Here is how you might do this. Given an undirected graph G, create a directed graph G' by just replacing each undirected edge $\{u, v\}$ with two directed edges, (u, v) and (v, u). Now, every simple path in the G is a simple path in G', and vice versa. Therefore, G has a Hamiltonian cycle if and only if G' does. Now, if you could develop an efficient solution to the DHC problem, you could use this algorithm and this little transformation solve the UHC problem. Here is your algorithm for solving the undirected Hamiltonian cycle problem. You take the undirected graph G, convert it to an equivalent directed graph G' (by edge-doubling), and then call your (supposed) algorithm for directed Hamiltonian cycles. Whatever answer this algorithm gives, you return as the answer for the Hamiltonian cycle.

UHC to DHC Reduction

```
bool Undir_Ham_Cycle(graph G) {
    create digraph G' with the same number of vertices as G
    for each edge {u,v} in G {
        add edges (u,v) and (v,u) to G'
    }
    return Dir_Ham_Cycle(G')
}
```

You would now have a polynomial time algorithm for UHC. Since you and your boss both agree that this is not going to happen soon, he agrees to let you off.

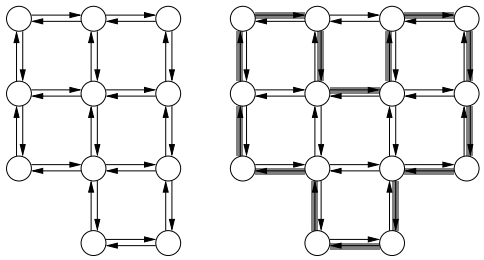

Figure 37: Directed Hamiltonian cycle reduction.

Notice that neither problem UHC or DHC has been solved. You have just shown how to convert a solution to DHC into a solution for UHC. This is called a *reduction* and is central to NP-completeness.

Lecture 28: NP-Completeness and Reductions

(Thursday, May 7, 1998)

Read: Chapt 36, through Section 36.4.

Summary: Last time we introduced a number of concepts, on the way to defining NP-completeness. In particular, the following concepts are important.

Decision Problems: are problems for which the answer is either "yes" or "no." The classes P and NP problems are defined as classes of decision problems.

P: is the class of all decisions problems that can be solved in polynomial time (that is, $O(n^k)$ for some constant k).

NP: is defined to be the class of all decision problems that can be *verified* in polynomial time. This means that if the answer to the problem is "yes" then it is possible give some piece of information that would allow someone to verify that this is the correct answer in polynomial time. (If the answer is "no" then no such evidence need be given.)

Reductions: Last time we introduced the notion of a reduction. Given two problems A and B, we say that A is *polynomially reducible* to B, if, given a polynomial time subroutine for B, we can use it to solve A in polynomial time. (Note: This definition differs somewhat from the definition in the text, but it is good enough for our purposes.) When this is so we will express this as

$$A \leq_P B.$$

The operative word in the definition is "if". We will usually apply the concept of reductions to problems for which we strongly believe that there is no polynomial time solution.

Some important facts about reductions are:

Lemma: If $A \leq_P B$ and $B \in$ P then $A \in$ P.

Lemma: If $A \leq_P B$ and $A \notin$ P then $B \notin$ P.

Lemma: (Transitivity) If $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.

The first lemma is obvious from the definition. To see the second lemma, observe that B cannot be in P, since otherwise A would be in P by the first lemma, giving a contradiction. The third lemma takes a bit of thought. It says that if you can use a subroutine for B to solve A in polynomial time, and you can use a subroutine for C to solve B in polynomial time, then you can use the subroutine for C to solve A in polynomial time. (This is done by replacing each call to B with its appropriate subroutine calls to C).

NP-completeness: Last time we gave the informal definition that the NP-complete problems are the "hardest" problems in NP. Here is a more formal definition in terms of reducibility.

Definition: A decision problem $B \in$ NP is *NP-complete* if

$$A \leq_P B \text{ for all } A \in \text{NP}.$$

In other words, if you could solve B in polynomial time, then every other problem A in NP would be solvable in polynomial time.

We can use transitivity to simplify this.

Lemma: B is NP-complete if

(1) $B \in NP$ and

(2) $A \leq_P B$ for some NP-complete problem A.

Thus, if you can solve B in polynomial time, then you could solve A in polynomial time. Since A is NP-complete, you could solve every problem in NP in polynomial time.

Example: 3-Coloring and Clique Cover: Let us consider an example to make this clearer. Consider the following two graph problems.

3-coloring (3COL): Given a graph G, can each of its vertices be labeled with one of 3 different "colors", such that no two adjacent vertices have the same label.

Clique Cover (CC): Given a graph G and an integer k, can the vertices of G be partitioned into k subsets, $V_1, V_2, \ldots, V_k$, such that $\bigcup_i V_i = V$, and that each V_i is a clique of G.

