1 More Hashing

1.1 Counting Bloom Filters

Recall from last section the bloom filter, a data structure for membership problems that are more space-efficient than hashing schemes (since we store just 1 bit per slot in the hash table), but at the cost of some probability of false positives. As review, Bloom filters involve \( k \) hash functions \( f_1, \ldots, f_k \) mapping into \( m \) bits, where bits \( f_1(x), \ldots, f_k(x) \) are set to 1 when adding \( x \) to the set, and similarly \( f_1(x), \ldots, f_k(x) \) are checked to be 1 when looking up \( x \).

We showed last section that these simple Bloom filters could not support deletions, since deletions could lead to false negatives. A slightly modified data structure, called the counting bloom filter, uses more space but supports deletions. Now, instead of each of the \( m \) slots having just 1 bit, they each have \( j \) bits, and adding \( x \) to the set increments the counters at slots \( f_1(x), \ldots, f_k(x) \), and deleting \( x \) from the set decrements the counters at those slots, after checking for \( x \)'s membership.

It’s possible, however, for these counters to overflow:

Exercise 1. What is the probability a counting bloom filter with \( k \) hash functions mapping into \( m \) \( j \)-bit counters overflows at a specific counter after adding \( n \) elements?

Solution

(This is for the \( k \) hash functions sharing all \( m \) counters. A similar analysis can be made for each hash function having its own table of \( m/k \) counters)

Each of \( n \) elements is hashed \( k \) times, and each of the \( nk \) hashes has probability \( 1/m \) to land in our counter, so by the story of the binomial the distribution of number of hashes in our counter is \( X = \text{Bin}(nk, 1/m) \). For a \( j \)-bit counter, the max value is \( 2^j - 1 \) so it overflows when \( X \geq 2^j \) hence the probability of overflowing at a specific counter is

\[
1 - \sum_{i=0}^{2^j-1} P(X = i) = 1 - \sum_{i=0}^{2^j-1} \binom{nk}{i} (1/m)^i (1-1/m)^{nk-i}
\]

2 Karger-Stein

Recall from lecture Karger’s algorithm to find the minimum cut in an unweighted graph \( G \):

procedure CONTRACT\((G = (V, E))\)
  if \(|V| = 2\) then output the only cut
  else
    pick a random edge \( e \)
    contract the edge \( e \) to form \( G' = (V', E') \)
  return CONTRACT\((G')\)
We showed in lecture that, for any minimum cut $C$, this randomized algorithm outputs $C$ with probability $1/\binom{n}{2}$ and moreover the contraction can be implemented to run in $O(n^2)$ time.

We now turn to Karger and Stein’s variant which outputs a minimum cut with higher probability. Notice that in the first round of contraction the probability that some mincut $C$ survives is pretty good: at least $1 - 2/n$. Meanwhile toward the end, the probability $C$ survives starts getting much farther away from one: we can only say it is at least $1/3$ in the last iteration.

The idea of Karger and Stein was to run contraction not until we get down to two vertices, but rather until we get down to $t$ vertices. Then the probability that $C$ has still survived up until this point, by contraction, is at least $t(t - 1)/(n(n - 1))$. By picking $t = \lceil n/\sqrt{2} \rceil + 1$, we have that the probability that $C$ survives after contracting down to $t$ vertices is at least $1/2$. The Karger-Stein algorithm, henceforth known as KS, is then as follows:

\begin{verbatim}
procedure KS(G = (V, E))
    if |V| < 6 then try all $2^{|V| - 1}$ cuts and return the smallest one
    else
        $t \leftarrow \lceil n/\sqrt{2} \rceil + 1$
        Obtain $G_1$ by contracting $G$ $n - t$ times to obtain $t$ vertices
        Obtain $G_2$ by contracting $G$ $n - t$ times to obtain $t$ vertices
            (using independent randomness from last step)
        $C_1 \leftarrow KS(G_1)$
        $C_2 \leftarrow KS(G_2)$
        return the smaller of the two cuts $C_1$ and $C_2$
end
\end{verbatim}

The reason we make $|V| < 6$ a special case is that $\lceil n/\sqrt{2} \rceil + 1 \geq n$ for $n < 6$, and thus the recursive
calls in the else statement would not actually shrink the problem size for $n < 6$.

**Exercise 2.** What is the running time of the above algorithm?

**Solution**
The original contraction algorithm can be implemented to run in time $O(n^2)$. Thus the running time $T(n)$ of KS satisfies the recurrence

$$T(n) \leq 2T(\lceil n/\sqrt{2} \rceil + 1) + O(n^2).$$

The Master theorem then implies $T(n) = \Theta(n^2 \log n)$.

The key benefit though will be that KS has much better success probability. Whereas the vanilla contraction algorithm had success probability at least $1/(n^2)$, KS has success probability at least $\Omega(1/\log n)$.

