Introduction to Algorithms	Prelim 2 Review
CS 4820, Spring 2011	April 6, 2011

1 General Thoughts on the Exam

The second exam will focus on two topics: network flow problems and NP-complete problems.

In studying network flow we focused on the basic definitions of a flow network, the equivalence of finding a max-flow to that of finding a min-cut, algorithms for computing max flows, and reducing problems to network flow problems.

For NP-completeness problems we looked at a class of problems known to be difficult and have worked to build a taxonomy of different looking but interrelated problems–NP-complete problems.

Both of these topics emphasize the usefulness of reductions, with a fundamental difference: *in studying network flow we are showing tractability of a problem by reducing it to a known problem, with NP-completeness we are considering intractability of a problem by showing that it is expressive enough to encode a problem known to be difficult.* In practical terms this has some impact on the way we construct reductions as discussed in the following section.

2 A Little Formalism on Reductions

The reductions you've done so far in the course have been done to show how to solve a problem, reductions for NP-completeness "go in the other direction" because they are used to show a problem is hard. We digress for a little formalism to make this point more concrete.

Given a problem A and a problem B, a polynomial time reduction r from A to B is a polynomial time algorithm from an input to problem A, a, to an input to problem B, $r(a)$, such that every solution to B of $r(a)$ corresponds to a valid solution to A on input a, and every valid solution to A on input a is a valid solution to B on $r(a)$. We write $A \leq_p B$, and $\leq_p$ is a partial ordering.

You can think of the notation $A \leq_p B$ as saying A is easier than or just as hard as B. Thinking of it this way, the difference for the direction of reduction in NP-completeness proofs makes sense. When we want to show a problem T is NP-complete, we want to show *it is at least as hard as* some NP-complete problem, ie $W \leq_p T$ where W is some NP-complete problem such as 3-SAT, vertex cover, etc. This requires a reduction from W to T. It would make no sense to construct a reduction from T to W as then we would show $T \leq_p W$ which would be saying T is easier than or just as hard as W. Remember, W is NP-complete, the most difficult type of problem we have encountered so far. Showing that an NP-Complete problem, W, is harder than some other problem you have encountered doesn't tell you much.

3 Flow Reductions

Flow reductions tend to be applicable for problems that involve assigning objects of type A to objects of another type B, like professors to committees or wins to baseball teams. In general,

the objects of type A must all be of the same size. (Unless you are allowed to split an object of type A into more than one piece and assign the pieces to different type B objects.) The load balancing problem, where you try to assign jobs of different sizes to different machines so that no machine has too large a load, is an NP-complete problem(described in 11.1), so trying to solve it by reducing to network flow will not work.

In being ready for flow problems for the exam, you'll want to:

1. Know the Definitions. If you don't understand how a residual graph behaves, you will find yourself in trouble. Other important concepts include cuts, cut capacities, residual graphs, augmenting paths, flow integrality, and the process by which you can get a min-cut from a graph on which you've constructed a max-flow.
2. Know how to evaluate Ford-Fulkerson on reasonable inputs. In general, examples you may have to work will require you to be able to calculate a max-flow in a flow network. Be sure you can do this. It is also good to be comfortable finding a minimum cut in a flow network after you have computed a max flow.
3. Keep in mind some of the lemmas that came up on the homework for flow related problems. Facts like those from Homework 5 problem 1 and 2 may not help you directly, but generally keeping some of the things we have proved about flow networks in mind will help give you good intuition for what is reasonable for a flow network.

3.1 Constructing the Reduction

This is frequently a fairly straightforward process of encoding constraints from the original problem to constraints that will be enforced by capacity constraints and conservation constraints in the flow reduction. When things get tricker you may have to construct gadgets. For example, in Homework 6 Problem 1 (Maxflowistan) these gadgets were needed to allow the generality of allowing students to go directly to a job or to prepare for a year after graduation at an agency, while enforcing employment constraints properly.

After constructing your reduction, don't forget to explain how to get a solution to the original problem back out of the reduction. What exactly does a max flow in this network mean? Are there conditions on that max flow? Does it need to be a certain value for some constraint in the original problem to be satisfied? You may think you have done all of the hard work in constructing the reduction, but there are still important pieces to be ironed out in relating your reduction back to the original problem.

3.2 Runtime

Be careful to express your runtime in terms of the original input to the problem. Be sure to know the runtimes of any flow problem you may wish to use to compute a max flow in your network. Know the runtime of Ford-Fulkerson and what the value C in its runtime means! Because C depends on the max flow in the graph, there are times when Ford-Fulkerson is a good choice, and times when it isn't. When it is be sure to back up why the max flow in a graph is bounded by something polynomial in the size of the input to the problem, and when it isn't be sure to know the runtime of some other flow algorithm like Edmonds-Karp or preflow-push.

3.3 Proof of Correctness

This is the same style for a proof of correctness as other reductions encountered during the semester. A valid max flow computed from your reduction corresponds to a valid solution to the original problem, a valid solution to the original problem corresponds to a valid flow in your network.

In practice this comes down to showing an equivalence between the satisfaction of constraints in the original problem with the capacity and conservation constraints in the flow network you construct for your reduction. Don't feel like your solution is wrong if you just feel like you are reading off facts about how you constructed your reduction. For many simple problems, the proof of correctness will be this straightforward. An example of this type of fairly straight-forward proof of correctness can be found in Problem 3 on Homework 5.