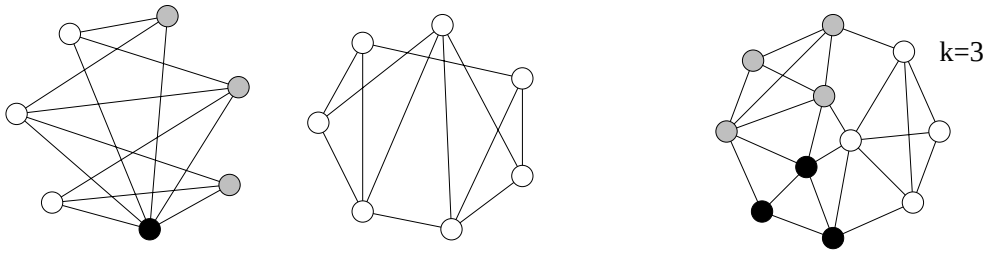

Figure 38: 3-coloring and Clique Cover.

Recall that the *clique* is a subset of vertices, such that every pair of vertices in the subset are adjacent to each other.

3COL is a known NP-complete problem. Your boss tells you that he wants you to solve the CC problem. You suspect that the CC problem is also NP-complete. How do you prove this to your boss?

There are two things to be shown, first that the CC problem is in NP. We won't worry about this, but it is pretty easy. (In particular, to convince someone that a graph has a clique cover of size k, just specify what the k subsets are. Then it is an easy matter to verify that they form a clique cover.)

The second item is to show that a known NP-complete problem (we will choose 3COL) is polynomially reducible to CC. To do this, you assume that you have access to a subroutine `CliqueCover(G, k)`. Given a graph G and an integer k, this subroutine returns true if G has a clique cover of size k and false otherwise, and furthermore, this subroutine runs in polynomial time. How can we use this "alleged" subroutine to solve the well-known hard 3COL problem? We want to write a polynomial time subroutine for 3COL, and this subroutine is allowed to call the subroutine `CliqueCover(G,k)` for any graph G and any integer k.

Let's see in what respect the two problems are similar. Both problems are dividing the vertices up into groups. In the clique cover problem, for two vertices to be in the same group they must be adjacent to each other. In the 3-coloring problem, for two vertices to be in the same color group, they must not be adjacent. In some sense, the problems are almost the same, but the requirement adjacent/non-adjacent is exactly reversed.

Recall that if G is a graph, then $\overline{G}$ is the *complement* graph, that is, a graph with the same vertex set, but in which edges and nonedge have been swapped. The main observation is that a graph G is 3-colorable, if and only if its complement $\overline{G}$, has a clique-cover of size $k = 3$. We'll leave the proof as an exercise.

Using this fact, we can reduce the the 3-coloring problem to the clique cover problem as follows. Remember that this means that, if we had a polynomial time procedure for the clique cover problem then we could use it as a subroutine to solve the 3-coloring problem Given the graph we want to compute the 3-coloring for, we take its complement and then invoke the clique cover, setting $k = 3$.

3COL to CC Reduction

```
bool 3Colorable(graph G) {
    let G' = complement(G)
    return CliqueCover(G',3)
}
```

There are a few important things to observe here. First, we never needed to implement the `Clique-Cover()` procedure. Remember, these are all "what if" games. If we could solve CC in polynomial time, then we could solve 3COL. But since we know that 3COL is hard to solve, this means that CC is also hard to solve.

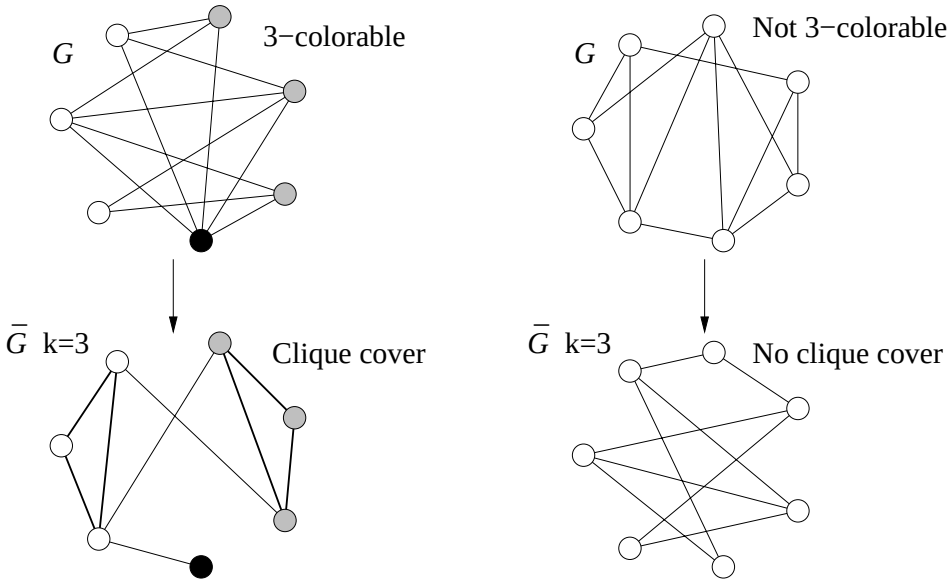

Figure 39: Clique covers in the complement.

