Rupak Majumdar

Mathematical Foundations of Random Testing

August 14, 2023

Springer Nature

Contents

Chapter 1
Testing and Coverage

1.1 A Basic Theorem

We first state and prove a general theorem on test coverage. We leave the notions of coverage goals, tests, or the notion of covering abstract—we will instantiate these notions in specific cases.

Let M be a nonempty set of *coverage goals*. Let T be a set of *tests*. A test $t \in T$ may or may not *cover* a coverage goal. A nonempty set F of tests is a *covering family* for M if for each $x \in M$, there is a test $t \in F$ such that t covers x.

Theorem 1.1 *Let M be a set of m coverage goals. Let $p > 0$ be a lower bound on the probability that a random test $t \in T$ covers a fixed coverage goal. Given $\epsilon > 0$, let F be a family of tests chosen independently and uniformly at random such that $|F| \geq p^{-1}(\log m - \log \epsilon)$. Then F is a covering family with probability at least $1 - \epsilon$. Moreover, there exists a covering family of size $\lceil p^{-1} \log m \rceil$ (or 1 if $m = 1$).*

Proof Consider a fixed coverage goal x. A random test does not cover x with probability at most $1 - p$. Since the tests in F are chosen independently, the probability that F does not cover x is at most $(1 - p)^{|F|}$. By the union bound, the probability that there exists a coverage goal not covered by F is at most $m(1 - p)^{|F|}$.

If $m > \epsilon$, using $|F| \geq p^{-1}(\log m - \log \epsilon)$ and the fact that $p < -\log(1 - p)$, we get

$$|F| > \frac{-\log m + \log \epsilon}{\log(1 - p)} = \log_{1-p}(m^{-1}\epsilon) \ .$$

Note that this trivially holds if $m \leq \epsilon$. Therefore, in both cases $m(1 - p)^{|F|} < \epsilon$, and the probability that F covers all coverage goals is at least $1 - \epsilon$. In particular, if we take $\epsilon = 1$, the probability that F covers all objects is positive. By the probabilistic method, there must exist a covering family of size $\lceil p^{-1} \log m \rceil$, or 1 if $m = 1$, since a covering family needs to be nonempty. □

There is always a trivial covering family of size $|M|$ (assuming each coverage goal can be covered by some test). The key observation in Theorem 1.1 is that there exist covering families of size proportional to $\log|M|$; thus, if we can show the probability p is "high," we can get an exponentially smaller covering family of tests. Moreover, a randomly chosen test set can cover all goals with high probability.

1.2 Applications

Let us demonstrate how Theorem 1.1 can be used to analyze the coverage notion involving sequences of operations motivated in Section **??**. Suppose we have $r \geq 1$ different operations, and we are generating a sequence of operations s uniformly at random. Suppose we also have a set T of target sequences of length $k \geq 1$, and we want to observe any target sequence $t \in T$ as a contiguous subsequence of s.

Theorem 1.2 *Let $\epsilon > 0$, let T be a set of sequences of operations of length $k \geq 1$, and let s be a sequence of $n \geq 1$ operations chosen independently and uniformly at random such that $n \geq k - kr^k|T|^{-1} \log \epsilon$. Then some target sequence $t \in T$ is a contiguous subsequence of s with probability at least $1 - \epsilon$.*

Proof Split the sequence s into $\lfloor n/k \rfloor$ non-overlapping subsequences of length k. The probability that some $t \in T$ occurs among these subsequences is clearly lower than the probability that some $t \in T$ occurs in s. However, we can think of the non-overlapping sequences as $\lfloor n/k \rfloor$ sequences of length k generated independently and uniformly at random.

The probability that one of these sequences matches a sequence in T is $p = |T|/r^k$. The number of coverage goals in this case is $m = 1$, namely any target sequence $t \in T$. Since $\lfloor n/k \rfloor > n/k - 1$, we have

$$\lfloor n/k \rfloor > -r^k|T|^{-1} \log \epsilon = p^{-1}(\log m - \log \epsilon) \quad .$$

Hence the result follows from Theorem 1.1. □

Exercise 1.1 (Combinatorial Testing and Covering Arrays) Suppose you want to test a software with N different Boolean "features." Each feature can be turned on or off. You could ask that you run a test for each combination of features, but this gives 2^N tests. In *K-wise combinatorial testing*, you fix a parameter K and ask that for each subset of K features, and for each of the 2^K settings of these features, there is a test that covers this setting.

In general, for a fixed K, the number of tests can be many fewer than 2^N. Using the probabilistic method, find the number of tests required to cover all K-wise combinatorial tests with high probability (say with probability greater than $1/10^6$.

1.3 Notes

[3]

Chapter 2
Sampling from Combinatorial Objects

Random testing proceeds by "randomly picking" a test from a space of tests. We assume we have, as basic primitives, the ability to uniformly sample a number from the interval $(0, 1)$, or an element from a finite set $\{1, \ldots, k\}$. In this chapter, we consider how this primitive can be used to sample *uniformly* from different combinatorial objects.

