19.1 2SAT

We show 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. Solutions for 2SAT formulae can be found in polynomial time (or it can be shown that no solution exists) using resolution techniques; in contrast, 3SAT is NP-complete.

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.

If the algorithm terminates with a truth assignment, it clearly returns a correct answer. The case where the algorithm does not find a truth assignment requires some care, and we will return to this point later. Assume for now that the formula is satisfiable, and that the algorithm will actually run as long as necessary to find a satisfying truth assignment.

Let n be the number of variables in the formula. We refer to each time the algorithm changes a truth assignment as a *step*. We are mainly interested in the number of steps executed by the algorithm. We note, however, that since a 2-SAT formula has $O(n^2)$ clauses, each step can be executed in $O(n^2)$ time. Faster implementations are possible but we do not consider them here.

Let S represent a satisfying assignment for the n variables and A_i represent the variable assignment after the j-th step of the algorithm. Let X_j denote the number of variables in the current assignment A_j that have the same value as in the satisfying assignment S. When $X_j = n$ the algorithm terminates with a satisfying assignment. Starting with $X_j < n$, we consider how X_j evolves over time, and in particular how long it takes before X_j reaches n. Why would X_j tend to move to n?

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 X_j. If they disagree on the value one of the two variables, then with probability 1/2 we choose that variable and make increase X_j by 1; with probability 1/2 we choose the other variable and decrease X_j by 1.

Hence, in the worse case, X_j behaves like a *random walk*– it either goes up or down by 1, randomly. This leaves us with the following question: if we start X_j at 0, which is the worst possible case, how many steps does it take (on average, or with high probability) for X_j 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 $n^2 - i^2$.

Formally, if $X_j = 0$, then for any change in variable value on the next step we have $X_{j+1} = 1$. Hence

$$\Pr(X_{j+1} = 1 | X_j = 0) = 1.$$

When $1 \leq X_j \leq n-1$, the probability that we increase the number of matches is at least $1/2$, and the probability that we decrease the number of matches is at most $1/2$. Hence, for $1 \leq k \leq n-1$,

$$\begin{aligned} \Pr(X_{j+1} = k+1 | X_j = k) &\geq 1/2; \\ \Pr(X_{j+1} = k-1 | X_j = k) &\leq 1/2. \end{aligned}$$

Let us suppose the worst case – that the probability X_j goes up is 1/2, and down is 1/2. Now let $T(i)$ be the average (or expected) time to walk from i to n. Then by our assumptions $T(i)$ satisfies:

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

These equations follow by considering one step of the chain and using what is known as *linearity of expectations*. These equations completely determine $T(i)$, and our proposed solution satisfies these equations! ($T(n)$ is known to be 0, and given that, the above system has n equations and n unknowns. It can be checked that the equations are linearly independent, and hence there is a unique finite solution for each value of n.)

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!)

What about dealing with the issue of no solution? How long should we run before we give up, because when we give up, we might just not have found the solution that is there. We can run our algorithm for say $100n^2$ steps, and report that no solution was found if none was found. Again, this algorithm might return the wrong answer– there may be a truth assignment, and we have just been unlucky. But it seems like most of the time it will be right. Can we quantify this?

Divide the execution of the algorithm into segments of $2n^2$ steps each. We claim that for each segment, the probability of not finding a solution when one exists is at least 1/2, regardless of how we ended after the previous segment. This is because the expected number of steps to find a solution is at most n^2, so the probability of not finding an assignment after $2n^2$ steps is at most $1/2$. (Formally, this is due to what's called "Markov's inequality.") So after $100n^2$ steps, the probability we have not found a solution (when one exists) is at most 2^{-50}; that's a pretty small number.

19.2 3SAT

We now generalize the technique used to develop an algorithm for 2SAT to obtain a randomized algorithm for 3SAT. Since 3SAT is NP-complete, so it would be rather surprising if a randomized algorithm could solve the problem in expected time polynomial in n. [1] We present a randomized 3-SAT algorithm that solves 3-SAT in expected time that is exponential in n, but for a time was the best known proven bounds for any 3-SAT algorithm.

