* **NPcomplete** problem L' is being reduced to *candidate* **NP-complete** problem L. Keep this order in mind.

Cook-Levin's Theorem and Reductions

• Unfortunately, to use this lemma, we need to have *at least one* **NPcomplete** problem to start the ball rolling. Stephen Cook and Leonid Levin showed (~ 1970) that such a problem existed. Cook-Levin's theorem is rather complicated to prove. First we'll try to give a brief intuitive argument as to why such a problem might exist.

• For a problem to be in **NP**, it must have an efficient verification procedure.

• Virtually all **NP** problems can be stated in the form, ``does there exists *X* such that *P(X)*'', where *X* is some structure (e.g. a set, a path, a partition, an assignment, etc.) and *P(X)* is some property that *X* must satisfy (e.g. the set of objects must fill the knapsack, or the path must visit every vertex, or you may use at most *k* colors and no two adjacent vertices can have the same color).

• In showing that such a problem is in **NP**, the certificate consists of giving *X*, and the verification involves testing that *P(X)* holds.

• In general, any set *X* can be described by choosing a set of objects, which in turn can be described as choosing the values of some Boolean variables.

• Similarly, the property *P(X)* that you need to satisfy, can be described as a Boolean formula.

• Cook and Levin were looking for the *most* general possible property he could, since this should represent the *hardest* problem in **NP** to solve.

• They reasoned that computers (which represent the most general type of computational devices known) could be described entirely in terms of Boolean circuits, and hence in terms of Boolean formulas.

• If any problem were hard to solve, it would be one in which *X* is an assignment of Boolean values (true/false, 0/1) and *P(X)* could be any Boolean formula. This suggests the following problem, called the ***Boolean satisfiability problem***.

Boolean Satisfiability Problem

SAT: Given a Boolean formula, is there some way to assign truth values (0/1, true/false) to the variables of the formula, so that the formula evaluates to true?

• A *Boolean formula* is a logical formula which consists of variables x_i, and the logical operations $\overline{x}$ meaning the *negation* of *x*, *Boolean or* $(x \vee y)$ and *Boolean and* $(x \wedge y)$.

• Given a Boolean formula, we say that it is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1. (As opposed to the case where no matter how you assign truth values the result is always 0.)

For example, $(x_1 \wedge (x_2 \vee \overline{x}_3)) \wedge ((\overline{x}_2 \wedge \overline{x}_3) \vee \overline{x}_1)$

is satisfiable, by the assignment $x_1 = 1, x_2 = 0, x_3 = 0.$ On the other hand,

$$(\overline{x}_1 \vee (x_2 \wedge x_3)) \wedge (x_1 \vee (\overline{x}_2 \wedge \overline{x}_3)) \wedge (x_2 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

is not satisfiable. (Observe that the last two clauses imply that one of x_2 and x_3 must be true and the other must be false. This implies that neither of the subclauses involving x_2 and x_3 in the first two clauses can be satisfied, but x_1 cannot be set to satisfy them either.)

Cook-Levin's Theorem: *SAT* is **NP-complete**.

We will not prove this theorem. The proof would take about a full lecture.

• The proof and many other interesting results on complexities are given in the ***Theory of Computation II*** course.

3Conjunctive Normal Form (3CNF)

• In fact, it turns out that a even more restricted version of the satisfiability problem is **NPcomplete**.

• A *literal* is a variable or its negation, x or $\overline{x}$.

• A formula is in *3conjunctive normal form* (*3CNF*) if it is the Booleanand of clauses where each clause is the Booleanor of exactly 3 literals.

• For example $(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \overline{x}_3 \vee \overline{x}_4)$ is in 3CNF form.

3SAT is the problem of determining whether a formula in 3CNF is satisfiable.

• It turns out that it is possible to modify the proof of Cook's theorem to show that ***3SAT*** is also **NPcomplete**.

• As an aside, note that if we replace the 3 in ***3SAT*** with a 2, then everything changes. If a Boolean formula is given in ***2SAT***, then it is possible to determine its satisfiability in polynomial time. (It turns out that the problem can be reduced to computing the strong components in a directed graph.)

• Thus, even a seemingly small change can be the difference between an efficient algorithm and none.

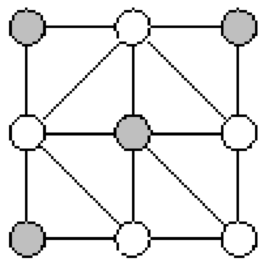
NPcompleteness proofs

Now that we know that ***3SAT*** is **NPcomplete**, we can use this fact to prove that other problems are **NPcomplete**. We will start with the ***independent set problem.***

Independent Set (IS) Problem

• ***Independent Set (IS):*** Given an undirected graph $G = (V, E)$ and an integer k does G contain a subset V' of k vertices such that no two vertices in V' are adjacent to one another.