What do we mean by uniform sampling? We assume that we are given a family C_n of objects, parameterized by their size n. We assume each $|C_n|$ is finite. Our goal is to pick $c \in C_n$ such that each $c \in C_n$ is chosen with probability $1/|C_n|$ (uniform sampling). Moreover, we wish to do this in time that is efficient in n, say a polynomial function of n.

Exercise 2.1 Show that if we do not require the efficiency constraint, we can perform uniform sampling by enumerating all elements of C_n.

Sometimes, uniform sampling is easy: for example, to sample uniformly from the space of n-bit binary strings, we can simply pick each bit independently and uniformly, using our primitives. Similarly, if we want to uniformly sample a random mapping from a set of n elements to a set of n elements, we can independently pick an element in the range for each element in the domain.

As we shall see, one requires more care in sampling from other combinatorial objects. In fact, we shall see that efficient sampling may not even be possible for some families (unless some complexity classes collapse).

2.1 Sampling a Subset from a Set

The simplest scenario for random testing is to sample exactly n items at random out of a set of m items in an unbiased way, that is, ensuring that every subset of n items has equal probability to be picked. We consider several different settings: the items may be chosen with or without replacement, and the total number of items m may or may not be known up front.

Since the total number m of items can be very large, we shall consider sequential sampling papproaches, where we do not assume that all items are available at the same time. Instead, we shall sequentially step through the items and decide whether or not to accept an item as it is scanned.

The simplest approach to sampling n items is to pick each item independently with probability $\frac{n}{m}$. However, this only leads to n items *on average* and a particular sample may end up with more or fewer items. In fact, the variance of the method is $m \cdot (n/m) \cdot (1 - n/m) = n(1 - n/m)$, so many samples will end up being too small or too large.

Instead, we pick the $i+1$th item with probability $\frac{n-k}{m-i}$, where we have already picked k items. Why is this appropriate? Of all the possible ways to choose n items out of m items such that k of them occur in the first i positions, exactly

$$\binom{m-k-1}{m-n-1} \Big/ \binom{m-i}{m-n} = \frac{n-k}{m-i}$$

of them select the $i + 1$th element. This leads to the sampling procedure in Algorithm 1. We prove the correctness of the procedure and also analyze its performance.

Algorithm 1 Sequential sampling SeqSample

assume $0 < n \leq m$
$i = 0$; $k = 0$; $S = \emptyset$
repeat
  pick r u.a.r. between $(0, 1)$
  if $(m - i)r \geq n - k$ **then**
    $i = i + 1$ // do not include this item in the sample, look at the next item
  else
    $S = S \cup \{item[i]\}$ pick the next item in the sample
    $k = k + 1$; $i = i + 1$;
  end if
until $k = n$ **return** S

Proposition 2.1 *The loop in Algorithm 1 runs for at most m steps. For any set S of items of size n, Algorithm 1 returns S with probability* $1/\binom{m}{n}$.

Let $p(k, i)$ be the probability that exactly k items are selected from the first i items. The probabiity satisfies the recurrence relation

$$p(k, i + 1) = \left(1 - \frac{n - k}{m - i}\right) p(k, i) + \frac{n - k}{m - i} p(k - 1, i)$$

From this recurrence, we get

$$p(k, i) = \binom{i}{k}\binom{m - i}{n - k} / \binom{m}{n}$$

for $0 \leq i \leq N$. Thus, $p(n, m) = 1$.

To show the algorithm picks every subset of n elements uniformly, fix a set S and assume its elements occur at positions

$$1 \leq i_1 < i_2 < \ldots < i_n \leq m$$

Define additionally $i_0 = 0$ and $i_{n+1} = m + 1$. The probability of picking this specific set is $p = \prod_{j=1}^{m} p_j$, where

$$p_j = \begin{cases} (m - (j - 1) - n + k)/(m - (j - 1)) & \text{for } i_k < t < i_{k+1} \\ (n - k)/(m - (j - 1)) & \text{for } j = i_{k+1} \end{cases}$$

The denominator of the product is $m!$: the term p_j contributes $(m - (j - 1))$. The numerator contains the terms n, $n - 1, \ldots, 1$ for the positions j corresponding to the i's and contains the terms $m - n, m - n - 1, \ldots, 1$ for the positions j that are not the i's. Thus, the numerator is $(m - n)!n!$ and we have

$$p = (m - n)!n!/m! = 1/\binom{m}{n}$$

Exercise 2.2 Show that the probability that any given item (say the $i + 1$th item) is picked into S is $\frac{n}{m}$. Why is this not a contradiction, given that the algorithm picks an item with a different probability $(n - k)/(m - i)$?

