Summary of the previous lecture

• Recall that we mentioned the following topics:

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

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

NP-Hard: L is **NP-hard** 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.

NP-Complete: L is **NP-complete** if (1) $L \in$**NP** and (2) L is **NP-hard**.

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

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

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

• An alternative way to show that a problem is **NP-complete** is to use transitivity of $\prec_P$.

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

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

Cook's Theorem and Reductions

• Unfortunately, to use this lemma, we need to have *at least one* **NP-complete** problem to start the ball rolling. Stephen Cook showed that such a problem existed. Cook's theorem is quite complicated to prove, but 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.

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

• He 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 $\bar{x}$ 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, $(x_1 \wedge (x_2 \vee \bar{x}_3)) \wedge ((\bar{x}_2 \wedge \bar{x}_3) \vee \bar{x}_1)$

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

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

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

• We will not prove this theorem. The proof would take about a full lecture (not counting the week or so of background on Turing machines).

• The proof and many other interesting results on complexities will be given in the ***Theory of Computation*** course (offered some time later).

3-Conjunctive Normal Form (3-CNF)

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

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

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

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

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

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

• 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.

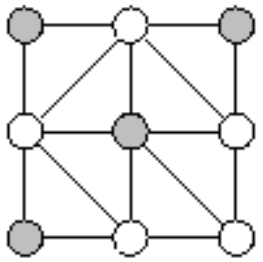
NP-completeness proofs

Now that we know that ***3SAT*** is **NP-complete**, we can use this fact to prove that other problems are **NP-complete**. 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 **NP-complete**.

• 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 **NP-hard**, which can be done by showing that some known **NP-complete** problem (***3SAT***) is polynomial-time reducible to ***IS***, that is, ***3SAT*** $\prec_P$ ***IS.***

• Let *F* be a boolean formula in *3-CNF* 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 **NP-complete**, 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 $\bar{x}$ 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 *3-CNF* 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 ← 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 (x1,x2,x3) {
    create an edge (xi,xj) between all pairs of vertices in the cluster;
}
for each vertex xi {
    create edges between xi and all its complement vertices x̄i;
}
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 *3-CNF* formula:

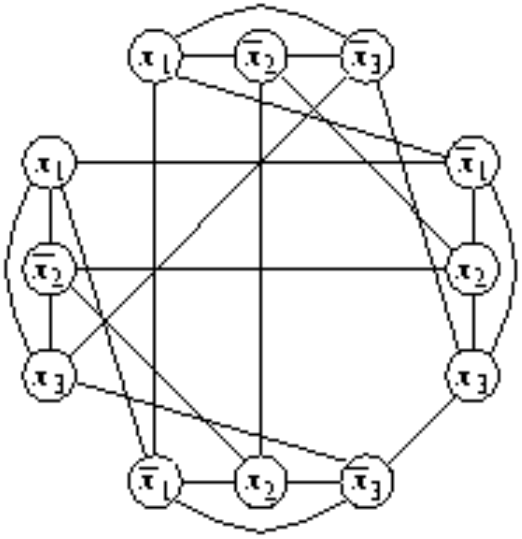
$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3).$$

• The reduction produces the graph shown in the following figure and sets $k = 4$.

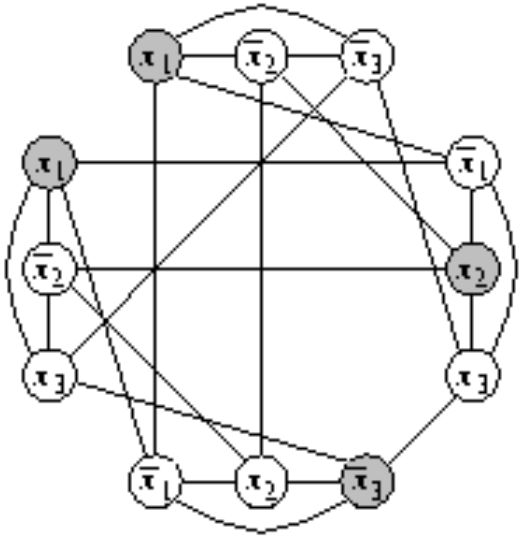
• In our example, the formula is satisfied by the assignment $x_1 = 1, x_2 = 1, x_3 = 0$.