Let us first consider the performance our original algorithm when a applied to a 3-SAT problem. That is, we find an unsatisfied clause, and choose one of the variables at random to change. Do you see where the problem is? Following the same reasoning as for the 2-SAT algorithm, we have that for $1 \le k \le n-1$,

$$\begin{aligned} \Pr(X_{j+1} = k+1 | X_i = k) &\ge 1/3; \\ \Pr(X_{j+1} = k-1 | X_i = k) &\le 2/3. \end{aligned}$$

Unfortunately, we're much more likely to take a "step backwards" than a step forwards. The corresponding equations for the expected time from i matching variables would be

$$\begin{aligned} T(n) &= 0 \\ T(i) &= \frac{T(i-1)}{3} + \frac{2T(i+1)}{3} + 1,\ 1 \le i \le n-1 \\ T(0) &= T(1) + 1. \end{aligned}$$

One can check the solution to these equations ar given by: Again, these equations have a unique solution, given by

$$T(i) = 2^{n+2} - 2^{i+2} - 3(n-i).$$

Since we could just try all the 2^n possible truth assignments, this isn't very satisfying.

With a little work, though, we can improve our analysis. There are two key observations:

- If we choose an initial satisfying assignment *uniformly at random*, the number of variables that match S has a binomial distribution with expectation $n/2$. With an exponentially small but non-negligible probability, the process then *starts* with an initial assignment that matches S in significantly more variables.
- Once the above process starts, it is more likely to move toward 0 than toward n. The longer we run the process, the more likely it has moved toward 0, and the worse it is for us. Therefore, instead of just running the process until we find a solution, we are better off re-starting the process with many randomly chosen initial assignments and running the process each time for a small number of steps.

So consider the following process for finding an assignment (assuming one exists):

1. Repeat until all clauses are satisfied:
   (a) Start with a truth assignment chosen uniformly at random.
   (b) Repeat the following up to $3n$ times terminating if a satisfying assignment is found.
      i. Choose an arbitrary clause that is not satisfied.
      ii. Choose one of the literals uniformly at random, and change the value of the variable in the current truth assignment.

Assuming that a satisfying assignment exists, the expected number of steps until a solution is found can be bounded by $O\left(n^{3/2}\left(\frac{4}{3}\right)^n\right)$. As before, we can modify this to give an algorithm that either finds a satisfying assignment or terminates after a number of steps that is successful with high probability.

[1] Technically, this would not settle the $P = NP$ question, since we would be using a randomized algorithm and not a deterministic algorithm to solve an NP-hard problem. It would, however, have similar far-reaching implications about the ability to solve all NP-complete problems.

19.3 Randomized Complexity Classes

We can formalize randomized algorithms by introducing randomized variants of our models of computation. For example, we could augment the Word-RAM model with an instruction like "$R[i] \leftarrow$ COINTOSS" or "$R[i] \leftarrow \text{RANDOM}(R[j])$" which fills register i with a random bit or a random number from $\{0,1,\ldots,R[j]-1\}$, respectively. Or we could augment a Turing machine with the ability to write a random $\{0,1\}$ value instead of a symbol in a cell. Alternatively, we can use the same modelling as we did for nondeterministic computation, but have the algorithm choose among the possible GOTOs/transitions uniformly at random (rather than nondeterministically choosing an accepting path if one exists). It turns out that all of these choices are essentially equivalent for the purposes of studying the power of randomized computation (up to small factors in running time).

Given these models, we can now define complexity classes associated with randomized polynomial-time algorithms. By convention, the definitions are typically given with respect to randomized algorithms that *always* halt in polynomial time ("strict polynomial time"). But it can be shown that giving the definitions in terms of expected polynomial time (where on every input $|x|$, the expectation of the running time is at most $\text{poly}(|x|)$) yields the same classes.

Definition 19.1 RP *is the class of languages L for which there exists a randomized polynomial-time algorithm A such that:*

1. $x \in L \Rightarrow \Pr[A(x) = 1] \geq 1/2$.
2. $x \notin L \Rightarrow \Pr[A(x) = 1] = 0$.

Here the probabilities are taken over the coin tosses of the algorithm A.

Note that the error probability of 1/2 in an RP algorithm on YES instances can be reduced to 2^{-n} by repeating the algorithm n times. Similarly, we get the same class if we replace the $1/2$ with any other constant.