Solution 2.1 This is not a contradiction because of conditional probabilities. The variable k depends randomly on the previous selections in the first i steps. If we consider the average over all possible choices in the first i steps, we get the probability of picking an element is n/m on average. The first element is picked with probability n/m (by definition). The second element is picked with probability $(n - 1)/(m - 1)$ if the first element is selected and with probability $n/(m - 1)$ if the first element is not selected. Thus, the overall probability is

$$(n/m)\,((n - 1)/(m - 1)) + (1 - n/m)\,(n/(m - 1)) = n/m$$

By induction, for the $(i + 1)$th item, the probability it is picked is

Algorithm 2 Reservoir sampling

assume $0 < n \leq m$
add the first n items into S; $i = n$
while there are more items to process **do**
  $i = i + 1$
  pick M u.a.r. between 1 and i, inclusive
  if $M \leq n$ **then**
    $SM] = item[i]$ // replace the Mth item in the sample with the new item
  else
    skip over the next item // if $M > n$, do not include this item in the sample, look at the next item
  end if
  skip
  $k = k + 1$; $i = i + 1$;
end whilereturn S

$$\sum_{j=0}^{i} \binom{i}{j} (n/m)^j (1 - n/m)^{i-j} ((n - j)/(m - i)) = n/m$$

While Algorithm 1 reaches the last item in the "worst case," on average, it can stop much earlier. Using the probability $p(k, i)$, the probability that $i = \hat{i}$ at termination is $q_{\hat{i}} = p(n, \hat{i}) - p(n, \hat{i} - 1) = \binom{\hat{i}-1}{n-1}/\binom{m}{n}$. The expected stopping point is therefore

$$\sum_{j=0}^{m} j q_j = (m + 1)n/(n + 1)$$

and the variance is

$$(m + 1)(m - n)n/(n + 2)(n + 1)^2$$

If $n \sim 0.1m$, the expected stopping point is

What happens if the total number of items m is not known a priori? For example, what if the items are presented to us, one at a time, as a stream (e.g., a file stream), and we do not know how many items there are (other than there are at least n items)? Then, we maintain n items that constitute the current sample, and update the current sample when processing the next item. The first n items all go into the current sample. When the $i + 1$th item is being processed, for $i \geq n$, it is added to the sample with probability $n/(i + 1)$ and a random member of the current sample is evicted to make room.

In fact, a small modification of this algorithm can be used to sample a subset even when the items are too big to fit into memory: we keep indices of the items in the reservoir and look up the selected items in a subsequent pass.

2.2 Sampling a Permutation

Next, consider the problem of picking a random permutation of n items. Since there are $n!$ permutations, one way to uniformly randomly generate a permutation is to pick a number between 1 and $n!$ uniformly at random, and then use that number as an index to a table of all permutations. When n is large, this is not efficient as $n!$ is exponentially large.

Exercise 2.3 Suppose we define a comparison operation

```
def cmp(a, b):
  if a == b:
    return 0
  if rand():
    return +1
  else
    return -1
```

Suppose we decide to pick a permutation by sorting based on this comparator. Does it give a uniform random permutation?

Instead, we can generate a random permutation element by element, by picking a sequence of random numbers. We pick x_1 uniformly at random from the set of items. In the ith step, having picked $x_1, \ldots, x_{i-1}$, we pick a number j uniformly from $[1, n-i+1]$, and set x_i to be the jth largest of the items not chosen so far.

Proposition 2.2 *The sequence $x_1 \ldots x_n$ is a permutation of $[1, n]$. Moreover, the probability of picking a specific permutation is $\frac{1}{n!}$.*

Proof Each element is picked at most once, and all n elements are picked. Thus, the sequence is a permutation.

A j-permutation is a sequence containing j of the n items. There are $n!/(n-j)!$ possible j-permutations.

We shall use induction to show that at the beginning of the ith step, for each $(j-1)$-permutation, the sequence $x_1 \ldots x_{j-1}$ contains this permutation with probability $(n-j+1)!/n!$. At the beginning, this holds vacuously because the sequence is empty and a 0-permutation has no elements.

Suppose that the property held before the $(j-1)$th step. Fix a j-permutation $(y_1, \ldots, y_j)$. The probability that the algorithm sets $(x_1 \ldots, x_j)$ to this permutation is the probability of the intersection of two events: that $(x_1, \ldots, x_{j-1})$ is set to $(y_1, \ldots, y_{j-1})$ and that y_j is chosen in the jth step. By induction hypothesis, the first event occurs with probability $(n-j+1)!/n!$ and the probability that y_j is chosen in the jth step given $(y_1, \ldots, y_{j-1})$ was chosen before is $1/(n-j+1)$. Thus, the probability at the next iteration is $(n-j)!/n!$, as required.

Finally, on termination, any n-permutation is chosen with probability $1/n!$. □

A different procedure, called the Fisher-Yates shuffle or the Knuth shuffle, starts with an arbitrary permutation (for example, the identity permutation), and then goes through all positions j from n to 2, swapping the element at position j with a randomly chosen element from positions 1 through j, inclusive.