• 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 inter-cluster 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 ($x_i, \bar{x}_i$) 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 $\bar{x}_i$ 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 **NP-complete** 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 **NP-complete** 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 **NP-complete** 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 clique-cover (***CCov***) problem that we discussed in an earlier lecture. The clique problem seeks to find a single clique of size k, and the clique-cover 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, $\overline{G}$.

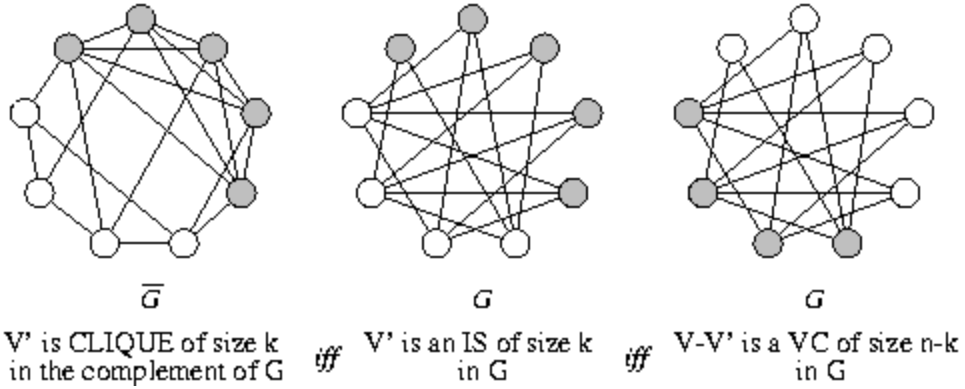
(ii) V' is an independent set of size k for G.

(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.



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 **NP-complete**.

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 **NP-complete**.

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.

Hamiltonian Cycle/Path Problems

(The reduction we present for Hamiltonian Path is completely different from the one in Chapter 34.5.3 of CLRS.)

• Today we consider a collection of problems related to finding paths in graphs and digraphs.

• Recall that given a graph (or digraph) a ***Hamiltonian cycle*** is a simple cycle that visits every vertex in the graph (exactly once).

• A ***Hamiltonian path*** is a simple path that visits every vertex in the graph (exactly once).

• The Hamiltonian cycle (HC) and Hamiltonian path (HP) problems ask whether a given graph (or digraph) has such a cycle or path, respectively.

• There are four variations of these problems depending on whether the graph is directed or undirected, and depending on whether you want a path or a cycle, but all of these problems are NP-complete.

• An important related problem is the ***traveling salesman problem*** (TSP). Given a complete graph (or digraph) with integer edge weights, determine the cycle of minimum weight that visits all the vertices. Since the graph is complete, such a cycle will always exist. The decision problem formulation is, given a complete weighted graph G, and integer X, does there exist a Hamiltonian cycle of total weight at most X?

• Today we will prove that Hamiltonian Cycle is NP-complete. We will leave TSP as an easy exercise.

Component design

- Up to now, most of the reductions that we have seen (for ***Clique*** and ***VC*** particular) are of a relatively simple variety. They are sometimes called local replacement reductions, because they operate by making some local change throughout the graph.
- We will present a much more complex style of reduction for the Hamiltonian path problem on directed graphs. This type of reduction is called a *component design reduction*, because it involves designing special subgraphs, sometimes called *components or gadgets* (also called *widgets*), whose job it is to enforce a particular constraint.
- Very complex reductions may involve the creation of many gadgets. This one involves the construction of only one. (See CLRS's presentation of HC for other examples of gadgets.)
- The gadget that we will use in the ***directed Hamiltonian path*** reduction, called a DHP-*gadget*, is shown in the figure on the next slide. It consists of three incoming edges labeled i_1, i_2, i_3 and three outgoing edges, labeled o_1, o_2, o_3 . It was designed so it satisfied the following property, which you can verify. Intuitively it says that if you enter the gadget on any subset of 1, 2 or 3 input edges, then there is a way to get through the gadget and hit every vertex exactly once, and in doing so each path must end on the corresponding output edge.