• For example, the graph shown in the figure below has an independent set (shown with shaded nodes) of size 4.



• The independent set problem arises when there is some sort of selection problem, but there are mutual restrictions pairs that cannot both be selected. (For example, a company party where an employee and his/her immediate supervisor cannot both be invited.)

• Note that if a graph has an independent set of size k, then it has an independent set of all smaller sizes. So the corresponding optimization problem would be to find an independent set of the largest size in a graph.

• Often the vertices have weights, so we might talk about the problem of computing the independent set with the largest total weight.

• However, since we want to show that the problem is hard to solve, we will consider the simplest version of the problem.

Claim: IS is **NPcomplete**.

• The proof involves two parts. First, we need to show that ***IS***$\in$ **NP**. The certificate consists of the k vertices of V' . We simply verify that for each pair of vertex $u,v \in V'$, there is no edge between them. Clearly this can be done in polynomial time, by an inspection of the adjacency matrix.

Independent Set (reduction)

• Secondly, we need to establish that ***IS*** is **NPhard**, which can be done by showing that some known **NPcomplete** problem (***3SAT***) is polynomialtime reducible to ***IS***, that is, ***3SAT*** $\prec_P$ ***IS.***

• Let *F* be a boolean formula in *3CNF* form. We wish to find a polynomial time computable function *f* that maps *F* into an input for the ***IS*** problem, a graph *G* and integer *k*. That is, *f(F) = (G, k),* such that *F* is satisfiable if and only if *G* has an independent set of size *k*.

• This will mean that if we can solve the independent set problem for *G* and *k* in polynomial time, then we would be able to solve ***3SAT*** in polynomial time.

• An important aspect to reductions is that we do not attempt to solve the satisfiability problem. (It is **NPcomplete**, and there is not likely to be any polynomial time solution.)

• So the function *f* must operate without knowledge of whether *F* is satisfiable. The idea is to *translate* the similar elements of the satisfiable problem to corresponding elements of the independent set problem.

What is to be selected?

3SAT: Which variables are to be assigned the value true, or equivalently, which literals will be true and which will be false.

IS: Which vertices will be placed in *V'*.

Requirements:

3SAT: Each clause must contain at least one true valued literal.

IS: *V'* must contain at least *k* vertices.

Restrictions:

3SAT: If x is assigned true, then $\overline{x}$ must be false, and vice versa.

IS: If *u* is selected to be in *V'*, and *v* is a neighbor of *u*, then *v* cannot be in *V'*.

Independent Set (reduction: cont.)

- We want a function *f* , which given any *3CNF* boolean formula *F* , converts it into a pair *(G, k)* such that the above elements are translated properly.
- Our strategy will be to turn each literal into a vertex. The vertices will be in *clause clusters* of three, one for each clause.
- Selecting a true literal from some clause will correspond to selecting a vertex to add to *V'* . We will set *k* equal to the number of clauses, to force the independent set subroutine to select one true literal from each clause.
- To keep the ***IS*** subroutine from selecting two literals from one clause and none from some other, we will connect all the vertices in each clause cluster with edges.
- To keep the ***IS*** subroutine from selecting a literal and its complement to be true, we will put an edge between each literal and its complement.
- A formal description of the reduction is given below. The input is a boolean formula *F* in *3-CNF*, and the output is a graph *G* and integer *k*.

```
$k \leftarrow$ number of clauses in $F$;
for each clause $C$ in $F$ {
    create a clause cluster of 3 vertices from the literals of $C$;
}
for each clause cluster $(x_1, x_2, x_3)$ {
    create an edge $(x_i, x_j)$ between all pairs of vertices in the cluster;
}
for each vertex $x_i$ {
    create edges between $x_i$ and all its complement vertices $\overline{x}_i$;
}
return $(G, k)$;
```

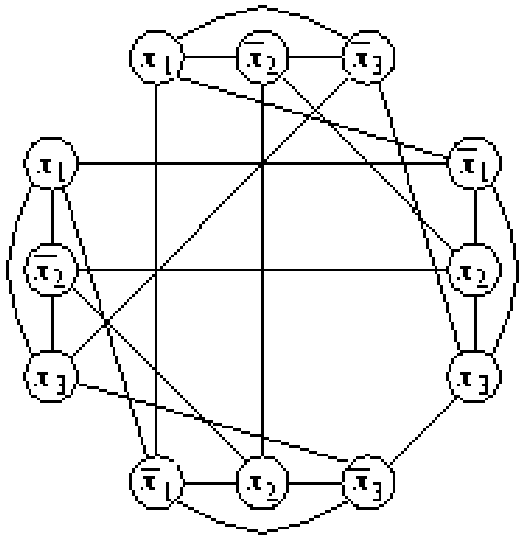
- If *F* has *k* clauses, then *G* has exactly *3k* vertices.
- Given any reasonable encoding of *F* , it is an easy programming exercise to create *G* (say as an adjacency matrix) in polynomial time.
- ***We claim*** that *F* is satisfiable if and only if *G* has an independent set of size *k*.