Algorithm 3 Fisher-Yates shuffle

Require: : array X, initially $X[i] = i$ for $i \in [1, n]$
  $j = n$
  while $j > 1$ **do**
    pick r u.a.r. from $[0, 1]$
    let $k = \lfloor jU \rfloor + 1$ ▷ k is a random integer between 1 and j
    exchange $X[j]$ and $X[k]$
    $j = j - 1$
  end while

Proposition 2.3 *The probability of generating a fixed permutation is $\frac{1}{n!}$.*

Exercise 2.4 Prove Proposition 2.3

Exercise 2.5 • Consider the following algorithm: for j going from 1 to n, pick a random number k in $[j, n]$ and swap $X[j]$ with $X[k]$. Show that this generates a uniform random permutation.
• Will the procedure still generate a uniform random permutation if k is chosen from $[1, n]$?

Solution 2.2 We show the property by induction. We show that for the jth iteration of the loop, the array $X[1 : j-1]$ contains any $j-1$ permutation with probability $(n-j+1)!/n!$. At the first iteration, $X[1:0]$ is empty, and there are no 0-permutations. Assume that at the beginning of the jth iteration, each possible $j-1$ permutation appears in the array $X[1 : j-1]$ with probability $(n-j+1)!/n!$.

Consider the jth iteration. Fix a specific j-permutation $\langle x_1, x_2, \ldots x_j \rangle$. This permutation consists of an $(j-1)$-permutation $\langle x_1, x_2, \ldots, x_{j-1} \rangle$ followed by a value $x_j = X[j]$ that the algorithm places in $X[j]$. By the induction hypothesis, the probability that the algorithm has placed this particular $j-1$ permutation in $X[1 : j-1]$ is $(n-j+1)!/n!$. What is the prbobability that at the end of the jth iteration we have the specific permutation? It is the probability that $\langle x_1, \ldots, x_{j-1} \rangle$ is picked in the first $j-1$ iterations and that x_j is placed in $X[j]$. This is the same as the probability

$\langle x_1, \ldots, x_{j-1} \rangle$ is picked times the probability x_j is picked given $\langle x_1, \ldots, x_{j-1} \rangle$ has been picked. The first probability is $(n-j+1)!/n!$ and the second is $1/(n-j+1)$ because the algorithm picks x_j uniformly at random from the $n-j+1$ values in $X[j:n]$. Thus, the probability is $(n-j)!/n!$.

The loop terminates when $j = n+1$, and we have that any permutation is picked with probability $1/n!$.

It turns out (but it is a difficult exercise!) that the modification produces non-uniform samples. ♣ RM: where is this shown?

Exercise 2.6 Suppose that in the Fisher-Yates shuffle algorithm, you picked k to be between 1 and $j-1$, rather than between 1 and j. What does the algorithm produce?

2.3 Sampling Binary Trees

A rooted binary tree is a tree with a distinguished root node such that each internal node has exactly two children. Nodes without any children are called leaves. A rooted binary tree with n internal nodes has $n+1$ leaves and $2n$ edges.

Let C_n denote the set of rooted binary trees with n nodes. The size of C_n, that is, the number of binary trees with n nodes, is given by the *Catalan numbers*

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

The numbers C_n grow exponentially in n, so we cannot enumerate trees and sample. Instead, we describe a clever algorithm due to Remy that uses a bijection on trees to recursively build up a sample. First, we need the following.

Proposition 2.4 *The Catalan numbers satisfy the following identity.*

$$(n+1)C_n = (4n-2)C_{n-1}$$

Remy's algorithm uses the above relation to get a bijection

$$(t^{(n)}, f) \quad \leftrightarrow \quad (t^{(n-1)}, \langle e, s \rangle$$

between trees $t^{(n)}$ in C_n and a marked leaf f of $t^{(n)}$ on one side, and smaller trees $t^{(n-1)} \in C_{n-1}$, an edge e of $t^{(n-1)}$ and a side (left or right) of the edge. The intuition is that we can pick a leaf of the larger tree and remove it and its parent from the tree, directing the edge from the grandparent to its sibling. We mark the edge between the grandparent and the sibling, and the direction in which the parent's child was removed.

♣ RM: make pictures

```
def generate(n):
    if n = 0, return unique tree in C_0
    t = generate(n-1)
    pick an edge and a side u.a.r. in t
    attach a leaf in the middle of e on chosen side
    return t
```

Proposition 2.5 *Algorithm* `generate(n)` *picks trees in C_n with uniform probability.*

Proof We prove this by induction. For $n = 0$, this is obvious.

By induction, the recursive call picks a tree u.a.r. from C_{n-1}. Then, each pair $(t^{(n-1)}, \langle e, s \rangle)$ is picked with probability

$$\frac{1}{|C_{n-1}|(2n-1)\cdot 2}$$

Applying the bijection, we get a pair $t^{(n)}, f)$, where $t^{(n)} \in C_n$ and f is a marked leaf u.a.r. Since each tree has the same number of leaves, the marginal distributrion on $t^{(n)}$ is u.a.r. on C_n. □

Note that the complexity of the procedure is linear in n (given in unary).

