COS 226 **Precept 6 Notes 2** September 10, 2021
Fall 2021
Tarjan

1 Percolation reprise

As a follow-up to the question of whether there is a simple formula for the percolation threshold, Kevin pointed out to be that for bond percolation in a square grid it is known that the threshold is ½, but this is not so easy to prove. In fact, we can use a symmetry argument of the kind we explored on September 3 to prove that the threshold is at least ½; the hard part is proving that the threshold is exactly ½.

Let me say a bit more about this. In bond percolation, all the sites (squares) are open, but the sides of the squares (the bonds) are not. Each bond is open with independently with probability p. We are interested in the value of p, call it p^* (called p_t in the previous set of notes) such that the system percolates with probability at least ½. *Percolate* means that there is a path from above the grid across an open bond into the grid, then across open bonds through the grid, and finishing by crossing an open bond at the bottom of the grid. As shown in the attached figure, there is a duality as in the hexagonal grid case, not for a square grid, but for an n-row, n+1-column grid: the grid percolates top-to-bottom if and only if the dual grid, which has n+1 rows and n columns, does not percolate left-to-right. The "only if" part uses the Jordan curve theorem as we have already seen; the "if" part uses a proof like that for the hexagonal grid in the notes on the game of Hex.

Incidentally, the square grid and its dual as shown in the figure are the basis of another game I used to play as a child, called "Bridg-It": the blue and red players alternate choosing unclaimed bonds, with the red player opening them and the blue player closing them. The goal of the blue player is to form a blue top-to-bottom path; that of the red player, to form a red left-to-right path. Unfortunately, not only is this game a win for the first player by a "strategy stealing" proof like that for Hex, but there is a simple winning strategy for the first player. (Can you find it?) Thus Hex is much more interesting to play

We conclude by symmetry that the bond top-to-bottom percolation threshold for an n by n+1 grid is exactly ½. It follows that for any fixed-size square grid, the bond percolation threshold is greater than 1/2, since the extra row makes percolation less likely. The hard part is to prove that the extra row has less and less effect as n increases. That is, for any p > ½, there is a large enough n such that the percolation threshold of an n by n grid is at most p. This is not surprising, and it is true, but its proof is as they say "beyond the scope of this class."

2 Counting iterations of nested loops

Turning to topics we covered on September 10 and in the lectures on the 7th and 9th, let me begin with a few comments about counting iterations of nested loops. A common form of such loops (which you saw in lecture) is a nested loop that iterates over pairs i, j, both of which are at least 0, at most n – 1, and such that $i < j$. A more complicated example is a triple nested loop that iterates over triples i, j, k, all at least 0, at most $n - 1$, and such that $i < j < k$. How many such pairs or triples are there? That is, how many iterations of the inner loop are there?

Counting such pairs or triples is a classical combinatorial problem that you will study in COS240 or 340, but let's quickly review the answer here. Counting such objects has two parts. Let's start with counting pairs. We know that i and j must be distinct. How many ways are there of choosing i and then choosing a distinct j? There are n ways to choose i. For each choice of i, there are exactly $n - 1$ ways to choose j, since one choice, that of $j = i$, is excluded. The total number of choices of i and j is thus $n(n - 1)$. But not all of these choices satisfy $i < j$. Indeed, for any particular pair of distinct integers in the desired range, say 5 and 7, there are two ways to obtain it: we can choose $i = 7$ and $j = 5$, or $i = 5$ *and* $j = 7$. In this case only the second choice satisfies the requirement that $i < j$. This is true in general: for each pair of distinct integers, there are two ways to obtain it, exactly one of which satisfies the additional requirement that $i < j$. We conclude that we must divide our original estimate by 2 to obtain the correct count: $n(n - 1)/2$. In combinatorics-ese, this is "n choose 2," the number of ways to choose 2 distinct items out of n. In tilde notation it is $\sim n^2/2$.