Example

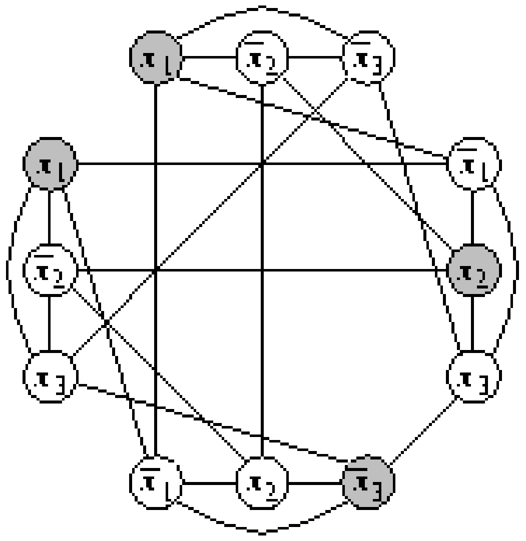
- Suppose that we are given the *3CNF* formula:

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3).$$

- The reduction produces the graph shown in the following figure and sets *k = 4*.
- In our example, the formula is satisfied by the assignment $x_1 = 1, x_2 = 1, x_3 = 0.$
- Note that this implies that the first literal of the first and last clauses are 1, the second literal of the second clause is 1, and the third literal of the third clause is 1.
- Observe that by selecting the corresponding vertices from the clusters, we get an independent set of size *k = 4*.



The reduction



Correctness (x1=x2=1, x3=0)

Correctness Proof

- ***We claim*** that *F* is satisfiable if and only if *G* has an independent set of size *k*.
- If *F* is satisfiable, then each of the *k* clauses of *F* must have at least one true literal.
- Let *V'* denote the corresponding vertices from each of the clause clusters (one from each cluster).
- Because we take vertices from each cluster, there are no intercluster edges between them, and because we cannot set a variable and its complement to both be true, there can be no edge of the form ($x_i, \overline{x}_i$) between the vertices of *V'*. Thus, *V'* is an independent set of size *k*.
- Conversely, if *G* has an independent set *V'* of size *k*. First observe that we must select a vertex from each clause cluster, because there are *k* clusters, and we cannot take two vertices from the same cluster (because they are all interconnected).
- Consider the assignment in which we set all of these literals to 1. This assignment is logically consistent, because we cannot have two vertices labeled x_i and $\overline{x}_i$ in the same cluster.
- Finally the transformation clearly runs in polynomial time. This completes the NP-completeness proof.
- Observe that our reduction did not attempt to solve the ***IS*** problem nor to solve the ***3SAT***.
- Also observe that the reduction had *no knowledge* of the solution to either problem. (We did not assume that the formula was satisfiable, nor did we assume we knew which variables to set to 1.) This is because computing these things would require exponential time (by the best known algorithms).
- Instead the reduction simply *translated* the input from one problem into an equivalent input to the other problem, while preserving the critical elements to each problem.

Clique and Vertex Cover Problems

• Now we give a few more examples of reductions.

• Recall that to show that a problem is **NPcomplete** we need to show (1) that the problem is in **NP** (i.e. we can verify when an input is in the language), and (2) that the problem is **NP-hard**, by showing that some known **NPcomplete** problem can be reduced to this problem (there is a polynomial time function that transforms an input for one problem into an equivalent input for the other problem).

Some Easy Reductions: We consider some closely related **NPcomplete** problems next.

Clique (CLIQUE): The clique problem is: given an undirected graph $G = (V, E)$ and an integer k, does G have a subset V' of k vertices such that for each distinct $u,v \in V'$, $\{u,v\} \in E$. In other words, does G have a k vertex subset whose induced subgraph is complete.

Vertex Cover (VC): A vertex cover in an undirected graph $G = (V, E)$ is a subset of vertices

$V' \subseteq V$ such that every edge in G has at least one endpoint in V' . The vertex cover problem (***VC***) is: given an undirected graph G and an integer k, does G have a vertex cover of size k?

• Don't confuse the clique (***CLIQUE***) problem with the cliquecover (***CCov***) problem that we discussed in an earlier lecture. The clique problem seeks to find a single clique of size k, and the cliquecover problem seeks to partition the vertices into k groups, each of which is a clique.