Exercise 2.7 (Hard: Motzkin trees) Suppose that each internal node can have 0, 1, or 2 children. How can you sample such trees using ideas similar to Remy's algorithm?

Exercise 2.8 (Binary Search Trees) Suppose you want to test a binary search tree implementation by uniformly sampling binary search trees. First, you generate a uniform permutation of the set $\{1, \ldots, n\}$ and insert the elements in the order of the permutation into an empty binary search tree.

Does this give you a uniform sample over binary search trees? (How many permutations are there? How many binary search trees are there?)

Second, consider generating a binary tree u.a.r. using Remy's algorithm. Can you label the nodes of the tree with integers or a special item "$\perp$" denoting absence of an element to get a binary search tree? Does this give you a random sample over binary search trees?

Now consider the following technique. To generate a tree with n elements, we do the following. We pick a number $x \in [0, 1]$ uniformly at random. We assign $\lfloor xn \rfloor$ nodes to the left subtree, put the next node at the root, and put the remaining nodes in the right subtree. We recurse on the left and right subtrees. What distribution on binary search trees do we get?

2.4 Sampling a Partition

Next, we consider sampling from the partitions of a set.

Throughout this section, let $U = \{1, \ldots, n\}$ be a fixed set (a "universe") of n elements. A *partition* of U is a set of nonempty subsets of U that are pairwise disjoint and in the union give the whole U. We refer to the sets in a partition as *blocks*. If a partition has k blocks, we call it a k-partition. A *balanced partition* is a partition with blocks differing in size at most by 1.

Let us recall a few results about partitions. The number of k-partitions is given by a quantity called *Stirling number of the second kind*, denoted $\left\{ {n \atop k} \right\}$ and read "n subset k" [2]. It is not difficult to see that $\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1$ whenever $n \geq 1$. Moreover, $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$, as a 2-partition is uniquely determined by the block that does not contain the nth element, and this block needs to be nonempty. In general, Stirling numbers of the second kind satisfy the following recurrence:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\} \quad . \tag{2.1}$$

Combinatorially, we can partition n elements into k blocks by partitioning the first $n-1$ elements into $k-1$ blocks and adding a singleton block consisting of the nth element, or by partitioning the first $n-1$ elements into k blocks and placing the nth element into one of these blocks in k ways.

Lemma 2.1 *For every $n \geq 1$ and k such that $1 \leq k \leq n$, we have*

$$\left\{ {n \atop k} \right\} k! \leq k^n \quad .$$

Proof The quantity on the right-hand side is the number of all functions from an n-element set to a k-element set, while the quantity on the left-hand side is the number of such functions that are *surjective*. To see this, note that a surjection induces a k-partition of the domain, and the induced blocks map to the codomain in one of $k!$ ways. □

For a fixed k, $\left\{ {n \atop k} \right\}$ asymptotically approaches $k^n / k!$. Intuitively, if we randomly assign n elements into k buckets and n is large, it is unlikely one of the buckets will be empty. Therefore, the difference between the left-hand side and right-hand side in Lemma 2.1 will be small.

2.5 Notes

Knuth's *The Art of Computer Programming* [**?**, **?**] contains a wealth of information about combinatorial algorihms and random generation. The sequential sampling algorithm is from [1].

Exercise 2.5(b) is from [**?**] (Exercise 3.4.2.18).

The results on splitting families and minority isolation are from [**?**].

Chapter 3
The Complexity of Sampling

In Chapter 2, we considered different sampling problems from combinatorial families. In this chapter, we look at uniform generation from constrained combinatorial families. That is, we consider sampling from families defined as the solution to a combinatorial problem. We show that the resulting computational complexity of sampling lies between the *existence* problem (check if a solution exists) and the *counting* problem (count the number of feasible solutions).

3.1 A Bit of Analytic Combinatorics

To provide a simple example of sampling with constraints, suppose that we want to generate n-bit binary strings that do not have any occurrence of the (contiguous) subsequence 000. One option is to perform *rejection* sampling: pick a random string of n bits, but throw it away if it contains 000 and try again. This process gives you a uniform distribution of strings without 000 almost surely. You may have to repeat the generation several times before you get an answer.

Let us see how many times you have to repeat the process in expectation. For this, we compute the probability that an n bit string does not have a contiguous substring of 000. We write the recursive definition of bitstrings B_{00} that do not have contiguous 000 strings:

$$B_{00} = E + Z_0 + Z_0Z_0 + B_{00}Z_1 + B_{00}Z_1Z_0$$

where E is the empty string, Z_0 and Z_1 are the strings 0 and 1 respectively. Informally, the recurrence states that every string in B_{00} is either the empty string, the singleton 0, or a shorter string in B_{00} ending in a 1 or a 10.