A second thing to observe that is the direction of the reduction. In normal reductions, you reduce the problem that you do not know how to solve to one that you do know how to solve. But in NP-complete, we do not know how to solve either problem (and indeed we are trying to show that an efficient solution does not exist). Thus the direction of the reduction is naturally backwards. You reduce the known problem to the problem you want to show is NP-complete. Remember this! It is quite counterintuitive.

Remember: Always reduce the known NP-complete problem to the problem you want to prove is NP-complete.

The final thing to observe is that the reduction really didn't attempt to solve the problem at all. It just tried to make one problem look more like the other problem. A reductionist might go so far as to say that there really is *only one* NP-complete problem. It has just been dressed up to look differently.

Example: Hamiltonian Path and Hamiltonian Cycle: Let's consider another example. We have seen the Hamiltonian Cycle (HC) problem (Given a graph, does it have a cycle that visits every vertex exactly once?). Another variant is the Hamiltonian Path (HP) problem (Given a graph, does it have a simple path that visits every vertex exactly once?)

Suppose that we know that the HC problem is NP-complete, and we want to show that the HP problem is NP-complete. How would we do this. First, remember what we have to show, that a known NP-complete problem is reducible to our problem. That is, $\mathrm{HC} \leq_P \mathrm{HP}$. In other words, suppose that we had a subroutine that could solve HP in polynomial time. How could we use it to solve HC in polynomial time?

Here is a first attempt (that doesn't work). First, if a graph has't a Hamiltonian cycle, then it certainly must have a Hamiltonian path (by simply deleting any edge on the cycle). So if we just invoke the HamPath subroutine on the graph and it returns "no" then we can safely answer "no" for HamCycle. However, if it answers "yes" then what can we say? Notice, that there are graphs that have Hamiltonian path but no Hamiltonian cycle (as shown in the figure below). Thus this will not do the job.

Here is our second attempt (but this will also have a bug). The problem is that cycles and paths are different things. We can convert a cycle to a path by deleting any edge on the cycle. Suppose that the

graph G has a Hamiltonian cycle. Then this cycle starts at some first vertex u then visits all the other vertices until coming to some final vertex v, and then comes back to u. There must be an edge $\{u, v\}$ in the graph. Let's delete this edge so that the Hamiltonian cycle is now a Hamiltonian path, and then invoke the HP subroutine on the resulting graph. How do we know which edge to delete? We don't so we could try them all. Then if the HP algorithm says "yes" for any deleted edge we would say "yes" as well.

However, there is a problem here as well. It was our intention that the Hamiltonian path start at u and end at v. But when we call the HP subroutine, we have no way to enforce this condition. If HP says "yes", we do not know that the HP started with u and ended with v. We cannot look inside the subroutine or modify the subroutine. (Remember, it doesn't really exist.) We can only call it and check its answer.