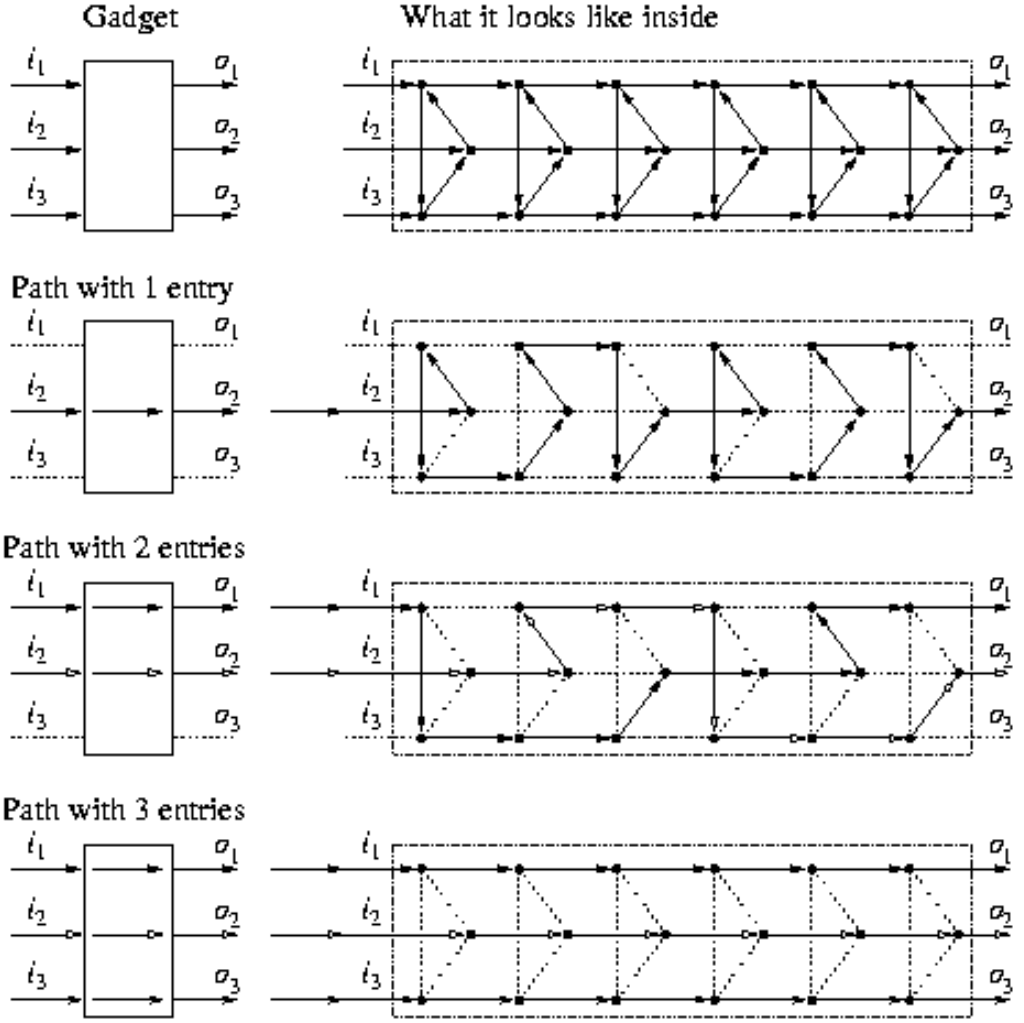
Claim: Given the DHP-gadget:

1. For any subset of input edges, there exists a set of paths which join each input edge i_1, i_2, i_3 to its respective output edge o_1, o_2, o_3 such that together these paths visit every vertex in the gadget exactly once.