The principle is the same to count ordered triples of distinct integers in a given range: there are n ways to choose i; for each of these, there are $n - 1$ ways to choose j distinct from i; for each choice of i and j, there are $n - 2$ ways to choose k distinct from both i and j. But in doing the choosing this way, we obtain each possible triple of distinct integers some number of times, in only one of which are the integers in increasing order. How many times? Given a triple of distinct integers, there are 3 ways to choose which is first. Having made this choice, there are only 2 ways to choose which is second. The unchosen integer must be third. This gives us 3 times 2 = 6 possibilities, only one of which has the integers in increasing order. We conclude that the number of distinct triples in increasing order is $n(n - 1)(n - 2)/6$. In tilde notation it is $\sim n^3/6$. Using the combinatorialists' notation $1(2)3(4)5\ldots n = n!$ (*n factorial*), the formula for the number of k-length strictly increasing sequences of integers between 0 and $n - 1$ (inclusive) is $n!/k!$. This is a useful formula to remember.

In one of the problems on the precept lesson we saw another interesting example that takes a bit more work to analyze. In that example the outer loop variable i starts at 1, increases by 1, and ends at n. The inner loop variable j starts at 1, increases by j, and ends when j is at least n. (I may be off by 1 here, but this won't affect what I have to say.). How many iterations are there of the inner loop?

If we understand what is going on, it is easy to at least write down a sum that estimates the number of iterations: we determine the number of iterations of the inner loop during the i^{th} iteration of the outer loop, and sum over i. The number of iterations of the inner loop when $i = 1$ is n. The number of iterations of the inner loop when $i = 2$ is $n/2$, since j increases by 2 each time. This estimate may be off by 1, but this won't matter if we only want a tilde estimate, as we shall see. The number of iterations of the inner loop when $i = 3$ is $n/3$ (to within 1). And so on. The number of iterations of the inner loop when $i = n$ (the last iteration of the outer loop) is 1. Summing over I, we find that the total number of iterations is $n + n/2 + n/3 + \ldots + n/n$.

Factoring out n, this sum is $n(1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n)$. The inner sum is what is called H_n, the n^{th} "Harmonic number." The mathematicians among you (and those who remember the appropriate slide in lecture) will know that H_n is roughly ln n, the natural logarithm of n. We conclude that the total number of iterations is $\sim n \ln n$. Since ln n grows with n, our error of at most 1 per value of i, totaling at most n over all iterations of the outer loop, is a lower-order term and does not affect our estimate if we use tilde notation.

What if we didn't know that the sum of the reciprocals of the first n integers is roughly ln n? The standard way to prove this is to use integral calculus, as Kevin mentioned in lecture. You don't have to prove this, but you may find it useful to memorize the formula.

3 Coupon collecting

An important setting in which the Harmonic numbers occur is the coupon collector's problem. Here is one version. To sell cereal, a cereal company runs a contest. In each box of cereal it puts a coupon. There are n different kinds of coupons. If you buy a box of cereal, it will contain one coupon of one of these kinds. If you collect one coupon of every kind, you win the grand prize! Assuming that the cereal company puts the same number of each kind of coupon in its boxes, one per box, how many boxes of cereal can you expect to buy before collecting at least one of each kind of coupon?

Let's solve this problem. Let's call a coupon useful if you don't have one of its type yet. Initially you have no coupons, so any coupon is useful, and you get a useful coupon in your first box of cereal. Once you have a useful coupon, others of the same kind are no longer useful. Thus your chance of getting a useful coupon in your second box is $(n - 1)/n$. The chance remains the same on the third and later boxes, until you get a coupon of a second kind. Once this happens, the chance of getting a third useful coupon drops to $(n - 2)/n$. More generally, if you have i useful coupons, the probability of getting a useful one (of a kind different from the ones you already have) in your next cereal box is $(n - i)/n$.

If your chance of getting a useful coupon is $p = (n - i)/n$, how many boxes of cereal on the average will you have to buy to get a useful coupon? The answer is $1/p = n/(n - i)$. This is a standard result from elementary probability that we could easily derive, but I'll leave it to you to derive it or look it up. Given this result, the total number of boxes we expect to open before collecting all n kinds of coupons is 1 for the first one plus $n/(n - 1)$ for the second one plus $n/(n - 2)$ for the third one plus…plus $n/1$ for the last one. This is exactly the sum we had at the end of the last section: you can expect to open $nH_n \sim n \ln n$ boxes before collecting all n kinds of coupons.

