CMSC 451: Lecture EXF
Final Review

The Final Exam will be **Thursday, May 15, 10:30am-12:30pm in IRB 0324** (*not* in our classroom). The exam will be closed-book and closed-notes, but you are allowed 2-sheets of notes, front and back.

Overview: This semester we have discussed general approaches to algorithm design. The intent has been to investigate basic algorithm design paradigms: depth-first search, greedy algorithms, dynamic programming, etc. And to consider how these techniques can be applied on a number of well-defined computational problems. We also studied network flow and the related problems of cuts and circulations. We studied the class of NP-complete problems, which are widely believed not solvable in polynomial time, but no one knows for sure. Finally we discussed studied methods for developing approximation algorithms for optimization problems, with a special focus on NP-hard problems.

How to use this information: The algorithms we have studied this semester are not always directly applicable in practice. Real world problems are often much messier, and have many domain-specific constraints. Nonetheless, I hope that the methods that we have discussed this semester will provide you with a strong foundation for tackling such problems. There are a number of important lessons to take away from this course.

Develop a clean mathematical model: Most real-world problems are messy. An important first step in solving any problem is to produce a simple and clean mathematical formulation. For example, this might involve describing the problem as an optimization problem on graphs, sets, or strings. If you cannot clearly describe what your algorithm is supposed to do, it is very difficult to know when you have succeeded.

Create good rough designs: Before jumping in and starting coding, it is important to begin with a good rough design. If your rough design is based on a bad paradigm (e.g. exhaustive enumeration, when depth-first search could have been applied) then no amount of additional tuning and refining will save this bad design.

Prove your algorithm correct: Many times you come up with an idea that seems promising, only to find out later (after a lot of coding and testing) that it does not work. Prove that your algorithm is correct before coding. Writing proofs is not always easy, but it may save you a few weeks of wasted programming time. If you cannot see why it is correct, chances are that it is not correct at all.

Can it be improved? Once you have a solution, try to come up with a better one: faster, simpler, or more general. Is there some reason why a better algorithm does not exist? (That is, can you establish a lower bound?) If your solution is exponential time, then maybe your problem is NP-hard.

Prototype to generate better designs: We have attempted to analyze algorithms from an asymptotic perspective, which hides many of details of the running time (e.g. constant factors), but give a general perspective for separating good designs from bad ones. After you have isolated the good designs, then it is time to start prototyping and doing

empirical tests to establish the real constant factors. A good profiling tool can tell you which subroutines are taking the most time, and those are the ones you should work on improving.

Still too slow? If your problem has an unacceptably high execution time, you might consider an approximation algorithm. The world is full of heuristics, both good and bad. You should develop a good heuristic, and if possible, prove a ratio bound for your algorithm. If you cannot prove a ratio bound, run many experiments to see how good the actual performance is.

There is still much more to be learned about algorithm design, but we have covered a great deal of the basic material. One direction is to specialize in some particular area, e.g. string pattern matching, computational geometry, parallel algorithms, randomized algorithms, or approximation algorithms. It would be easy to devote an entire semester to any one of these topics.

Another direction is to gain a better understanding of average-case analysis, which we have largely ignored. Still another direction might be to study numerical algorithms (as covered in a course on numerical analysis), or to consider general search strategies such as simulated annealing. Finally, an emerging area is the study of algorithm engineering, which considers how to design algorithms that are both efficient in a practical sense, as well as a theoretical sense.

Material for the final exam:

Old Material: Know general results, but I will not ask too many detailed questions. Do not forget about the material from before the midterm (asymptotics, DFS/BFS, greedy algorithms, greedy approximations). There may be an algorithm design problem that will involve one of these techniques.

Dynamic Programming: Although dynamic programming was covered on the midterm, it is an important technique, and you may be asked to devise an algorithm using DP on the final.

Network Flow: Know the many concepts that we introduced, such as s-t network, flow, residual network, augmenting path, and cuts. Also remember the basic Ford-Fulkerson algorith. Know the Max-Flow/Min-Cut Theorem, and how to apply it. Expect to answer questions that involve reducing problems to network flow (or variations, such as the circulation problem).

NP-completeness:

Basic concepts: Decision problems, polynomial time, the class P, the class NP, polynomial time reductions.

NP-completeness reductions: You are responsible for knowing the following reductions.

- 3-coloring to Clique-Cover
- 3SAT to Independent Set (IS)
- IS to Vertex Cover (VC) and IS to Clique

- VC to Dominating Set (DS)
- 3SAT to Directed Hamiltonian Cycle (DHC)
- VC to Subset Sum (SS)

It is also a good idea to understand any reductions that appeared in the homework solutions, since modifications of these may appear on the final.

Further tips: NP-complete reductions can be challenging. If you cannot see how to solve the problem, here are some suggestions for maximizing partial credit. All NP-complete proofs have a very specific form. Explain that you know the template, and try to fill in as many aspects as possible. Suppose that you want to prove that some problem B is NP-complete.

- B $\in$ NP. This almost always easy, so don't blow it. This basically involves specifying the certificate (that is, the guess), and then giving a polynomial-time algorithm to verify the problem specifications.
- For some known NP-complete problem A, prove that A $\leq_P$ B. This means that you want to find a polynomial time function f that maps an instance of A to an instance of B. (Make sure to get the direction correct!)
- Show that your reduction is correct, by showing that $x \in$ A if and only if $f(x) \in$ B. First, assuming that you have a solution to x, show how to map this to a solution for $f(x)$. Next, assuming that you have a solution to $f(x)$, show how to map this to a solution for x.

If you are stuck coming up with a function f, think about what you would like f to do. Explain which elements of problem A will likely map to which elements of problem B. Remember that you are trying to translate the elements of one problem into the common elements of the other problem.

Approximation: We discussed the concept of approximations and approximation ratios. We presented the following:

- A factor-2 approximation for Vertex Cover
- Two approximations for the metric TSP problem: A factor-2 approximation based on the twice-around tour of the MST and Christofides factor-(3/2) approximation based on combining this with a minimum-weight matching.
- A polynomial-time approximation scheme (PTAS) for the subset sum problem. Recall that this involved starting with an exact DP solution, and repeatedly pruning the resulting lists of sums, keeping only values that are sufficiently distinct in value.