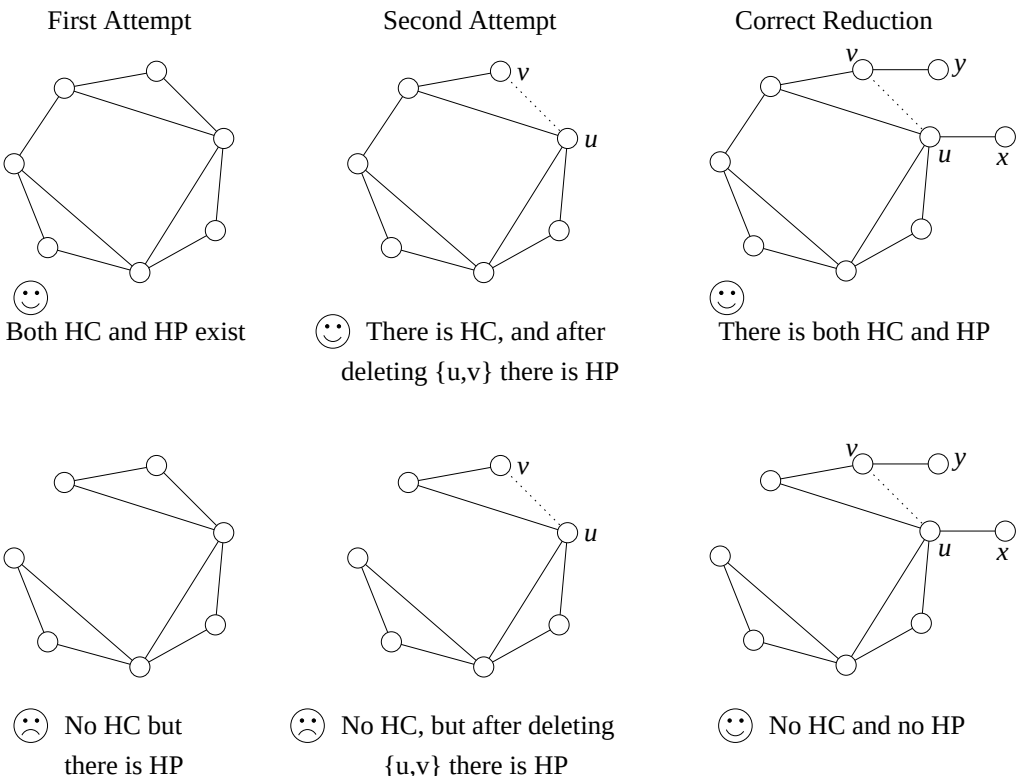

Figure 40: Hamiltonian cycle to Hamiltonian path attempts.

So is there a way to force the HP subroutine to start the path at u and end it at v? The answer is yes, but we will need to modify the graph to make this happen. In addition to deleting the edge from u to v, we will add an extra vertex x attached only to u and an extra vertex y attached only to v. Because these vertices have degree one, if a Hamiltonian path exists, it must start at x and end at y.

This last reduction is the one that works. Here is how it works. Given a graph G for which we want to determine whether it has a Hamiltonian cycle, we go through all the edges one by one. For each edge $\{u, v\}$ (hoping that it will be the last edge on a Hamiltonian cycle) we create a new graph by deleting this edge and adding vertex x onto u and vertex y onto v. Let the resulting graph be called G'. Then we invoke our Hamiltonian Path subroutine to see whether G' has a Hamiltonian path. If it does, then it must start at x to u, and end with v to y (or vice versa). Then we know that the original graph had a Hamiltonian cycle (starting at u and ending at y). If this fails for all edges, then we report that the original graph has no Hamiltonian cycle.

HC to HP Reduction

```
bool HamCycle(graph G) {
    for each edge {u,v} in G {
        copy G to a new graph G'
        delete edge {u,v} from G'
        add new vertices x and y to G'
        add new edges {x,u} and {y,v} to G'
        if (HamPath(G')) return true
    }
    return false                          // failed for every edge
}
```

This is a rather inefficient reduction, but it does work. In particular it makes $O(e)$ calls to the `HamPath()` procedure. Can you see how to do it with fewer calls? (Hint: Consider applying this to the edges coming out of just one vertex.) Can you see how to do it with only one call? (Hint: This is trickier.)

As before, notice that we didn't really attempt to solve either problem. We just tried to figure out how to make a procedure for one problem (Hamiltonian path) work to solve another problem (Hamiltonian cycle). Since HC is NP-complete, this means that there is not likely to be an efficient solution to HP either.

Lecture 29: Final Review

(Tuesday, May 12, 1998)

Final exam: As mentioned before, the exam will be comprehensive, but it will stress material since the second midterm exam. I would estimate that about 50–70% of the exam will cover material since the last midterm, and the remainder will be comprehensive. The exam will be closed book/closed notes with three sheets of notes (front and back).

Overview: This semester we have discussed general approaches to algorithm design. The goal of this course is to improve your skills in designing good programs, especially on complex problems, where it is not obvious how to design a good solution. Finding good computational solutions to problems involves many skills. Here we have focused on the higher level aspects of the problem: what approaches to use in designing good algorithms, how generate a rough sketch the efficiency of your algorithm (through asymptotic analysis), how to focus on the essential mathematical aspects of the problem, and strip away the complicating elements (such as data representations, I/O, etc.)

Of course, to be a complete programmer, you need to be able to orchestrate all of these elements. The main thrust of this course has only been on the initial stages of this design process. However, these are important stages, because a poor initial design is much harder to fix later. Still, don't stop with your first solution to any problem. As we saw with sorting, there may be many ways of solving a problem. Even algorithms that are asymptotically equivalent (such as MergeSort, HeapSort, and QuickSort) have advantages over one another.

The intent of the course has been to investigate basic techniques for algorithm analysis, various algorithm design paradigms: divide-and-conquer graph traversals, dynamic programming, etc. Finally we have discussed a class of very hard problems to solve, called NP-complete problems, and how to show that problems are in this class. Here is an overview of the topics that we covered this semester.

Tools of Algorithm Analysis: