Techniques for Proving NP Completeness

Winter 2023

When you cannot find an efficient algorithm for a problem, the best thing that you can do is often to prove that it is NP-Complete/NP-Hard. This provides reasonably strong evidence that an efficient algorithm may well not exist, and leaves you looking for other kinds of solutions. However, how does one prove such a thing?

1 Reductions

Reductions are used to prove that one problem is at least as hard as another. If you want to prove that problem B is at least as hard as A, you want to find a reduction from A to B. This means showing that *if* you had an efficient algorithm to solve B you could use it to solve A efficiently.

This usually amounts to finding a way to take an instance of A and encode it as an instance of B so that if you could solve that instance of B you could use the solution to solve the instance of A. Often for decision problems, this means finding an instance of B that has a solution if and only if the original instance of A does, usually because there is some way of relating a solution to the A-instance to a solution to the B-instance.

Sometimes this is relatively easy. For example, sometimes B is just more general than A and an instance of A is automatically an instance of B. For example, Knapsack where you can only take one copy of each item and Knapsack where the maximum number of copies of each item is specified. If you just specify the number of copies to be 1, you have the original problem again.

More frequently you will have to do some work. Often you want to build some kind of special set of instances of B where solutions will be of some relatively nice form. You will then need to find a way to add restrictions so that only solutions that correspond to solutions of A will work.

2 Proving NP-Completeness

To prove that a problem is NP-Complete or NP-Hard, you need to find a reduction to that problem from *some* other NP-Hard or Complete problem. The fact that you can chose the problem is quite powerful and finding the correct problem to reduce from can make your job much easier. For example, it is much easier to find a reduction to TSP from Hamiltonian Cycle than from Knapsack.

To prove NP-completeness, you will often want to find instances of your problem that encode some kind of logic. This means that you want to have "variables" with some kind of "constraints" on them. This means building some kind of instances where a solution will involve making several discrete choices and then forcing those choices to obey certain logical constraints (and then relating those constraints to some other NP-Complete problem).

Another idea that often shows up in reduction problems is the idea of a gadget. This is some portion of your instance that can only be properly solved in one of a few well understood ways. These can be used to implement variables in your problem or to implement some of the constraints you are looking for.

3 Examples

[[Please do something not covered in class. Maybe graph 3-colorability or minesweeper.]]