Perhaps not so coincidentally, the coupon collector's problem occurs in the percolation assignment. You need to open sites one at a time in random order. The obvious way to do it is to generate a random number between 1 and n (if there are n sites) and open site number n if it is not already open. This method does extra work, because as sites are opened the chance of selecting a site that is already open decreases, so you need more tries to open a new site. Indeed, if you had to open all the sites, you would be in exactly the coupon collector's situation. Fortunately, you only have to open about .58 of the sites before percolation occurs, so on each try you have at least a .42 chance of selecting a closed site and opening it, which means that at most 1/.42 ~ 2.4 tries are needed on average to open a new site. It only takes this many tries near the end of the process; at the beginning, when very few sites are open, the average number of tries will be close to 1. (As an exercise, you might compute the total number of tires needed to open say 60% of the sites.) The unsuccessful tries only cost a small constant factor in time. On the other hand, if the assignment required you to open all the sites, the unsuccessful tries would cost you a logarithmic factor in time, too much to pay. In this case it would be better to sample in a way that gives a new site on every try. How would you do this? One part of this week's programming assignment provides the answer: use a randomized queue.

3 Average case versus worst case versus randomized algorithms

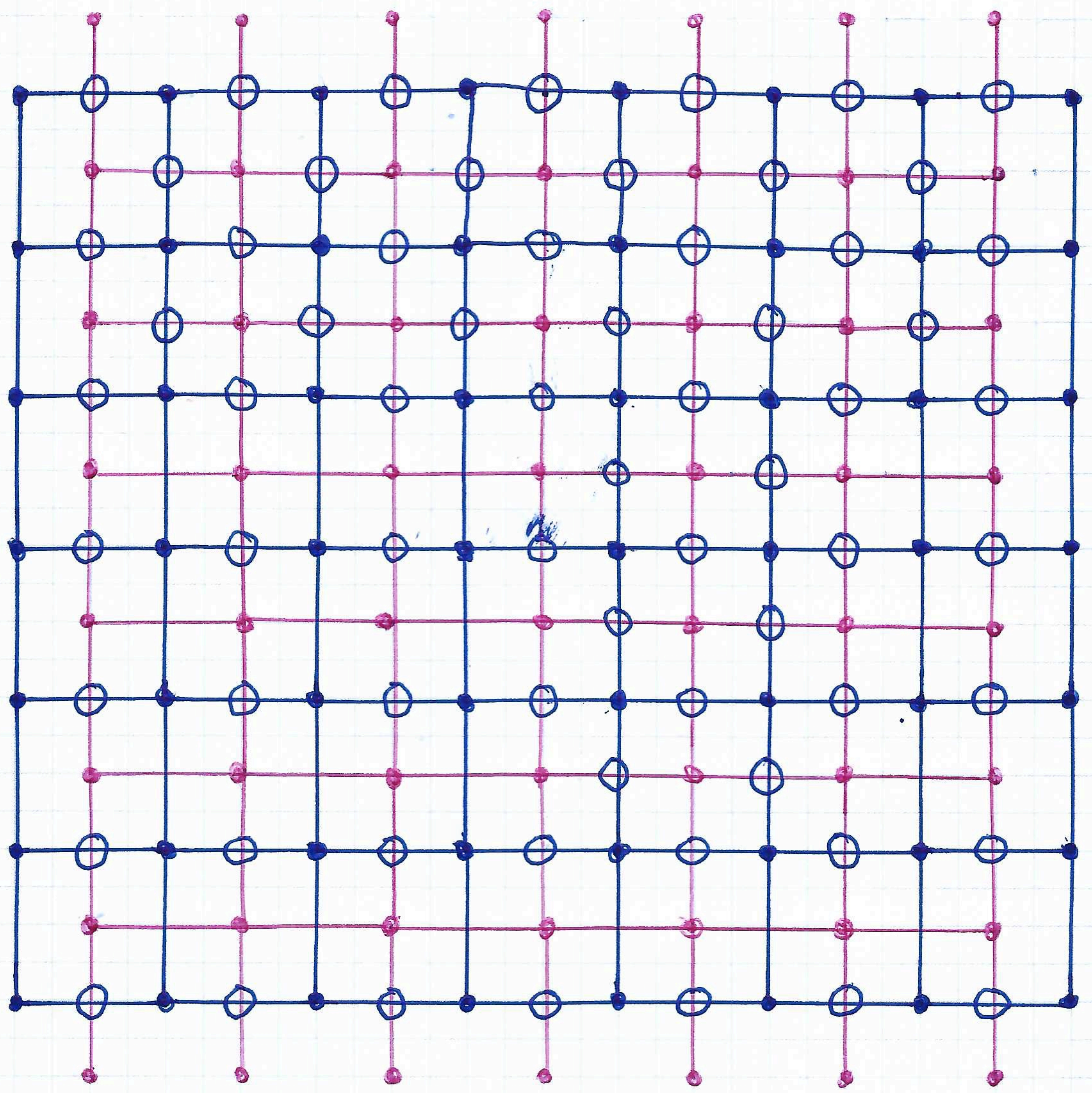
Simple algorithms, such as those consisting of nested loops with a number of iterations that depend only on some single parameter n, use time and space that depend only on n. But many algorithms have more complicated inputs, and the running time and space usage depend not only on the size of the input (n) but also on the details of the input. For example, soon we shall study sorting algorithms, which take a sequence of n numbers and rearrange the sequence into sorted order. The running time of such an algorithm can depend dramatically on the original arrangement of the numbers. For example, the QuickSort algorithm can sort n numbers in $\Theta(n)$ time in the best case but takes $\Theta(n^2)$ time in the worst case. Should we use the best case or the worst case, or something else, to measure the efficiency of QuickSort?