The random-walk algorithm for 2-SAT given above shows that 2-SAT is in RP. However, we also know that 2-SAT is in P (see lecture notes 14), so the advantages of the randomized algorithm are its simplicity and its space usage ($O(n)$ instead of $O(n^2)$). Still, it leaves open the question of whether there are any problems in RP but not in P. A prominent candidate is the following.

POLYNOMIAL IDENTITY TESTING
Input: an arithmetic formula $F(x_1,\ldots,x_n)$ over the integers. For example $(x_1+3x_2)\times(7x_3-2x_1)+6x_2\times(x_1+x_3)\times x_4+\cdots+6x_1\times(2x_2+4)$.
Question: Is F equivalent to the zero polynomial? That is, when we expand and collect terms to write F as a linear combination of monomials $F(x_1,\ldots,x_n)=\sum_{i_1,\ldots,i_n} c_{i_1,\ldots,i_n} x_1^{i_1} x_2^{i_2}\cdots x_n^{i_n}$, are all the coefficients $c_{i_1,\ldots,i_n}=0$. As will follow from what we show below, this is equivalent to F evaluating to 0 on all inputs in $\mathbf{Z}^n$.

There is a very simple randomized algorithm for this problem: evaluate F at a randomly chosen input $\alpha = (\alpha_1,\ldots,\alpha_n)$, chosen uniformly at random from S^n for any large enough finite set $S \subseteq \mathbf{Z}$. The correctness of such an algorithm follows from the following lemma.

Lemma 19.2 (Schwartz–Zippel Lemma) *Let $F(x_1,\ldots,x_n)$ be a nonzero polynomial of degree at most d over $\mathbf{Z}$, and let $S \subseteq \mathbf{Z}$. Then:*

$$\Pr_{(\alpha_1,\alpha_2,\ldots,\alpha_n)\leftarrow S^n}[F(\alpha_1,\ldots,\alpha_n)=0] \leq \frac{d}{|S|}.$$

Note that when $n = 1$, the lemma is the familiar fact that a nonzero polynomial of degree d has at most d roots. The proof of the full lemma is obtained by inductive application of this fact.

So to obtain an error probability of at most $1/2$, it suffices to choose evaluation points using a set S of size at least $2d$. How can we bound d without expanding the polynomial? An inductive proof shows that the degree is at most the size of the formula F (where each occurrence of a variable or an operation contributes 1 to the size), so it suffices to take $S = \{0, 1, \ldots, 2|F|\}$.

Note that this shows that the language PIT $= \{\langle F\rangle : F$ an arithmetic formula over $\mathbb{Z}$ s.t. $F \equiv 0\}$ is in co-RP. PIT is not known to be in P, and in fact no subexponential-time algorithm is known for PIT.

RP and co-RP only allow for one-sided error, namely false negatives or false positives, but not both. If we allow both, we obtain "bounded-error probabilistic polynomial time":

Definition 19.3 BPP *is the class of languages L for which there exists a randomized polynomial-time algorithm A such that:*

1. $x \in L \Rightarrow \Pr[A(x) = 1] \geq 2/3$.

2. $x \notin L \Rightarrow \Pr[A(x) = 0] \leq 1/3$.

Here the probabilities are taken over the coin tosses of the algorithm A.

For BPP, it can also be shown that the error probability can be made exponentially small by repeating the algorithm many times and taking a majority vote (the proof requires a nontrivial probability fact called the "Chernoff Bound").

It is not hard to see that RP $\subseteq$ NP, but we do not know how to rule out the possibility that BPP $=$ NEXP, where NEXP is the class of problems that can be decided in nondeterministic *exponential* time!

On the other hand, as discussed in Avi Wigderson's talk last week, there is strong evidence that randomness does not provide so much of an advantage. Indeed, we can prove this if the NP-complete problems are as hard as we think they are:

Theorem 19.4 *If* SAT *requires time* $2^{\Omega(n)}$ *even for "nonuniform" algorithms, then* BPP $=$ P.

At a high level, this theorem is proved by showing how to use the hardness of a problem like SAT to deterministically generate a small family of sequences of "pseudorandom" bits that no polynomial-time algorithm can tell apart from truly random bits (so we can use these to "derandomize" the algorithm). If you're interested in seeing how such results are proven, take CS 225!