**Exercise 3.** Show that KS outputs a minimum cut with probability $\Omega(1/\log n)$.

**Solution**
Let $p_k$ be a lower bound on the probability that KS returns a minimum cut on its input at the $k$th level of recursion, where $k = 0$ corresponds to the base case $n < 6$. Then clearly $p_0 = 1$ is a valid lower bound since we try all cuts. What about $p_{k+1}$? By our choice of $t$, the probability that $G_1$ still contains a minimum cut of $G$ is at least $1/2$ (and similarly for $G_2$). Thus the probability that $C_1$ is a minimum cut of $G$ is at least $(1/2) \cdot p_k$. The same is true for $C_2$. Since $G_1$ and $G_2$ are independent, the probability that neither $C_1$ nor $C_2$ is a minimum cut of $G$ is thus at most $(1 - (1/2)p_k)^2$. Therefore, a valid lower bound on the probability that KS returns a minimum cut at the $(k+1)$st level of recursion is

$$p_{k+1} = 1 - (1 - (1/2)p_k)^2 = p_k - (1/4)p_k^2. \quad (1)$$

Define new variables $z_k$ by $p_k = 4/(z_k + 1)$. Then $z_0 = 3$ and a substitution into (1) shows that $z_{k+1} = 1 + z_k + 1/z_k$. Induction on $k$ then implies that $z_k \geq k$ for all $k \geq 0$. Then, given this, induction on $k$ then implies $z_k \leq 3 + 2k$ for all $k \geq 0$. Notice the highest level of recursion of interest is some $k^* = O(\log n)$, since in each recursive call $n$ decreases by a factor of roughly $\sqrt{2}$ (which can only happen $O(\log n)$ times). Thus $z_{k^*} = O(\log n)$; remembering the definition of $z_k$, this implies $p_{k^*}$, the success probability at the root of the recursion tree, is $\Omega(1/\log n)$.

Given the above lemma, as with the analysis for the vanilla contraction algorithm, we see that it suffices to run KS $\Theta(n^2 \log(1/P))$ times to have success probability $1 - P$. Since the KS running time itself is $O(n^2 \log(n \log(1/P)))$, that implies that the total running time to obtain success probability $1 - P$ is $O(n^2 (\log n)^2 \log(1/P))$.

### 3 Randomized return to 2-SAT

In lecture, we gave the following randomized algorithm for solving 2-SAT. 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.
A random walk is an iterative process on a set of vertices $V$. In each step, you move from the current vertex $v_0$ to each $v \in V$ with some probability. The simplest version of a random walk is a one-dimensional random walk in which the vertices are the integers, you start at 0, and at each step you either move up one (with probability 1/2) or down one (with probability 1/2).

We used a random walk with a completely reflecting boundary at 0 to model our randomized solution to 2-SAT. Fix some solution $S$ and keep track of the number of variables $k$ consistent with the solution $S$. In each step, we either increase or decrease $k$ by one. Using this model, we showed that the expected running time of our algorithm is $O(n^2)$.

**Exercise 4.** Solve the 2SAT recurrence relations analytically (in lecture we just plugged in a function to see it was a solution):

\[
T(n) = 0
\]
\[
T(i) = \frac{T(i-1) + T(i+1)}{2} + 1, \quad i \geq 1
\]
\[
T(0) = T(1) + 1.
\]

**Solution**

Summing all $n + 1$ equations gives

\[
\sum_{i=0}^{n} T(i) = \frac{T(0) + T(1) + T(n-1) + T(n)}{2} + n + \sum_{j=1}^{n-2} T(j)
\]
\[
T(0) + T(n-1) = T(1) + 2n
\]
\[
T(n-1) = 2n - 1
\]

Now we can apply induction: suppose for $i = k, k - 1$ that $T(i) = n^2 - i^2$. The base cases of $i = n, n - 1$ are true by the above work. Then we have that $T(k - 1) = \frac{T(k-2) + T(k)}{2} + 1 \implies T(k - 2) = 2T(k - 1) - T(k) - 2$ and plugging in the inductive hypothesis gives $T(k - 2) = 2(n^2 - (k - 1)^2) - (n^2 - k^2) - 2 = n^2 - (k - 2)^2$, completing the induction, so $T(i) = n^2 - i^2$ for all $0 \leq i \leq n$.

(Fun solution) Recall from section 7 that the expected number of steps for a random walk starting at $i$ and ending at either $a < i$ or $b > i$ is $T(i) = (b - i)(i - a)$. The 2SAT random walk is different from the normal random walk since we must go from 0 to 1 because, given an assignment with 0 'correct' variables, switching any will give you a correct variable. If we identify $-i$ with $i$ for $1 \leq i \leq n$, however, the 2SAT random walk is a normal random walk that ends whenever we hit $n$ or $-n$. Starting from $i$, then, the expected number of steps to end is just $(n - i)(i - (n-i)) = n^2 - i^2$.

(Fun solution #2) Consider the quantity $a^2 - s$, where $a$ is the value you are currently at, and $s$ is the number of steps you have made. Note that the expected value of this quantity never changes, since after one step you are either at $(a + 1)^2 - (s + 1)$ or $(a - 1)^2 - (s + 1)$ with equal probability, and

\[
\frac{1}{2}((a + 1)^2 - s + 1) + \frac{1}{2}((a - 1)^2 - s + 1) = a^2 - s
\]

Starting at $i$, the quantity is $i^2 - 0 = i^2$. When we hit $n$, this expected value of this quantity should still be $i^2$, so $n^2 - T(i) = i^2 \implies T(i) = n^2 - i^2$