Component design (cont.)

2. Any subset of paths that start on the input edges and end on the output edges, and visit all the vertices of the gadget exactly once, must join corresponding inputs to corresponding outputs. (In other words, a path that starts on input i_1 must exit on output o_1 .)

- The proof is not hard, but involves a careful inspection of the gadget.
- It is probably easiest to see this on your own, by starting with one, two, or three input paths, and attempting to get through the gadget without skipping vertex and without visiting any vertex twice.
- To see whether you really understand the gadget, answer the question of why there are 6 groups of triples.
- Would some other number work?



DHP-Gadget and examples of path traversals.

DHP is NP-Complete

• This gadget is an essential part of our proof that the directed Hamiltonian path problem is NP-complete.

Theorem: The ***directed Hamiltonian Path*** problem is **NP-complete.**

Proof: DHP $\in$**NP**: The certificate consists of the sequence of vertices (or edges) in the path. It is an easy matter to check that the path visits every vertex exactly once.

3SAT $\prec_P$ ***DHP***: This will be the subject of the rest of this section.

• Let us consider the similar elements between the two problems.

• In ***3SAT*** we are selecting a truth assignment for the variables of the formula. In ***DHP***, we are deciding which edges will be a part of the path.

• In ***3SAT*** there must be at least one true literal for each clause. In ***DHP***, each vertex must be visited exactly once.

• We are given a boolean formula F in ***3-CNF*** form (three literals per clause). We will convert this formula into a digraph. Let $x_1, x_2, ..., x_m$ denote the variables appearing in F .

• We will construct one DHP-gadget for each clause in the formula. The inputs and outputs of each gadget correspond to the literals appearing in this clause.

• Thus, the clause $(\overline{x}_2 \vee x_5 \vee \overline{x}_8)$ would generate a clause gadget with inputs labeled $\overline{x}_2, x_5, \overline{x}_8$, and the same outputs.

DHP is **NP-Complete** (cont.)

• The general structure of the digraph will consist of a series vertices, one for each variable.

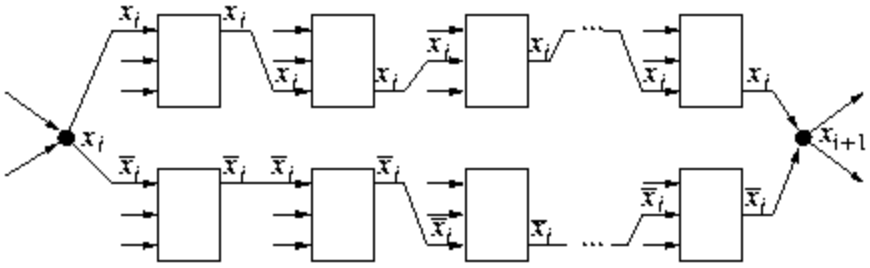
• Each of these vertices will have two outgoing paths, one taken if x_i is set to true and one if x_i is set to false. Each of these paths will then pass through some number of DHP-gadgets.

• The true path for x_i will pass through all the clause gadgets for clauses in which x_i appears, and the false path will pass through all the gadgets for clauses in which $\overline{x}_i$ appears. (The order in which the path passes through the gadgets is unimportant.)

• When the paths for x_i have passed through their last gadgets, then they are joined to the next variable vertex, x_{i+1} .

• This is illustrated in the following figure. (The figure only shows a portion of the construction. There will be paths coming into these same gadgets from other variables as well.)

• We add one final vertex x_e, and the last variable's paths are connected to x_e . (If we wanted to reduce to Hamiltonian cycle, rather than Hamiltonian path, we could join x_e back to x_1 .)



General structure of reduction from 3SAT to DHP.

