Polynomial-Time Reducibility

Recall that reducibility was an important concept for computability theory. It provided a useful tool for relating the decidability or recognizability of two languages.

It also plays an important role in complexity theory, relating the complexity of deciding two languages.

Suppose that A and B are languages over an alphabet Σ.

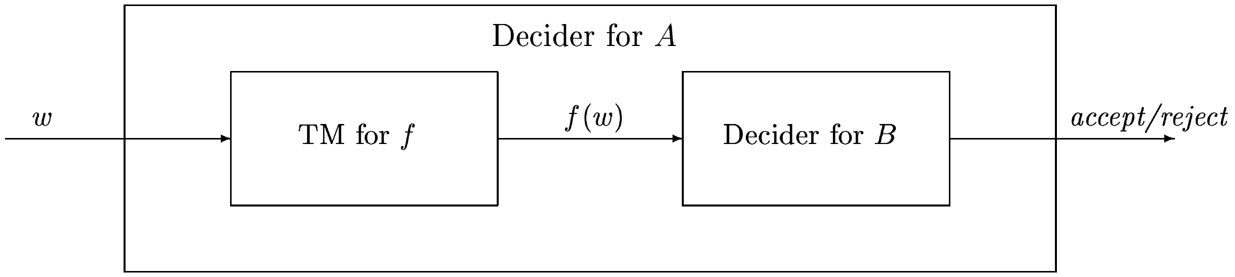
Definition. A function $f : \Sigma^* \longrightarrow \Sigma^*$ is *polynomial time (polytime) computable* if there is a polynomial time transducer TM that, when given any input w, halts with only $f(w)$ on its tape.

Definition. Suppose there is a polytime computable function f having the property that $w \in A$ iff $f(w) \in B$. Then we say A is *polynomial time (polytime) reducible* to B, written $A \leq_{\mathrm{P}} B$. We also call f a *polynomial time (polytime) reduction* from A to B.

Note: Such an f is just a mapping reduction from A to B, but here we impose the additional condition that it must run in polynomial time.

Implications of Polytime Reducibility

Suppose that there is a polytime reduction f from language A to language B. The following diagram depicts how a decider for A can be constructed by combining a TM that computes f with a decider for B:



Here is a description of the above TM, where F denotes the TM that computes f and D_B denotes the decider for B:

```
D_A =  "On input w:
   1.    Run F on w to compute f(w).
   2.    Run D_B on f(w). If it accepts, accept; if it rejects, reject."
```

Theorem. Suppose $A \leq_{\mathrm{P}} B$. Then:

1. If $B \in \mathrm{P}$, then $A \in \mathrm{P}$.
2. If $A \notin \mathrm{P}$, then $B \notin \mathrm{P}$.

Proof. Part 2 is just the contrapositive of part 1, so we'll just prove that. Let f denote the polytime reduction. Recall that this means that it has the property that $f(w) \in B$ iff $w \in A$. Since $B \in \mathrm{P}$, it has a polytime decider, so that's what we use as our decider D_B for B.

Now D_A is obviously a decider for A, but does it run in polynomial time? Yes, but let's look closely at this:

Let the running time of F be $O(n^k)$, where n denotes the size of its input string w, and let the running time of D_B be $O(m^l)$, where m denotes the size of the input $f(w)$ to D_B. How big could $f(w)$ be? No transducer TM can produce output longer than the number of computation steps it uses since it must use at least one step for each symbol it writes on its tape. Therefore $m = |f(w)| = O(n^k)$. This means the running time of D_B is $O(m^l) = O((n^k)^l) = O(n^{kl})$, so the overall running time of D_A is $O(n^k) + O(n^{kl})$, showing that its running time is, indeed, polynomial.

NP-Completeness

Definition. A language B is *NP-complete* if it satisfies these two conditions:

1. $B \in \mathrm{NP}$
2. $A \leq_{\mathrm{P}} B$ for every $A \in \mathrm{NP}$.

An immediate consequence of this definition and the previous theorem is the following. Suppose a polytime decider is found for any particular NP-complete language. Then:

- such a decider can be used as a subprocedure to create a polytime decider for any other language in NP; and therefore
- all languages in NP would be polytime decidable; so
- $\mathrm{P} = \mathrm{NP}$.

How does one go about proving a language is NP-complete?

Answer in most cases (for us, all cases): Use the following result.

Theorem. If $B \in \mathrm{NP}$ and $A \leq_{\mathrm{P}} B$ for some NP-complete language A, then B is NP-complete.

Proof. Since the composition of polytime reductions is a polytime reduction, if every language in NP is polytime reducible to A and A is polytime reducible to B, then every language in NP is polytime reducible to B.