The "base cases" tell us that when $n = 0$, there is only the empty string. When $n = 1$, there are two strings 0 (corresponding to the case Z_0) or 1 (corresponding to $B_{00}^{(0)}Z_1$. When $n = 2$, there are four strings (00, corresponding to Z_0Z_0, the two strings in $B_{00}^{(1)}$ followed by 1, corresponding to $B_{00}Z_1$, and 10, corresponding to EZ_1Z_0).

For $n \geq 3$, we can count the size for the recursive cases. If we denote c_n as the number of strings of size n, we see $c_n = c_{n-1} + c_{n-2}$ (for $n \geq 3$) and the probability that a string belongs to B_{00} is $c_n/2^n$. A little thought shows that the numbers c_n follow the Fibonacci numbers. Thus, $c_n \sim \phi^n$, where $\phi = 1.618\ldots$ is the golden ratio (the largeest solution of $x^2 + x + 1 = 0$.)

As n grows, the expected number of steps before getting a sample is thus $(2/\phi)^n$, which grows exponentially.

We can alternately use the counts c_n to *directly* sample a solution: essentially, we pick one of the options, then recursively generate strings for each case and concatenate them. However, we should not pick them u.a.r., since each option may have a different number of solutions. We should pick them with probabilities proportional to the size of the sets each option represents. Thus, we should pick the last two options with probabilities c_{n-1}/c_n and c_{n-2}/c_n, respectively. These are the probabilities $c_{n-1}/c_n = 1/\phi$ and $c_{n-2}/c_n = 1/\phi^2$, using the above analysis.

The above example is not a coincidence. In general, the ability to count the number of solutions allows us to generate uniform samples.

Consider again the case of rooted binary trees. We can write down a recursion

$$T = E + Z.T.T$$

where Z is a singleton node. Informally, a tree is either empty (E) or is created by taking a node Z and attaching two trees $T.T$ to it.

We can write down a recursive expression for the number of trees

$$T_n = \sum_{k=1}^{n-1} T_k T_{n-k}$$

whose solution are the Catalan numbers.

Then, we can pick k proportional to

$$\frac{C_k C_{n-k-1}}{C_n}$$

to sample trees of size n recursively.

Note however that precomputing the Catalan numbers takes time that is not linear (Remy's algorithm was linear in n).

Exercise 3.1 The Motzkin trees are defined by the recursion

$$T = E + Z.T + Z.T.T$$

Solve the recurrence to get a sampling procedure for Motzkin trees.

3.2 Sampling from Combinatorial Problems

In order to formally discuss sampling problems for combinatorial problems, we start with some formalization. Let Σ be a finite alphabet, and let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on words over Σ. We interpret $(x, y) \in R$ as asserting that y is (an encoding of) a feasible solution to the instance x of a combinatorial problem.

For example, we can define a binary relation R as those (x, y) such that x encodes a universe of n elements and y is a subset or a permutation of those elements. More interestingly, x can encode a graph G_x and y can encode the edges of a spanning tree of G_x, or x can encode a Boolean formula and y can encode a satisfying assignment to the formula.

The specific encoding to these problems is not important, but we do require that the relation R can be verified efficiently—otherwise, even for a pair (x, y), we may not be able to efficiently identify if y is indeed a solution. This motivates the definition of p-relations.

Definition 3.1 (p-Relations) A relation $R \subseteq \Sigma^* \times \Sigma^*$ is a *p-relation* if there is a deterministic polynomial time Turing machine that recognizes R and if there is a polynomial p such that for all $x \in \Sigma^*$, we have $(x, y) \in R$ implies that $|y| \leq p(|x|)$. We define $R(x) = \{y \in \Sigma^* \mid (x, y) \in R\}$.

We associate the following natural problems with any p-relation R.

- **[Existence]** Given $x \in \Sigma^*$, does there exist $y \in \Sigma^*$ such that $(x, y) \in R$?
- **[Construction]** Given $x \in \Sigma^*$, construct a word y such that $(x, y) \in R$, if one exists.
- **[Counting]** Given $x \in \Sigma^*$, count the number of solutions, i.e., compute $|R(x)|$.
- **[Uniform Sampling]** Given x, generate uniformly at random a word y from $R(x)$.

Note that the existence problem for p-relations captures the class NP: indeed, for any language $L \in \mathsf{NP}$, we have a p-relation R and a polynomial function p such that $x \in L$ iff $R(x, y)$ for some $y \in \Sigma^*$, $|y| \leq p(|x|)$. Similarly, the counting problem captures the class #P. For problems in NP, access to an oracle for the existence problem enables one to construct an explicit witness with polynomial overhead.

In the rest of the chapter, we study the relationship between sampling and counting, in both an *exact* and an *approximate* setting.

Exercise 3.2 Consider the p-relation $R(\varphi, y)$, where φ is a Boolean formula with n variables, $y \in \{0,1\}^n$, and y is a satisfying assignment for φ. Show that given an oracle for SAT, we can solve the construction problem for R with polynomially many calls to the SAT oracle.