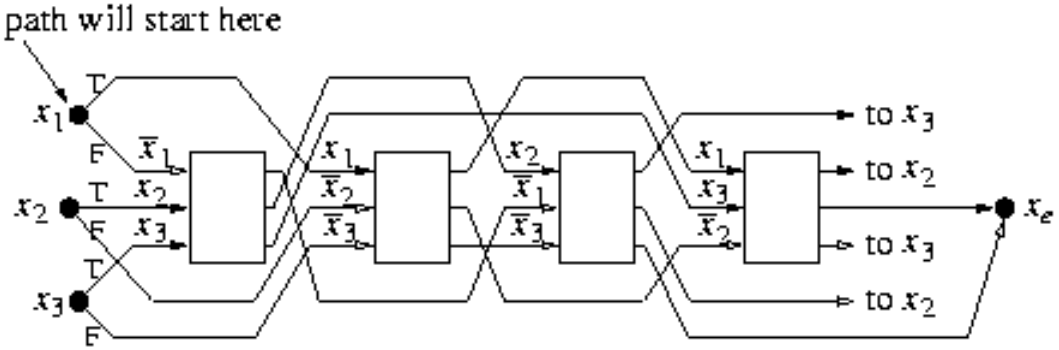
DHP: Example

• Note that for each variable, the Hamiltonian path must either use the true path or the false path, but it cannot use both.

• If we choose the true path for x_i to be in the Hamiltonian path, then we will have at least one path passing through each of the gadgets whose corresponding clause contains x_i , and if we chose the false path, then we will have at least one path passing through each gadget for x_i .

• For example, consider the following boolean formula in ***3-CNF***. The construction yields the digraph shown in the following figure.

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



Example of the 3SAT to DHP reduction.

DHP: Reduction

• Let us give a more formal description of the reduction. Recall that we are given a boolean formula F in ***3-CNF***. We create a digraph G as follows.

• For each variable x_i appearing in F , we create a *variable* vertex, named x_i . We also create a vertex named x_e (the ending vertex).

• For each clause *c*, we create a DHP-gadget whose inputs and outputs are labeled with the three literals of *c*. (The order is unimportant, as long as each input and its corresponding output are labeled the same.)

• We join these vertices with the gadgets as follows. For each variable x_i, consider all the clauses $c_1, c_2, ..., c_k$ in which x_i appears as a literal (uncomplemented). Join x_i by an edge to the input labeled with x_i in the gadget for c_1 , and in general join the the output of gadget c_j labeled x_i with the input of gadget c_{j+1} with this same label. Finally, join the output of the last gadget c_k to the next vertex variable x_{i+1}. (If this is the last variable, then join it to x_e instead.) The resulting chain of edges is called the true path for variable x_i .

• Form a second chain in exactly the same way, but this time joining the gadgets for the clauses in which $\bar{x}_i$ appears. This is called the false path for x_i .

• The resulting digraph is the output of the reduction. Observe that the entire construction can be performed in polynomial time, by simply inspecting the formula, creating the appropriate vertices, and adding the appropriate edges to the digraph.

• The following lemma establishes the correctness of this reduction.

DHP: Correctness of the Reduction

Lemma: The boolean formula F is satisfiable if and only if the digraph G produced by the above reduction has a Hamiltonian path.

Proof: We need to prove both the ``only if'' and the ``if''.

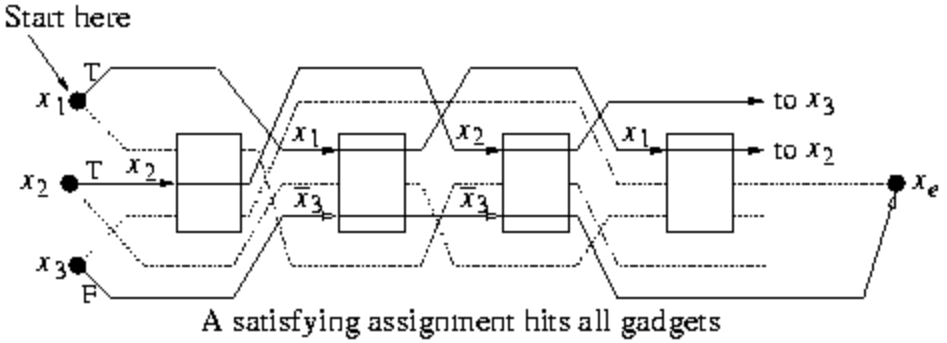
➔: Suppose that F has a satisfying assignment. We claim that G has a Hamiltonian path.

