21.1 2SAT

We begin by showing yet another possible way to solve the 2SAT problem. Recall that the input to 2SAT is a logical expression that is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_1}).$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied– that is, there is at least one true literal in each clause. For example, the assignment $x_1 = T, x_2 = F, x_3 = F, x_4 = T$ satisfies the 2SAT formula above.

Here is a simple randomized solution to the 2SAT problem. Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied or you get tired of flipping coins.

In the example above, when we begin with all variables set to F, the clause $(x_1 \vee x_2)$ is not satisfied. So we might randomly choose to set x_1 to be T. In this case this would leave the clause $(x_4 \vee \overline{x_1})$ unsatisfied, so we would have to flip a variable in the clause, and so on.

Why would this algorithm tend to lead to a solution? Let us suppose that there is a solution, call it S. Suppose we keep track of the number of variables in our current assignment A that match S. Call this number k. We would like to get to the point where $k = n$, the number of variables in the formula, for then A would match the solution S. How does k evolve over time?

At each step, we choose a clause that is unsatisfied. Hence we know that A and S disagree on the value of at least one of the variables in this clause– if they agreed, the clause would have to be satisfied! If they disagree on both, then clearly changing either one of the values will increase k. If they disagree on the value one of the two variables, then with probability 1/2 we choose that variable and make increase k by 1; with probability 1/2 we choose the other variable and decrease k by 1.

Hence, in the worse case, k behaves like a *random walk*– it either goes up or down by 1, randomly. This leaves us with the following question: if we start k at 0, how many steps does it take (on average, or with high probability) for k to stumble all the way up to n, the number of variables?

We can check that the average amount of steps to walk (randomly) from 0 to n is just n^2. In fact, the average amount of time to walk from i to n is $n^2 - i^2$. Note that the time average time $T(i)$ to walk from i to n is given by:

$$\begin{aligned} T(n) &= 0 \\ T(i) &= \frac{T(i-1)}{2} + \frac{T(i+1)}{2} + 1, \; i \geq 1 \\ T(0) &= T(1) + 1. \end{aligned}$$

These equations completely determine $T(i)$, and our solution satisfies these equations!

Hence, on average, we will find a solution in at most n^2 steps. (We might do better– we might not start with all of our variables wrong, or we might have some moves where we must improve the number of matches!)

We can run our algorithm for say $100n^2$ steps, and report that no solution was found if none was found. This algorithm might return the wrong answer– there may be a truth assignment, and we have just been unlucky. But most of the time it will be right.

Specifically, one lower bound we can put on the probability it will be right is $\frac{99}{100}$: if it has probability more than $\frac{1}{100}$ of needing to run more than $100n^2$ steps before finding an answer, that alone would make the expected number of steps more than n^2, contradicting the calculation above. (This argument is called "Markov's Inequality".)

We can prove a better bound on the error probability without changing the algorithm: for instance, in the first $4n^2$ steps, the probability that we find a satisfying assignment (if one exists) is at least $\frac{3}{4}$, by the same argument. In the second $4n^2$ steps, the probability that we find a satisfying assignment is again at least $\frac{3}{4}$, so the probability that we haven't found a satisfying assignment is at most $\frac{1}{4}\frac{1}{4}$, that is, the probability that the algorithm finds a satisfying assignment is at least $\frac{15}{16}$, better than the bound of $\frac{7}{8}$ from Markov's Inequality on the first $8n^2$ alone. Similarly, by dividing the steps into 25 groups of $4n^2$, in each of which the probability of finding a satisfying assignment is at least $\frac{3}{4}$ (if one exists), we can see that the probability of finding a satisfying assignment in $100n^2$ steps is at least $1-(\frac{1}{4})^{25}$, which is close enough to 1 that we should worry more about hardware failure, meteor strikes, and the like than the error introduced by the algorithm.

21.2 Application to 3SAT

Unfortunately, the natural generalization of this lecture's random walk algorithm to 3SAT may need to run for exponentially many steps before it has a high probability of finding a satisfying assignment (if one exists), so this doesn't solve one of the biggest open problems in computer science. (Specifically, this would show that the class NP of problems checkable in polynomial time equalled the class RP ("randomized polynomial time"), which is not quite the same as P = NP, the actually-biggest open question in computer science, but which modern computer scientists disbelieve just as strongly.

The problem is that, if we take a clause unsatisfied by an assignment of truth values and flip a random one of its variables to satisfy the clause, we may have only a $\frac{1}{3}$ chance of flipping a variable to match a satisfying assignment to the whole clause, and a $\frac{2}{3}$ chance of flipping away from a variable that already matched a satisfying assignment to the whole clause. So we may move *away* from a satisfying assignment, even in expectation, whereas in the 2SAT algorithm we had, at worst, an expected change of 0.

If we do the same sort of calculations as above, the expected number of steps to find a satisfying assignment (if one exists) is $2^{n+o(n)}$, about the same as the time for a simple brute-force search of all possibilities.

However, a small modification to this approach gives us an algorithm whose expected running time is only $(\frac{4}{3})^{n+o(n)}$—still not polynomial, but more practical. If we choose a random solution that happens to be close to a satisfying assignment, then after just a few random variable flips, we expect to still be close to a satisfying assignment (and possibly find it)—but if we don't find it, then in expectation we've moved farther from a satisfying assignment, and we may as well give up and start over.

So, the algorithm is: pick a random assignment. For $3n$ steps, choose an unsatisfied clause and flip a random variable in it. If, by the end of those $3n$ steps, no satisfying assignment is found, then start over with a new random assignment. This finds a satisfying assignment (if one exists) in time $(\frac{4}{3})^{n+o(n)}$.