Another practical point to consider when doing the proof of correctness for your flow reductions is the notion of integrality. In particular, many reductions won't make sense if the flow values on edges are not integer valued. Examples of this would be Homework 5 Problem 3 and Homework 6 Problem 1. For example in Homework 6, Problem 1, it wouldn't make sense to try to assign half of a graduating student to an agency and half to a firm. You should be familiar with the discussion in section 7.2 about integer valued flows, and why their existence is guaranteed in the context of a problem like Homework 6 Problem 1. When doing a reduction, you should state the result that if all capacities are integer valued then you can find an integer valued max flow explicitly.

4 NP-Completeness Problems

There are five steps to showing a problem, Foo, is NP-complete.

1. Give a polynomial time verifier.
2. Give a reduction from some NP-complete problem $HARD$ to Foo.
3. Show your reduction takes polynomial time.
4. Show a solution in your reduction corresponds to a solution of the original problem.
5. Show a solution to the original problem corresponds to a solution to your constructed reduction.

Follows are comments on each step. Throughout these comments Foo will be some problem we are trying to show to be NP-complete, and $HARD$ will be some NP-complete problem.

4.1 Giving a Polynomial Time Verifier

The purpose of this step is to show that Foo is in NP. This means regardless of any difficulty there may be in giving an answer to an arbitrary instance of Foo, given a candidate for an answer, A, to Foo, it is possible to check in polynomial time that A is in fact a correct answer to the given instance of Foo.

This step entails giving an algorithm which takes an instance of Foo and a solution candidate, A, and outputs 'Yes' if A is a correct solution to the given instance of Foo and 'No' otherwise.

This step is usually not difficult.

4.2 Give a reduction from some NP-Complete problem to Foo

There is no attempt to be encyclopedic here. For a good sample of most of the NP-Complete problems we'll talk about and general thoughts on reduction involving them, look at section 8.10 titled A Partial Taxonomy of Hard Problems in Kleinberg and Tardos. However follows is a reminder about the general strategy, and afterwards we'll talk about some other tips and tricks.

First pick an NP-complete problem, $HARD$. In terms of the formalism discussed in section 1, we want to show that $HARD \leq_p Foo$ where Foo is the problem you are trying to prove NP-complete. From the notation you should notice that you need to reduce from $HARD$ to Foo, if you were to reduce from Foo to $HARD$ then you would be showing that $Foo \leq_p HARD$, this is silly. We know $HARD$ is hard, so showing $Foo \leq_p HARD$ doesn't really help us. The reduction takes the form described in section 1, namely an algorithm that takes an arbitrary instance to $HARD$ and transforms it so that Foo can solve it. Sometimes the transformation will be straightforward and will basically consist of mapping a type of object in one problem to a type of object in the other problem (see the posted dodgeball problem where we map a pair of team assignments to a color). Other times the reduction will require the construction of gadgets (like in the reduction from 3-SAT to Visit Day Scheduling). When reducing from 3-SAT in general, there will be gadgets for binary decisions, and gadgets for enforcing the three-strikes-and-you're-out rule for clauses.

Some other general tips on constructing your reductions:

1. If a problem asks you to decide if there exists a set of at least k objects satisfying some properties, try reducing from another problem that involves picking at least k objects, e.g. Independent Set or Clique.
2. Similarly, if a problem asks you to decide if there exists a set of at most k objects satisfying some properties, try reducing from another problem that involves picking at most k objects, e.g. Vertex Cover or Set Cover.
3. When a problem doesnt easily fit into any of the general categories listed above, the most useful starting point for reductions is the 3Sat problem. For some reason, its unusually easy to design gadgets reducing 3Sat to other problems that dont bear any obvious relation to Boolean variables and clauses.

4.3 Show your reduction takes polynomial time.

This should really be implicit with the notation $\leq_p$. Don't forget to give the runtime of your reduction in terms of the parameters given and not the general runtimes for any known algorithms you might use.

4.4 Show a solution to the original problem corresponds to a solution in your reduction.

Because NP-complete problems are decision problems, this can be restated as a "yes" instance of $HARD$ corresponds to a "yes" instance of Foo. This is generally the easier direction. Don't feel like you are doing the wrong thing if you feel like you are just reading off an explanation of how you constructed your reduction and how it ensures you will get a "yes" instance of Foo.

4.5 Show a solution to your reduction corresponds to a solution in the Problem.

Because NP-Complete problems are decision problems this can be restated as follows: a "yes" instance of Foo corresponds to a "yes" instance of $HARD$.

This can be a bit trickier. This section requires showing that in the reduction you construct, there is no unintended way of getting a "yes" instance of Foo that would correspond to something you couldn't translate back to a "yes" instance of $HARD$. While you are designing your reduction, particularly while creating gadgets for graph related reductions, you may find that you have some wiggle room in the number of nodes or edges in some places. For example in the detailed solution for Test Network Design, the only requirement on the number of "edge chain nodes" is that there are $n-k+1$ of them. Any polynomial number of "edge chain nodes" greater than this would have been fine and n was chosen because it is a cleaner number to talk about.

It is also important you consider carefully such sources of wiggle room. You test your assumptions about the numbers of various nodes and edges in this section. You may find that in trying to do this part of the problem that there was some corner case you missed, and you will want to go back and fix it. Also don't be afraid of simplicity. If in trying to prove something about your reduction you realize it would be cleaner to add a polynomial number of nodes, edges or whatever gadgets the problem calls for, go for it! Be sure to fix your reduction and your runtimes accordingly, but it is better to give a clearer correct solution and analyse it properly than try for really confusing and tight bounds. Remember you are trying to show a problem is hard! Having really fast reductions isn't that helpful.