Note the analogy with proving undecidability or non-Turing-recognizability:

- To prove a language B is undecidable, find an undecidable language A that mapping reduces to B.
- To prove a language B is non-Turing-recognizable, find a non-Turing-recognizable language A that mapping reduces to B.
- To prove a language B is NP-complete, find an NP-complete language A that polytime reduces to B (and also check that $B \in \mathrm{NP}$).

But this process has to start with a direct proof first ...

The Cook-Levin Theorem

Theorem. *SAT* is NP-complete.

Proof Idea. Step 1 is to show that $SAT \in$ NP. We've already done this, but for completeness we'll repeat it here: Use as a certificate the satisfying assignment. Verifying that it is a satisfying assignment can clearly be done in polynomial time.

The hard part is step 2, showing that every language in NP is polytime reducible to *SAT*. This is done using a direct proof, which can be found on pp. 277-282 of Sipser. We omit it since it's beyond the scope of this course.

In some sense, it's not surprising that there is an intimate relationship between TM computation and Boolean formulas: Computer circuits are built out of logic gates.

3SAT is NP-Complete

Theorem. *3SAT* is NP-complete.

Proof. Step 1: We've shown earlier that $\mathit{3SAT} \in \mathrm{NP}$, but for completeness we'll repeat that argument: Use as a certificate the satisfying assignment. It is clear that we can verify that it is a satisfying assignment for the given formula in polynomial time.

Step 2: We now show that $\mathit{SAT} \leq_{\mathrm{P}} \mathit{3SAT}$. Since we know by the Cook-Levin Theorem that *SAT* is NP-complete, it will then follow that *3SAT* is NP-complete. Here is the reduction:

```
F =  "On input ⟨$\phi$⟩, where $\phi$ is a Boolean formula:
     1.   Convert $\phi$ to a CNF formula $\phi'$, as described on the next page.
     2.   Convert $\phi'$ to a 3CNF formula $\phi''$, as described below.
     3.   Output ⟨$\phi''$⟩."
```

We discuss stage 1 in more detail on the following page. While it is well-known that every Boolean formula can be converted to a logically equivalent CNF formula in the same variables, it can be shown that this cannot necessarily be done in time polynomial in the size of the formula, so a more elaborate technique is necessary that introduces more variables and more clauses while still preserving satisfiability.

In fact, the technique we use has the property that every clause in the resulting CNF formula ϕ' has either 2 or 3 literals (except in the trivial case when the original formula consists of just a single literal), so all stage 2 needs to do to guarantee that there are exactly 3 literals in every clause is to repeat one of the literals, say the first, in any such clause. The resulting formula ϕ'' is obviously logically equivalent to ϕ' since $x \vee x$ is logically equivalent to x for any Boolean variable x.

Since the process to be described on the next page, along with the process of repeating a literal in any 2-literal clause, runs in time polynomial in the size of the Boolean formula ϕ, this is a polytime algorithm. Furthermore, as we also discuss on the next page, ϕ is satisfiable iff ϕ', and hence ϕ'', is, so this is a polytime reduction from *SAT* to *3SAT*. Therefore, *3SAT* is NP-complete.

Polytime Conversion to CNF
Preserving Satisfiability

Here we elaborate on stage 1 of the reduction from SAT to 3SAT. The steps are as follows:

1. Create what is essentially a parse tree for the formula, where each node represents and AND or OR of exactly 2 arguments, or a NOT of a single argument. The leaves of this tree should be literals.

2. Label each node of this tree with a new variable.

3. Create a new formula ϕ_1 that is a conjunction of all the relationships (using the $\Leftrightarrow$ Boolean operator) among the new and old variables according to this parse tree. Each conjunct will involve either 2 or 3 variables.

4. Use the fact that $p \Rightarrow q$ is logically equivalent to (henceforth denoted by $\equiv$) $\overline{p} \vee q$, so

$$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q) \equiv (\overline{p} \vee q) \wedge (p \vee \overline{q})$$

   to rewrite this formula using only the AND, OR, and NOT operators. This will create a logically equivalent formula ϕ_2 having two conjuncts for every conjunct in ϕ_1.

5. The individual conjuncts in this formula may not be in clause form, but they can be converted individually into this proper form by appropriate use of straightforward logical identities involving de Morgan's laws and the distributive laws for OR and AND. Note that each conjunct to be so converted involves at most 3 variables, so this process is guaranteed to generate no more than $2^3 = 8$ clauses per conjunct of the formula ϕ_2 created in the previous step. This is the final CNF result, which we call ϕ'.

Since the number of new variables is equal to the number of internal nodes in the parse tree, there is one for every operator in ϕ (not counting negations of single variables) so this number is essentially linear in the size of ϕ. In ϕ_1 there is one conjunct per newly added variable, and the conversion to ϕ_2 in step 4 doubles the number of conjuncts. Then the conversion to ϕ' in step 5 creates at most 8 clauses per conjunct in ϕ_2, so this results in at most 16 clauses per newly added variable. Thus the overall size of the final result ϕ_3, and hence the time to compute it, is polynomial in the size of ϕ.

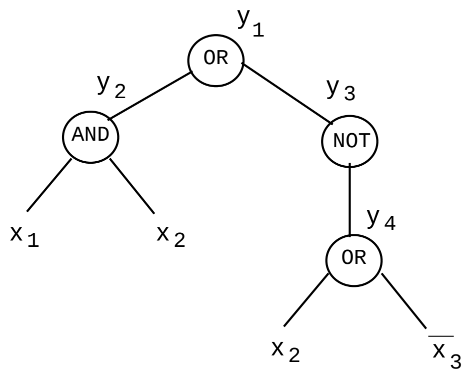
We also need to check that the final CNF formula ϕ' is satisfiable iff ϕ is. One direction is true because any satisfying assignment for ϕ' can be restricted to just the original variables, clearly giving a satisfying assignment for ϕ. For the other direction, if we have a satisfying assignment for ϕ, then this gives a well-defined set of values to all the newly added variables, and this combination of values must be a satisfying assignment for ϕ' by the way it is constructed.

Polytime Transformation of SAT Problem Instances to 3SAT Problems Instances: An Example

Here we illustrate the entire process of transforming a Boolean formula according to the description given on the previous page, combined with the additional easy step of modifying the CNF result so that every clause contains exactly 3 literals.

Consider the Boolean formula $\phi = (x_1 \wedge x_2) \vee \overline{(x_2 \vee \overline{x_3})}$.

Here is result of steps 1 and 2 of the polytime CNF conversion process described on the previous page when applied to ϕ. The additional variables y_1, y_2, y_3, y_4 are used to label the internal nodes of the tree as shown.



Based on this tree, step 3 of the conversion process produces the formula

$$\begin{aligned}\phi_1 \quad = \quad & (y_1 \Leftrightarrow y_2 \vee y_3) \wedge \\ & (y_2 \Leftrightarrow x_1 \wedge x_2) \wedge \\ & (y_3 \Leftrightarrow \overline{y_4}) \wedge \\ & (y_4 \Leftrightarrow x_2 \vee \overline{x_3}),\end{aligned}$$

where each conjunct represents the "constraint" that the value at that node must match the correct value for the computation performed at that node on its children.

Polytime Transformation of SAT Problem Instances to 3SAT Problem Instances: An Example (Continued)

Step 4 of the conversion process then yields the logically equivalent formula

$$\begin{aligned}\phi_2 \;=\;& (\overline{y_1} \vee y_2 \vee y_3) \wedge (y_1 \vee \overline{y_2 \vee y_3}) \wedge \\ & (\overline{y_2} \vee (x_1 \wedge x_2)) \wedge (y_2 \vee \overline{x_1 \wedge x_2}) \wedge \\ & (\overline{y_3} \vee \overline{y_4}) \wedge (y_3 \vee \overline{\overline{y_4}}) \wedge \\ & (\overline{y_4} \vee x_2 \vee \overline{x_3}) \wedge (y_4 \vee \overline{x_2 \vee \overline{x_3}}),\end{aligned}$$

which, finally, is logically equivalent to the CNF formula

$$\begin{aligned}\phi' \;=\;& (\overline{y_1} \vee y_2 \vee y_3) \wedge (y_1 \vee \overline{y_2}) \wedge (y_1 \vee \overline{y_3}) \wedge \\ & (\overline{y_2} \vee x_1) \wedge (\overline{y_2} \vee x_2) \wedge (y_2 \vee \overline{x_1} \vee \overline{x_2}) \wedge \\ & (\overline{y_3} \vee \overline{y_4}) \wedge (y_3 \vee y_4) \wedge \\ & (\overline{y_4} \vee x_2 \vee \overline{x_3}) \wedge (y_4 \vee \overline{x_2}) \wedge (y_4 \vee x_3).\end{aligned}$$

If we now repeat the first literal in each clause containing fewer than 3 literals to bring the number up to exactly 3, we end up with the 3CNF formula

$$\begin{aligned}\phi'' \;=\;& (\overline{y_1} \vee y_2 \vee y_3) \wedge (y_1 \vee y_1 \vee \overline{y_2}) \wedge (y_1 \vee y_1 \vee \overline{y_3}) \wedge \\ & (\overline{y_2} \vee \overline{y_2} \vee x_1) \wedge (\overline{y_2} \vee \overline{y_2} \vee x_2) \wedge (y_2 \vee \overline{x_1} \vee \overline{x_2}) \wedge \\ & (\overline{y_3} \vee \overline{y_3} \vee \overline{y_4}) \wedge (y_3 \vee y_3 \vee y_4) \wedge \\ & (\overline{y_4} \vee x_2 \vee \overline{x_3}) \wedge (y_4 \vee y_4 \vee \overline{x_2}) \wedge (y_4 \vee y_4 \vee x_3).\end{aligned}$$

CLIQUE is NP-Complete

Theorem. *CLIQUE* is NP-complete.

Proof. Step 1: We've shown earlier that *CLIQUE* $\in$ NP because a list of the k nodes forming the clique can be used as the certificate and verifying that it's a clique can be done in polynomial time.

Step 2: We now show that *3SAT* $\leq_{\mathrm{P}}$ *CLIQUE*. Since we know *3SAT* is NP-complete, it will then follow that *CLIQUE* is NP-complete. Here is the reduction:

```
F =  "On input ⟨φ⟩, where φ is a 3CNF Boolean formula having k clauses:
        1.   Create the graph G as described below.
        2.   Output ⟨G, k⟩."
```

We now show how the graph G is constructed. Write the given Boolean formula as the AND of its clauses, as $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$, where $c_i = l_1^i \vee l_2^i \vee l_3^i$ for $1 \leq i \leq k$, with each l_j^i being the literal x_m or $\overline{x_m}$ for some variable x_m.

Then define G as follows: There is one node u_j^i for each literal l_j^i with $1 \leq j \leq 3$ and $1 \leq i \leq k$. Call each of the 3 nodes corresponding to a single clause a *triple*. Since there are k clauses, and hence k triples, there are a total of $3k$ nodes in G. For each pair of literals from different clauses there is an edge between corresponding nodes except when they represent contradictory literals. More formally, there is an edge (u_j^i, u_q^p) in G for every $i \neq p$ and l_j^i is not the negation of l_q^p. Expressed in terms of the Boolean formula, there are no within-clause connections, nor is any literal connected to its logical negation.

Now suppose $\langle \phi \rangle \in$ *3SAT*, where k is the number of clauses in ϕ. This means that ϕ has a satisfying assignment, so at least one literal in every clause must be 1. Let the literals assigned the value 1 in each clause be $l_{i_1}^1, l_{i_2}^2, \ldots, l_{i_k}^k$. Then the corresponding nodes $u_{i_1}^1, u_{i_2}^2, \ldots, u_{i_k}^k$ form a k-clique since there are edges between every pair of these since they're all from different clauses and no pair of them corresponds to contradictory literals. Therefore $\langle \phi \rangle \in$ *3SAT* implies $\langle G, k \rangle \in$ *CLIQUE*.

For the other direction, suppose $\langle G, k \rangle \in$ *CLIQUE*, which means G has a k-clique. Since there are no within-triple edges, a k-clique must consist of exactly one node from each of the k triples. Let the nodes in the k-clique be $u_{i_1}^1, u_{i_2}^2, \ldots, u_{i_k}^k$. If we assign 1 to each of the corresponding literals $l_{i_1}^1, l_{i_2}^2, \ldots, l_{i_k}^k$, this would make each clause TRUE, which would represent a satisfying assignment unless this led to a contradictory assignment where both a variable x_m and its negation $\overline{x_m}$ were assigned the value 1. But this can't happen since G was constructed so that no pair of nodes corresponding to contradictory literals has an edge between them, so any two such nodes could not be part of a clique. Therefore this way of assigning values according to a k-clique gives a valid satisfying assignment for ϕ. (It may happen that not every variable has a corresponding node in the clique, but any such variables are free to be assigned 0 or 1 arbitrarily and the result will be a satisfying assignment.) Thus $\langle G, k \rangle \in$ *CLIQUE* implies $\langle \phi \rangle \in$ *3SAT*.

Therefore $\langle \phi \rangle \in$ *3SAT* iff $\langle G, k \rangle \in$ *CLIQUE* so this is a valid mapping reduction. Since the number of nodes in G is $O(k)$ and the number of edges is $O(k^2)$, it's not hard to see that construction of G from ϕ can be done in time polynomial in the size of ϕ. Therefore this is a polytime reduction from *3SAT* to *CLIQUE*, and since *3SAT* is NP-complete, it follows that *CLIQUE* is NP-complete.

Transforming 3SAT Problem Instances to CLIQUE Problem Instances: An Example

Figure 7.33 of the Sipser text shows the graph corresponding to one example. Here we'll do a different example.

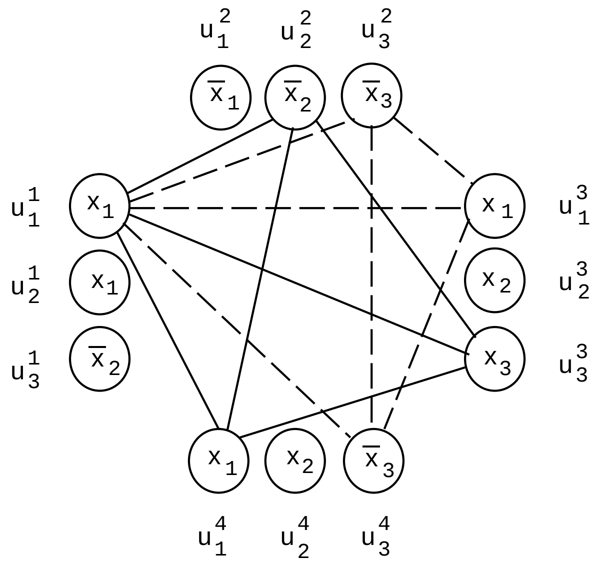
Consider the 3CNF Boolean formula

$$\phi = (x_1 \vee x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3})$$

Its has the four clauses

$$\begin{aligned} c_1 &= x_1 \vee x_1 \vee \overline{x_2} \\ c_2 &= \overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \\ c_3 &= x_1 \vee x_2 \vee x_3 \\ c_4 &= x_1 \vee x_2 \vee \overline{x_3} \end{aligned}$$

Below is part of the graph G constructed to correspond to this ϕ. All the nodes of G are shown, but many edges of G are not shown to prevent the diagram from being hopelessly cluttered. The node labels shown (outside each node) are consistent with the labeling used in the proof on the previous page, where u_j^i is the node corresponding to the j^{th} literal in clause c_i. The actual literal each node corresponds to in ϕ is shown inside the node.



The reduction from *3SAT* to *CLIQUE* transforms the question of satisfiability of this formula ϕ to the question of the existence of k-cliques in this graph G, where $k = 4$ is the number of clauses in ϕ. For this reason the above diagram displays just those edges belonging to two of several 4-cliques that exist in the graph. One 4-clique has its edges displayed using solid lines and the other has its edges displayed using dashed lines.

Transforming 3SAT Problem Instances to CLIQUE Problem Instances: An Example (Continued)

The 4-clique whose edges are displayed as solid lines is the set of nodes

$$\left\{u_1^1, u_2^2, u_3^3, u_1^4\right\}.$$

This 4-clique corresponds to assigning values to literals as follows:

$$x_1 = 1, \overline{x_2} = 1, x_3 = 1, x_1 = 1,$$

which gives the satisfying assignment

$$x_1 = 1, x_2 = 0, x_3 = 1.$$

The 4-clique whose edges are displayed as dashed lines is the set of nodes

$$\left\{u_1^1, u_3^2, u_1^3, u_3^4\right\}.$$

This 4-clique corresponds to assigning values to literals as follows:

$$x_1 = 1, \overline{x_3} = 1, x_1 = 1, \overline{x_3} = 1.$$

Since this does not constrain the value of x_2, this gives two satisfying assignments

$$x_1 = 1, x_2 = 0, x_3 = 0 \text{ and } x_1 = 1, x_2 = 1, x_3 = 0.$$

VERTEX-COVER is NP-Complete

Theorem. *VERTEX-COVER* is NP-complete.

Proof. Step 1: We've shown earlier that *VERTEX-COVER* $\in$ NP because a list of the k nodes forming a vertex cover can be used as the certificate and verifying that it's a vertex cover can be done in polynomial time.

Step 2: We now show that *CLIQUE* $\leq_{\mathrm{P}}$ *VERTEX-COVER*. Since we know *CLIQUE* is NP-complete, it will then follow that *VERTEX-COVER* is NP-complete. Here is the reduction:

```
F =  "On input ⟨G, k⟩, where G = {V, E} is an undirected graph:
      1.   Create the complement graph $\overline{G}$ as described below.
      2.   Output $\langle \overline{G}, |V| - k \rangle$."
```

For any undirected graph $G = (V, E)$, its complement $\overline{G}$ is defined to be $(V, \overline{E})$, where $\overline{E}$ consists of those pairs (i, j) such that (i, j) is not an edge in G. In other words, $\overline{G}$ is the graph consisting of all nodes in G and exactly those edges that are not in G. We now show that G has a clique of size k iff $\overline{G}$ has a vertex cover of size $|V| - k$.

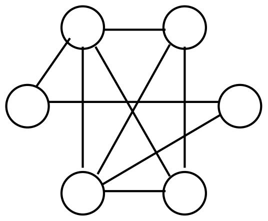
Assume G has a clique S of size k. This means that every pair of nodes in S has an edge between them. So in $\overline{G}$, there are no edges between any pair of nodes in S. Thus every edge in $\overline{G}$ is connected to some node not in S, which are the nodes in $V - S$. Since every edge in $\overline{G}$ is connected to some node in $V - S$, $V - S$ forms a vertex cover for $\overline{G}$. Since $V - S$ has $|V| - k$ elements in it, $\overline{G}$ has a vertex cover of size $|V| - k$. Therefore $\langle G, k \rangle \in$ *CLIQUE* implies $\langle \overline{G}, |V| - k \rangle \in$ *VERTEX-COVER*.

For the other direction, assume $\langle \overline{G}, |V| - k \rangle \in$ *VERTEX-COVER*. This means there is a set S' of nodes in $\overline{G}$ of size $|V| - k$ such that every edge in $\overline{G}$ includes some node in S'. This means there can be no edges between any pair of nodes not in S', i.e., between any pair of nodes in $V - S'$. Let $S = V - S'$. Then S consists of $|V| - (|V| - k) = k$ nodes of $\overline{G}$ having no edges between any of them. But this means that in G, these k nodes are fully connected with each other, so they form a k-clique. Therefore $\langle \overline{G}, |V| - k \rangle \in$ *VERTEX-COVER* implies $\langle G, k \rangle \in$ *CLIQUE*.

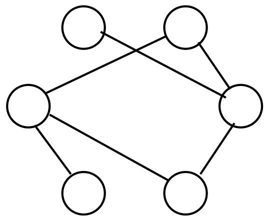
Thus this is a valid mapping reduction from *CLIQUE* to *VERTEX-COVER*, and it is easily implemented in polynomial time. Therefore *VERTEX-COVER* is NP-complete since *CLIQUE* is.

Transforming CLIQUE Problem Instances to VERTEX-COVER Problem Instances: An Example

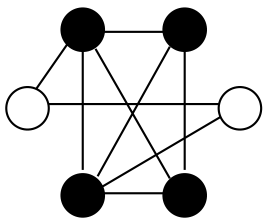
G



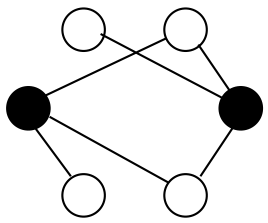
$\overline{G}$



4-clique in G



Size-2 vertex cover in $\overline{G}$



The proof on the previous page is really a proof of this result:

Theorem. For any undirected graph $G = (V, E)$, $S \subseteq V$ is a k-clique in G iff $V - S$ is a vertex cover in $\overline{G}$.

HAMPATH is NP-Complete

Theorem. *HAMPATH* is NP-complete.

Proof. Step 1: We've shown earlier that *HAMPATH* $\in$ NP because a list of all nodes in the Hamiltonian path can be used as the certificate and verifying that it's a Hamiltonian path from s to t can be done in polynomial time.

Step 2: Now we will show that *3SAT* $\leq_{\mathrm{P}}$ *HAMPATH*. Since we know *3SAT* is NP-complete, it will then follow that *HAMPATH* is NP-complete. Here is the reduction:

```
F =  "On input ⟨$\phi$⟩, where $\phi$ is a 3CNF Boolean formula:
      1.  Construct the graph $G$ as described on pp. 286-290 of Sipser, with distinguished nodes
          $s$ and $t$.
      2.  Output ⟨$G, s, t$⟩."
```

The graph constructed in stage 1 involves a lot of nodes, so we omit the details here. Here's an executive summary:

- For each of the k clauses in ϕ there's a single node.
- For each of the m variables in ϕ there's an elaborate diamond-shaped structure consisting of $3k + 5$ nodes, but adjacent diamonds share a node.
- The total number of nodes is $(3k + 4)m + k + 1$.
- There are directed edges laid out in a standard pattern within each diamond.
- There are additional directed edges connecting parts of each diamond structure to the clause nodes, based on which variables appear in which clauses, and the exact way they're connected depends on whether that variable appears in positive form or is negated.
- The end result is as desired: The original formula is satisfiable iff the graph as constructed has a Hamiltonian path from s to t.

UHAMPATH is NP-Complete

Theorem. *UHAMPATH* is NP-complete.

Proof. Step 1: We've shown earlier that $UHAMPATH \in \mathrm{NP}$ since a list of nodes forming the Hamiltonian path can be used as the certificate and verifying that it's a Hamiltonian path can be done in polynomial time.

Step 2: Now we show that $HAMPATH \leq_{\mathrm{P}} UHAMPATH$. Since we know *HAMPATH* is NP-complete, it will then follow that *UHAMPATH* is NP-complete. Here is the reduction:

```
F =  "On input ⟨G, s, t⟩, where G is a directed graph containing nodes s and t:
     1.   Construct the undirected graph G′ as described below, with distinguished nodes
          s^out and t^in.
     2.   Output ⟨G′, s^out, t^in⟩."
```

The undirected graph G' is constructed from G as follows:

Nodes in G': For every node u in G, except s and t, there are 3 nodes in G': $u^{\text{in}}, u^{\text{mid}}, u^{\text{out}}$. Call each such set of 3 nodes a *triple*. For node s, there is a single node s^{out} and for node t there is a single node t^{in}. (This still applies if $s = t$.)

Edges in G': For every node u in G except s and t there are 2 edges: $(u_i^{\text{in}}, u_i^{\text{mid}})$ and $(u_i^{\text{mid}}, u_i^{\text{out}})$. For every directed edge (u_i, u_j) in G there is an (undirected) edge $(u_i^{\text{out}}, u_j^{\text{in}})$. This includes all directed edges leaving s and all directed edges into t. Any directed edges in G that go into s or out of t have no counterparts in G'.

Suppose $\langle G, s, t\rangle \in HAMPATH$. Then there is a Hamiltonian path visiting all the nodes of G in the order $s, u_1, u_2, \ldots, u_k, t$. But then it's easy to see that

$$s^{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \ldots, u_k^{\text{in}}, u_k^{\text{mid}}, u_k^{\text{out}}, t^{\text{in}}$$

is a Hamiltonian path in G'. Thus $\langle G', s^{\text{out}}, t^{\text{in}}\rangle \in UHAMPATH$.

For the other direction, suppose there is a Hamiltonian path from s^{out} to t^{in} in G'. Then the first edge in this path must go to some triple (since it can't go directly to t^{in} unless there are only two nodes in G', in which case there are only two nodes in G, and the conclusion is easily seen to be true in this case). Consider this triple to have the index 1. Since s^{out} is not connected to u_1^{mid} or u_1^{out}, this path must go from s^{out} to u_1^{in}. From there, the path must go to u_1^{mid} since u_1^{mid} is only connected to two nodes and one of these has already been visited. By the same reasoning, the next node must be u_1^{out}.

Following this, the next node must belong to another triple. Inductively, the same reasoning as above shows that entire triples must be visited in the order $u_i^{\text{in}}, u_i^{\text{mid}}, u_i^{\text{out}}$. Thus the Hamiltonian path in G' must have the form

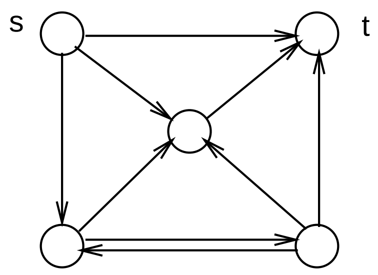
$$s^{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \ldots, u_k^{\text{in}}, u_k^{\text{mid}}, u_k^{\text{out}}, t^{\text{in}}.$$

Therefore $s, u_1, u_2, \ldots, u_k, t$ is a directed path in G from s to t. Thus $\langle G', s^{\text{out}}, t^{\text{in}}\rangle \in UHAMPATH$ implies $\langle G, s, t\rangle \in HAMPATH$.

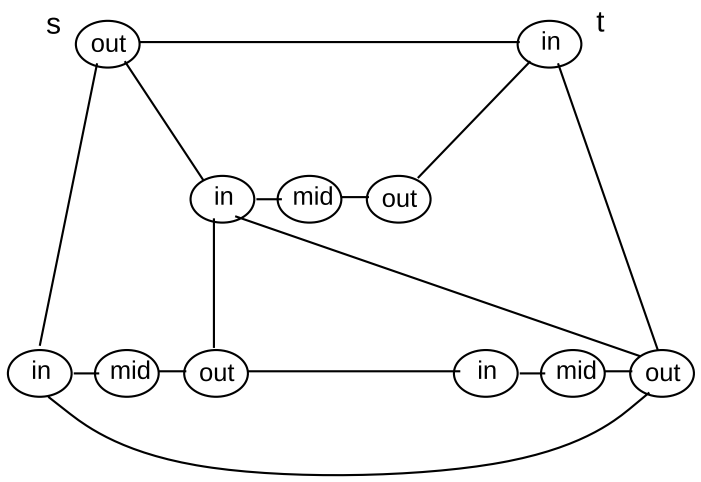
Therefore this is a valid mapping reduction from *HAMPATH* to *UHAMPATH*, and it is easy to see it can be performed in polynomial time. Since *HAMPATH* is NP-complete, it follows that *UHAMPATH* is NP-complete.

Transforming HAMPATH Problem Instances to UHAMPATH Problem Instances: An Example

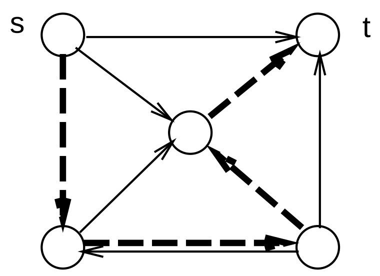
Directed graph G



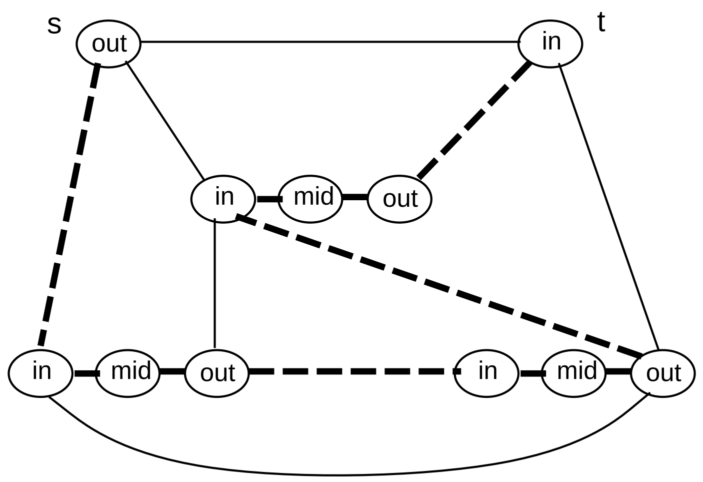
Undirected graph G'



Hamiltonian path in G from s to t



Hamiltonian path in G' from s^{out} to t^{in}



G has a Hamiltonian path from s to t iff G' has a Hamiltonian path from s^{out} to t^{in}.

Summary of Polytime Reductions Explicitly Described (or Summarized) in This Handout

- $\mathit{SAT} \leq_{\mathrm{P}} \mathit{3SAT}$
- $\mathit{3SAT} \leq_{\mathrm{P}} \mathit{CLIQUE}$
- $\mathit{CLIQUE} \leq_{\mathrm{P}} \mathit{VERTEX\text{-}COVER}$
- $\mathit{3SAT} \leq_{\mathrm{P}} \mathit{VERTEX\text{-}COVER}$
- $\mathit{3SAT} \leq_{\mathrm{P}} \mathit{HAMPATH}$
- $\mathit{HAMPATH} \leq_{\mathrm{P}} \mathit{UHAMPATH}$

Epilogue: Relationship Between P and NP

Define the class of languages co-NP to be

$$\text{co-NP} = \{L \mid \overline{L} \in \text{NP}\}$$

Examples of languages in co-NP:

- $\overline{SAT}$
- $\overline{HAMPATH}$

How could you verify in polynomial time that

- a Boolean formula is not satisfiable?
- a directed graph does not have a Hamiltonian path?

Relationships between P, NP and co-NP:

- Recall that P $\subseteq$ NP since a TM is just a special case of an NTM.
- Thus P $\subseteq$ co-NP since P is closed under complement.
- Therefore P $\subseteq$ NP $\cap$ co-NP

So far this is all that can be proved about these language classes.

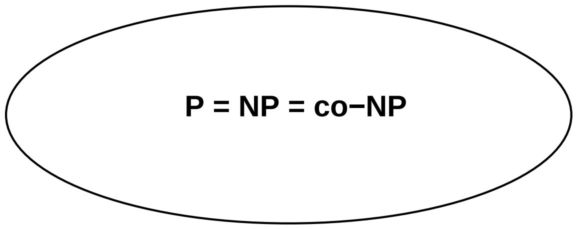
What's the significance of the NP-complete languages?

- If a polytime decider can be found for any one of them, this would lead to polytime deciders for all of them (as well as for all other languages in NP).
  - This would further imply that many combinatorial problems of practical interest (not just yes/no decision problems) can be solved in polynomial time.
- If it can be proven for any one of them that it cannot be decided in polynomial time, this would imply that none of them can be decided in polynomial time.
  - This would further imply that many combinatorial problems of practical interest (not just decision problems) cannot be solved in polynomial time.

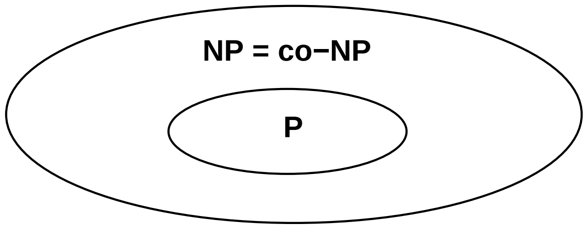
Relationship Between P and NP (Continued)

The current state of our knowledge is that any one of these four mutually exclusive possibilities may be true:

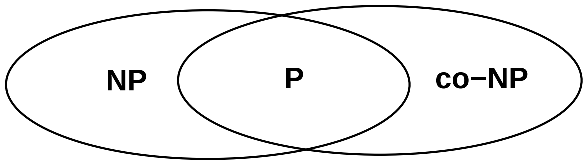
P $=$ NP $=$ co-NP



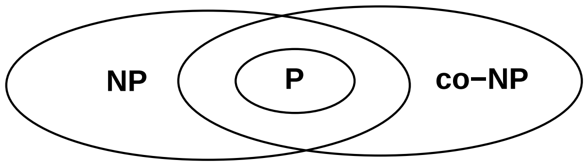
P $\neq$ NP $=$ co-NP



NP $\neq$ P $\neq$ co-NP and P $=$ NP $\cap$ co-NP



NP $\neq$ co-NP and P $\neq$ NP $\cap$ co-NP



Prove which one of these is true and you'll be famous!