CSCI3390-Lecture 0

August 28, 2018

1 A sampler of computational problems

Problem 1. I give you an undirected graph and two vertices, and ask for the shortest path between these two vertices.

In the example illustrated in the figure, there is a path of length 3 from 1 to 6: 1,2,5,6. And another path of length 3: 1,3,4,6. But there are not paths of length 2, so both of these paths are solutions to the problem.

As you may know, there is a nice algorithm for solving this problem called *breadth-first search*. Place the start vertex in a queue, and mark that vertex as visited. At each step, remove the vertex at the head of the queue, and add its unmarked neighbors to the rear of the queue, marking them as visited. Each edge of the graph is examined at most twice, and there is a little bit of additional book-keeping each time a vertex is placed in the queue. So this algorithm is *fast*, even for very large graphs, since its running time is roughly proportional to the 'size' of the graph (total number of vertices and edges).

Problem 2. What if I asked for the *longest* path from 1 to 6? We have to qualify this a bit, because if we allow the path to run around a loop or retrace its steps, we can make it as long as we like: 1,2,5,4,3,1,2,5,4,3,...,1,2,5,6. To avoid this, we should restrict to paths in which no vertex is repeated. With this understanding, in the pictured graph 1,3,4,5,6 and 1,2,5,4,6 are both longest paths.

Is there a fast algorithm for solving this problem, analogous to the one in Problem 3? There is a *dumb* algorithm: List all sequences starting at 1 and ending at 6 with no repeated vertex, and check to see if any of them is a path. The problem is that as the number $|V|$ of vertices grows larger and larger, the number of sequences grows very rapidly, something like $|V|!$ (that's not an exclamation, it's a factorial). For a graph with 100 vertices, our shortest path algorithm requires

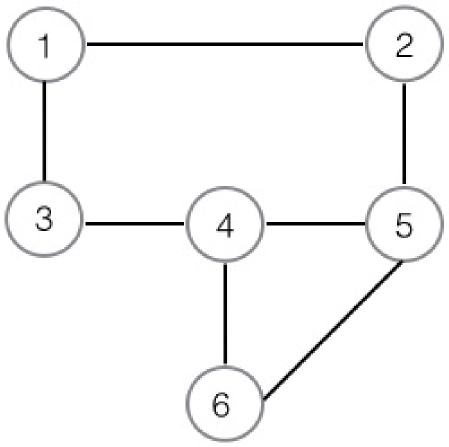

Figure 1: What is the shortest path from vertex 1 to vertex 6? What is the longest?

a couple of hundred steps, and on a typical personal computer will give the answer apparently instantaneously. The dumb algorithm for longest path could require tabulating $100! \approx 10^{158}$ sequences, which is about the *square* of the estimated number of atoms in the observable universe. Can we do better? can we do as well as Problem 3? Of can we prove that we *can't* do that well? (Solve this problem and you may win a million-dollar prize.)

Problem 3. I have a computer program, written in Python, or whatever your favorite language is. I would like to eliminate 'dead code'–that is, statements that can never be reached in any possible execution. Is it possible to write a program that checks programs for dead code, answering 'Yes' if the input program contains such code, 'No' if it doesn't?

Problem 4. (Known as *Hilbert's Tenth Problem.*) I give you a polynomial in several variables with integer coefficients (What does this mean? Just start with the variables and repeatedly add, subtract, multiply.) Determine if there are *integer* values of the variables that make the polynomial zero. For example, determine if there are any solutions to an equation like:

$$y^2 - 4z^3 + 1 = 0.$$

In this case, the answer is 'no' (can you prove that?) But in other cases, like

$$2xy^2 = z^3,$$

the answer is yes (for example, $x = 2, y = z = 4$ is a solution). Is there a general procedure for answering this question?

What might such a 'general procedure' look like? If the polynomial has only one variable, then the highest-degree term dominates the value, and we can use this fact to show that any solution must occur in a finite interval. For example, in

$$y^3 + 6y^2 - 4y + 2 = 0$$

we have $|6y^2 - 4y + 2| \leq 12y^2$ and $|y^3| > 12y^2$ if $|y| > 12$. In particular $y^3 + 6y^2 - 4y + 2$ cannot be zero if $|y| > 12$, so we only have to check all the integers between -12 and 12 inclusive to determine if there are any integer solutions. This method works for *any* polynomial with *one* variable. Is there something like it for polynomials in several variables?

Problem 5. I give you a large integer N and ask you to determine whether it is a prime, or, if it is not, to find its prime factorization. Here is a brute-force algorithm: Divide N by 2,3,4,5,... until you find a divisor or the candidate divisors exceed $\sqrt{N}$. In the latter case N is prime, in the former, we have found a prime factor, and can repeat with the quotient by this factor.

If N is a really large number, say with 200 decimal digits, then this algorithm requires about 10^{100} divisions to establish that N is prime, and we're back in the number-of-atoms-in-the-universe dilemma. Is there a better method?

There are actually two different problems here: determining if an integer is prime, and determining the prime factorization. And the answers to the 'better method' question are different for these two problems. We can determine if N is prime quickly—the method gives *no* information about the factorization of N if it turns out that N is composite, and the *best* method is a 'randomized' algorithm that flips coins. For the factorization problem, the situation is not quite as bad as the brute-force algorithm, but the best algorithms can barely factor 200-digit integers, and doing so requires the equivalent of hundreds of years of processor time. (The work is distributed among many different processors.)

This is not just a theoretical question: the easiness of testing for primality, and the hardness of factoring, are the crucial ingredients of practical cryptographic methods that we use every time we log into a secure website.

Problem 6. An algorithm that harnesses the power of 1000 supercomputers working co-operatively runs faster than the same algorithm on your laptop, which runs faster than the same algorithm on a first-generation 8-bit microcomputer from the 1970's, which in turn is faster than the vacuum-tube computers of 1950, themselves faster than the relay-based computers built in the late 1930's, which would have left the mechanical computers Charles Babbage dreamed of in the 1840's in the dust. But all these methods are subject to the same limitations when they execute an algorithm whose running time grows very rapidly with respect to the input size. If the best you can do with the brute-force algorithm for longest path today is a graph with 20 vertices, and you make your computers a million times times faster, you might get up to 25 vertices. There is a sense in which all of these computers work the same way, and are subject to similar limitations.

But..is there a different kind of machine? That runs a different kind of algorithm? Computer scientists have dreamed up (but not yet built) quantum-mechanical computers that, because they work on quite different physical principles, execute a different kind of algorithm. Quantum computers can factor numbers rapidly (and break all those cryptographic systems).

2 Themes

1. What is an algorithm? Can *every* computational problem be solved by an algorithm? It turns out that our dead code problem, and our polynomial equation problem, cannot. But how would you ever prove such a thing?
2. A surprisingly related question: Can every true mathematical statement be proved? The answer, in a sense we can make precise, is no.
3. What is an efficient algorithm? Do the longest-path problem and the factorization problem have efficient algorithms? If they don't, how would you prove that?
4. What happens if we allow computer programs to flip coins (as in the primality-testing methods mentioned above)? Can we make programs more powerful by incorporating randomness?
5. Can computers built on different physical principles do more than present-day computers?