• We have discussed the facts that cliques are of interest in applications dealing with clustering.

• The vertex cover problem arises in various servicing applications. For example, you have a computer network and a program that checks the integrity of the communication links. To save the space of installing the program on every computer in the network, it suffices to install it on all the computers forming a vertex cover. From these nodes all the links can be tested.

Clique and Vertex Cover Problems (Reductions)

• The ***CLIQUE*** problem is obviously closely related to the independent set problem (***IS***): Given a graph G does it have a k vertex subset that is completely disconnected.

• It is not quite as clear that the vertex cover problem is related. However, the following lemma makes this connection clear as well.

Lemma: Given an undirected graph $G = (V, E)$ with n vertices and a subset $V' \subseteq V$ of size k. The following are equivalent:

(i) V' is a clique of size k for the complement, $\overline{G}$.

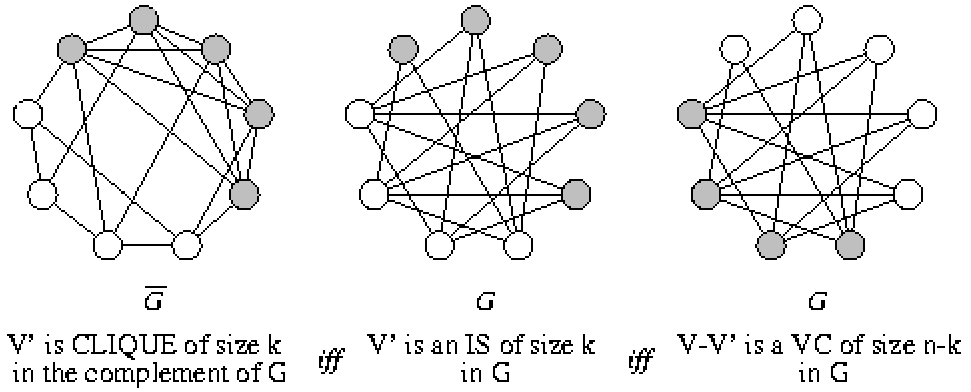
(ii) V' is an independent set of size k for G.

(iii) $V \setminus V'$ is a vertex cover of size n-k for G.

Proof: (i)➔(ii): If V' is a clique for $\overline{G}$, then for each u,v in V', $\{u,v\}$ is an edge of $\overline{G}$ implying that $\{u,v\}$ is not an edge of G, implying that V' is an independent set for G.

(ii) ➔(iii): If V' is an independent set for G, then for each $u,v \in V'$, $\{u,v\}$ is not an edge of G, implying that every edge in G is incident to a vertex in $V \setminus V'$, implying that $V \setminus V'$ is a ***VC*** for G.

(iii) ➔(i): If $V \setminus V'$ is a ***VC*** for G, then for any u,v in V' there is no edge $\{u,v\}$ in G, implying that there is an edge $\{u,v\}$ in $\overline{G}$, implying that V' is a clique in $\overline{G}$. V' is an independent set for G.


$\overline{G}$
V' is CLIQUE of size k in the complement of G
iff
G
V' is an IS of size k in G
iff
G
V-V' is a VC of size n-k in G

Clique and Vertex Cover (NP-completeness)

• Thus, if we had an algorithm for solving any one of these problems, we could easily translate it into an algorithm for the others. In particular, we have the following.

Theorem: CLIQUE is **NPcomplete**.

CLIQUE $\in$ **NP**: The certificate consists of the *k* vertices in the clique. Given such a certificate we can easily verify in polynomial time that all pairs of vertices in the set are adjacent.

IS $\prec_P$ ***CLIQUE:*** We want to show that given an instance of the ***IS*** problem *(G, k),* we can produce an equivalent instance of the ***CLIQUE*** problem *(G',k')* in polynomial time.

• (Important: We do not know whether *G* has an independent set, and we do not have time to compute it.)

• Given *G* and *k*, set $G' = \overline{G}$ and *k'*= *k*, and output the pair *(G', k')*. By the above lemma, this instance is equivalent.

Theorem: VC is **NPcomplete**.

VC $\in$ **NP**: The certificate consists of the *k* vertices in the vertex cover. Given such a certificate we can easily verify in polynomial time that every edge is incident to one of these vertices.

IS $\prec_P$ ***VC***: We want to show that given an instance of the ***IS*** problem *(G, k),* we can produce an equivalent instance of the ***VC*** problem *(G', k')* in polynomial time. We set *G'*= *G* and *k'* = *n* – *k*. By the above lemma, these instances are equivalent.