• This path will start at the variable vertex x_1, then will travel along either the true path or false path for x_1, depending on whether it is 1 or 0, respectively, in the assignment, and then it will continue with x_2, then x_3, and so on, until reaching x_e.

• Such a path will visit each variable vertex exactly once. Because this is a satisfying assignment, we know that for each clause, either 1, 2, or 3 of its literals will be true. This means that for each clause, either 1, 2, or 3, paths will attempt to travel through the corresponding gadget. However, we have argued in the above claim that in this case it is possible to visit every vertex in the gadget exactly once.

• Thus every vertex in the graph is visited exactly once, implying that G has a Hamiltonian path.

Correctness of the ***3SAT*** to ***DHP*** reduction. The figure shows the Hamiltonian path resulting from the satisfying assignment, x1 = 1, x2 = 1, x3 = 0.



A satisfying assignment hits all gadgets

DHP: Correctness of the Reduction (cont.)

⇐: Suppose that G has a Hamiltonian path. We assert that the form of the path must be essentially the same as the one described in the previous part of this proof.

• In particular, the path must visit the variable vertices in increasing order from $x_1$until x_e , because of the way in which these vertices are joined together.

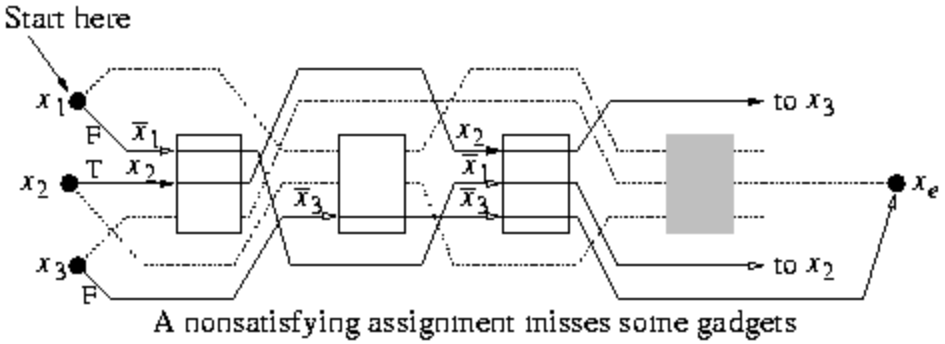
• Also observe that for each variable vertex, the path will proceed along either the true path or the false path. If it proceeds along the true path, set the corresponding variable to 1 and otherwise set it to 0.

• We show that the resulting assignment is a satisfying assignment for F .

• Any Hamiltonian path must visit all the vertices in every gadget. By the above claim about DHP-gadgets, if a path visits all the vertices and enters along input edge then it must exit along the corresponding output edge. Therefore, once the Hamiltonian path starts along the true or false path for some variable, it must remain on edges with the same label. That is, if the path starts along the true path for x_i , it must travel through all the gadgets with the label x_i until arriving at the variable vertex for x_{i+1}. If it starts along the false path, then it must travel through all gadgets with the label $\overline{x}_i$.

• Since all the gadgets are visited and the paths must remain true to their initial assignments, it follows that for each corresponding clause, at least one (and possibly 2 or three) of the literals must be true. Therefore, this is a satisfying assignment.

Correctness of the ***3SAT*** to ***DHP*** reduction. The figure shows the the non-Hamiltonian path resulting from the nonsatisfying assignment x1 = 0, x2 = 1, x3 = 0.



A nonsatisfying assignment misses some gadgets