NP-Completeness notes
By Cole Frederick

Complexity classes

We can classify problems based on the best ways we know to solve them (the best algorithms we've discovered). We consider problems with known polynomial time algorithms to be “tractable”. Saying that another way, if we know of an algorithm that runs in polynomial time in the worst case for a problem, we say that the problem is tractable. If there is no possible polynomial time algorithm to solve some problem, we say that it is intractable. That is, we know for sure there never will be a polynomial time algorithm. (There is a nice discussion at this URL: http://www.cs.ucc.ie/~dgb/courses/toc/handout29.pdf.)

We can prove a problem is tractable by showing there is a polynomial time algorithm. We can prove a problem is intractable if we have a proof that no such algorithm could exist. There are many problems that have not been proved either way yet. The complexity class $P$ (Wikipedia definition: https://en.wikipedia.org/wiki/P_(complexity)) is the set of decision problems for which we have found polynomial time algorithms. Some of the problems not known to be in the class $P$ have a special property: an affirmative answer can be verified in polynomial time given a certificate. The complexity class $NP$ (Wikipedia definition: https://en.wikipedia.org/wiki/NP_(complexity)) is the set of decision problems for which we have found polynomial time algorithms for verifying affirmative answers given a certificate for the purported solution. An example:

![Graph](image)

We could ask if this graph contains a clique of size 4 or greater. We do not know of a worst-case polynomial time algorithm for answering this question for arbitrary graphs, so the clique decision problem is not in the class $P$. But if we find a clique of size 4 or greater in this graph, we can give a certificate of its existence, naming the nodes in the clique: 1, 2, 6, 7. To verify a clique certificate, we ensure each of the nodes in the purported clique has an edge to every other node. This can be done in polynomial time relative to the size of the graph. Therefore clique is in the class $NP$. 
All problems in P are also in NP; any problem with a polynomial time algorithm has an algorithm for verifying an affirmative answer given a certificate in polynomial time. The algorithm is this: throw away the certificate, solve it yourself in polynomial time. In this way, trivially, we see that P is a subset of NP.

**Reductions**

We say that some problem A is reducible to problem B if we know an algorithm that takes any instance of problem A and outputs an instance of problem B, such that the answer for the B instance can be easily used to determine the answer for the original A instance. That means that we can design an algorithm for solving problem A that goes like this: reduce the input to an instance of problem B, run an algorithm that solves problem B, use the answer to determine the answer to the original input. If the reduction process is relatively fast, then this algorithm's complexity is dominated by the complexity of the algorithm that solves problem B.

We can keep such an algorithm in our back pocket as we look for ways of solving problem A. Maybe we’ll find something really good, but at the very least we can always fall back to the reduction algorithm and solve A by using an algorithm for problem B. In this way we’ve established an upper bound on the complexity of solving problem A. Even if we can’t come up with anything better, we can always just reduce to problem B. So the complexity of the best algorithm for solving B is an upper bound on the complexity of the best algorithm for solving problem A. Or we can simply say that the complexity of B is an upper bound on the complexity of A.

Reductions are transitive, therefore so are the complexity bounds they imply. If A reduces to B, and B reduces to C, then we can reduce A to C. So the complexity of C is an upper bound on the complexity of A.

Saying that B is an upper bound on A’s complexity is equivalent to stating that A is a lower bound on B’s complexity.

**Decision vs Optimization**

All the problems we discuss in connection with NP-Completeness theory, members of class P and class NP, are decision problems. One of the goals of this area of theory is to give evidence that a problem is most likely intractable. In practice we normally care about optimization problems, not decision problems, and so we need to establish a relationship between them. A decision problem like “Is there a clique of size 4 or greater?” can be reduced to the optimization version of clique “What is the largest clique in this graph?” like this: find the largest clique by running the optimization algorithm, then compare the clique it returns to the target number (in the example, 4). Hence the complexity of a decision problem is a lower bound on the complexity of the corresponding optimization problem. So if we show that a decision problem is intractable, it means the optimization problem is also intractable.
NP-complete

The complexity class NP-complete (Wikipedia definition: https://en.wikipedia.org/wiki/NP-completeness) is the set of problems in NP with the property that their complexity is an upper bound on the complexity of all problems in NP (within a polynomial factor). They are as hard as any problem in NP (again, within a polynomial factor). The existence of this class of problems is proved by showing that any problem in NP is reducible to a target problem in NP. This target problem is the first NP-complete problem. If this first NP-complete problem is reducible to a second problem, then this problem is also NP-complete (since reduction is transitive). Any problems that the first NP-complete problem can reduce to (directly or transitively) are also NP-complete.

If we discover a new problem, problem X, and we want to show that it is NP-complete there are two steps. First, we must prove that X is in NP. We give an algorithm to verify a certificate in polynomial time. Next, we give a reduction from a known NP-complete to problem X. The reduction should be polynomial time or else it’s not useful to us. This shows that the complexity of X is an upper bound on the complexity of the known NP-complete problem. And since any NP-complete problem is an upper bound on the complexity of all problems in NP (within a polynomial factor), problem X is transitively an upper bound on the complexity of all problems in NP. So problem X is shown to be NP-complete.

If we instead gave a reduction from problem X to a known NP-complete problem (many people initially find this more intuitive), we would only have shown that the known NP-complete problem is an upper bound on the complexity of problem X. Since the first step showed that X was in NP, we already knew this.

Who cares?

Proving that any single NP-complete problem is tractable (discovering a polynomial time algorithm for it, proving it is in the class P) proves all problems in NP are tractable (P = NP). This is simply due to the fact that any algorithm for solving an NP-complete problem is an upper bound on the complexity of the best algorithm for solving all problems in NP (within a polynomial factor). So if one NP-complete problem gets a polynomial algorithm, by chains of reductions all problems in NP would have polynomial time algorithms.

Although we don't have a proof that P does not equal NP, it seems likely. Nobody has had luck with any of the NP-complete problems, which feels like mounting evidence that there is probably no luck to be had.

Recognizing that a class of problems is likely intractable helps us. Faced with a single problem to solve, we might be tempted to spend the day or the week approaching it from different angles, hoping that we might find a good method of attack not considered before. But faced with an NP-complete problem, we know that finding a “trick” to solving it efficiently would unlock the secret to giving efficient solutions to hundreds of problems that nobody has made progress on.
It’s more clear that this is a pipe dream, and so we explore alternatives and maybe even get out of the office early to beat rush hour traffic.

Summarizing from this discussion (again, http://www.cs.ucc.ie/~dgb/courses/toc/handout29.pdf), alternatives include:

- Use heuristics that tend to solve typical instances reasonably well. Even though exponential worst cases exist, it doesn’t mean you’ll be bumping into them all the time.
- Restrict your problem to a special case of it that is tractable
- Use probabilistic algorithms that are sometimes incorrect, but fast
- On optimization problems, use approximation algorithms that don’t find the very best answer, but are fast