Solution 3.1 Note that if φ is satisfiable, then either $\varphi\,|_{x_1=0}$ is satisfiable or $\varphi_{x_1=1}$ is satisfiable (or both), where these formulas denote φ with variable x_1 set to 0 or 1, respectively. We build a satisfying assignment bit-by-bit. We ask the oracle if φ is satisfiable; if not, we stop and output there is no satisfying assignment. Otherwise, we ask if $\varphi\,|_{x_1=0}$ is satisfiable. If so, we set $x_1 = 0$ and continue recursively with this formula (which is over $n-1$ variables. If not, we know that $\varphi\,|_{x_1=1}$ is satisfiable, so we set x_1 to 1 and recursively continue. This requires $n+1$ calls to the SAT oracle.

Sampling from $R(x)$ is clearly at least as hard as constructing an element from $R(x)$. Thus, if $\{x \in \Sigma^* \mid R(x) \neq \emptyset\}$ is NP-complete for a relation R, then polynomial-time uniform sampling will imply NP = RP.

However, there are natural problems for which construction is computationally easier than uniform generation. That is, there are problems for which, given an input x, constructing a y such that $R(x, y)$ is in polynomial time, but uniform sampling in polynomial time implies NP = RP.

As an example, consider the relation $\{(G, C) \mid G$ ia a directed graph and C is a directed simple cycle$\}$. One can generate a directed cycle in a graph in polynomial time. However, there is no uniform sampling procedure for directed simple cycles unless NP = RP.

Theorem 3.1 *Intractability of Uniform Sampling Unless* NP = RP*, there is no randomized polynomial time algorithm that samples directed simple cycles uniformly at random.*

Proof Idea TODO

3.3 Counting Implies Uniform Sampling

3.4 Approximate Counting implies Approximate Sampling

3.5 Sampling Satisfying Assignments from DNF Formulas

3.6 Counting and Sampling from SAT

Chapter 4
Randomized Algorithms for SAT

♣ RM: have an additional section on random walks - random walks on undirected graphs: cover time
- derandomizing Schoening's algorithm
- PLS

4.1 Better Algorithms for 3SAT k-SAT

The *k-SAT problem* takes as input a Boolean formula in conjunctive normal form (CNF), where each clause has at most k literals, and asks if the given formula is satisfiable. In the following, we consider formulas over n Boolean variables. There is an exhaustive algorithm for k-SAT that iterates over all the 2^n assignments to the variables and evaluates the formula for each assignment. This gives an $O(2^n poly(n))$ algorithm for k-SAT. We show that the naive bound can be improved.

In particular, the 2SAT problem takes as input a CNF formula in which every clause has at most two literals and asks if the formula is satisfiable. 2SAT can be solved in deterministic polynomial time, in contrast to the NP-complete 3SAT problem.

Exercise 4.1 Show that 2SAT can be solved in polynomial time. [Hint: Create a graph whose nodes are the literals in the formula and draw an edge $u \to v$ if $(\neg u \vee v)$ is a clause. In this graph, check that there is no path from a literal to its negation. Argue why this algorithm is sound and complete and why it runs in polynomial time.] Show that #2SAT, the problem of *counting* the number of solutions to a 2SAT problem is #P-complete. Thus, the decision problem is easy but the counting problem is hard.

Since 3SAT is NP-complete, we cannot hope for a (randomized) polynomial time algorithm (unless RP = NP). But we can try to beat the naive $O(2^n poly(n))$ algorithm. We first show a randomized algorithm that runs in time $O((3/2)^t poly(n))$ on instances with at most t 3-clauses. The algorithm always returns "UNSAT" on unsatisfiable formulas and returns "SAT" on satisfiable formulas with high probability.

Let φ be a 3SAT instance with at most t 3-clauses. The algorithm proceeds as follows.

```
repeat for ℓ = 20 · (3/2)^t steps:
  let φ′ be the 2-CNF and 1-CNF clauses in φ
  for each 3-CNF clause C = v1 ∨ v2 ∨ v3 in φ,
    randomly pick a literal v_i from C and remove it, getting a new clause C′; add C′ to φ′
  solve the resulting 2SAT instance in polynomial time. If φ′ is satisfiable, return ''SAT''
return ''UNSAT''
```

Clearly, if the formula is unsatisfiable, the above algorithm will always return "UNSAT." Now, if φ' is satisfiable, then the same assignment will also satisfy φ, since φ' was obtained from φ by throwing away some literals from some clauses.

Claim Suppose φ is satisfiable and let A be a satisfying assignment. For each clause C of φ, we have

$$\Pr[A \text{ satisfies } C'] \geq 2/3$$

The reason is that, in the worst case, exactly one literal in C is true and we removed that particular literal with probability $1/3$. Note that if multiple literals of C were set to true by A, then C' will remain satisfiable.