For many algorithms we shall use the worst-case running time as a function of the input size as our measure of efficiency. Worst-case bounds have the advantage that they provide a performance guarantee: no matter how bad the input, the running time will not exceed the worst-case bound. Obtaining a worst-case bound requires us to understand the worst case, but it does not require understanding how the algorithm behaves on every possible input.

Instead of the worst case, one might ask, "What happens in the average case?" To make "average case" meaningful, we must know something about the distribution of the possible input instances, or we must make an assumption about this distribution. For certain kinds of problems, sorting in particular, it is reasonable to do this: we assume that the input sequence is a uniformly random permutation of the input numbers. Under this assumption, QuickSort sorts n numbers in $\Theta(n\log n)$ time, as we shall see, and this is a more reasonable estimate of the efficiency of QuickSort than the worst-case bound, since bad cases are rare. (Note, though, that for certain versions of quicksort natural cases, such as an input sequence that is already sorted, are worst case.)

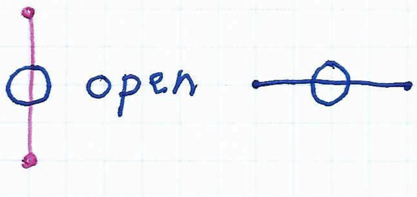
An average-case analysis may be more accurate than a worst-case analysis, but it can be harder to obtain, since it requires estimating the efficiency of the algorithm on every possible input, multiplying this estimate by the probability of the input, and summing. Furthermore such an analysis is only as accurate as our model of the input distribution. For problems on graphs and strings, we may not be able to obtain an accurate model of the input. Worst-case bounds provide much more robust efficiency estimates.

There is another way to use average-case analysis in algorithm design: allow the algorithm itself to be randomized, to “flip coins” or “roll dice” if you will, and make decisions based on the outcomes. By making the algorithm randomized, the algorithm designer can obtain the benefits of the average case without needing any assumptions about the input distribution. As an example, if pivots in QuickSort are chosen uniformly at random, QuickSort runs in expected $\Theta(n\log n)$ time on *any* input sequence. Depending on the problem, the best randomized algorithm can be simpler, or faster, or both (or neither) than the best deterministic algorithm.



Bond percolation on an n-row, n+1-column grid

Red N-S path = percolation



Red NS path iff no Blue WE path