Since we independently remove literals from each clause, we have that

$$\Pr[A \text{ satisfies } \varphi'] \geq (2/3)^t$$

If we run the loop ℓ times, the probability that none of the φ' are satisfied by A is at most $(1-(2/3)^t)^\ell$, and for the particular choice $\ell = 20 \cdot (3/2)^t$, the probability is bounded above by

$$(1-(2/3)^t)^{20\cdot(3/2)^t} \leq e^{-20} < 10^{-9}$$

Of course, randomization is not necessary to improve the naive upper bound. In the next exercise, we explore an algorithm based on *branching* that beat the 2^n bound. Branching algorithms pick variables in the formula and set them to true or false. Upon setting a variable, we can remove the clauses that are satisfied by that assignment, and reduce the size of the other clauses. We continue by setting other variables. If at any point, the formula is seen to be unsatisfiable, we backtrack to a previous setting and try a different assignment to the variable.

Exercise 4.2 The following algorithm beats the 2^n bound using the following observation. If a clause has k literals, then it can have at most $2^k - 1$ satisfying assignments, even though there are 2^k assignments to the k literals.

```
fn SAT(φ) // φ is a k-SAT instance
  if φ has no clauses, return ``SAT''
  if φ has an empty clause, return ``UNSAT''
  pick the shortest clause C = (v_1 ∨ ... ∨ v_ℓ) in φ
  for all the 2^ℓ − 1 satisfying assignments a ∈ {0,1}^ell to C:
    call SAT(φ[a])
    // φ[a] means the formula φ where v_1,...,v_ℓ are replaced by
    // corresponding values from a
    if the recursive call returns ``SAT'', return ``SAT''
  return ``UNSAT''
```

Show that the algorithm is sound and complete. For the running time, note that $\ell \leq k$, since each clause in the original formula has k clauses and setting literals to constants can only reduce the size of clauses. Thus, the run time satisfies the recurrence

$$T(n) \leq (2^k - 1)T(n-k) + O(poly(n))$$

from which we obtain

$$T(n) \leq (2^k-1)^{n/k} poly(n) = 2^n(1-1/2^k)^{n/k} \leq 2^n e^{-n/k2^k}$$

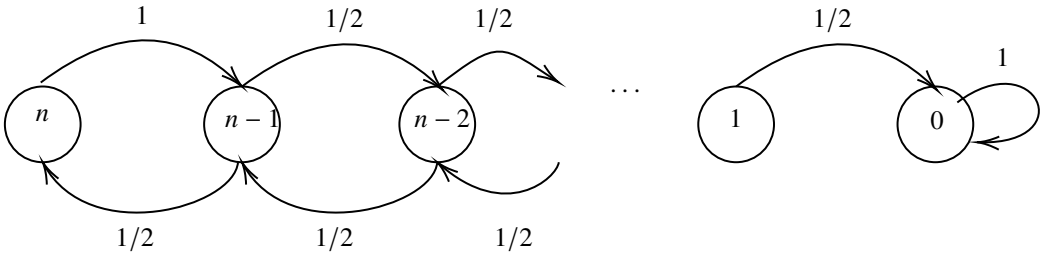
4.2 A Random Walk Algorithm for 2SAT

There is a different way to get a "better than 2^n" algorithm for k-SAT using a random-walk-based strategy. We show a simple random walk algorithm for 2SAT first, and then generalize the ideas to 3SAT and beyond.

The simple randomized algorithm for 2SAT starts with a random assignment and repeats the following step for ℓ times (where ℓ is a parameter we shall determine later): if the current assignment satisfies all clauses, stop and return "satisfiable;" otherwise, pick an arbitrary clause that is not satisfied, pick one of its literals uniformly at random, and switch its value in the current assignment. If the algorithm has not found a satisfying assignment after ℓ rounds, stop and return "maybe unsatisfiable."

Let us analyze the algorithm. First, note that each round takes polynomial time, so if ℓ is polynomial in the size of the formula, then the algorithm runs in polynomial time. Second, if the algorithm ever returns satisfiable, it has a witness assignment, so it is never wrong. What we do next is to bound the probability that the algorithm is wrong when it returns "maybe unsatisfiable."

The analysis of the algorithm is based on a random walk on a graph. Suppose that the formula is satisfiable, and let us fix a particular satisfying assignment A. We construct a graph with $n+1$ nodes, labeled 0 to n; the node i denotes that the current assignment differs from A in i positions. If we are in node 0, we agree with A and therefore we have found a satisfying assignment. On the other hand, if we are in node n, we disagree with A on the valuation of every variable. In each step, if we are in node i, then with probability at least $1/2$ we move to node $i-1$ by picking the "correct" literal and with probability at most $1/2$ we move to $i+1$ by flipping a literal that agreed with A. (When we are in node n, we move to $n-1$ with probability 1.) We can view the behavior of the algorithm as a random walk:



Our goal is to hit the node 0. Since we begin with a random assignment, we can start at any of the nodes. If we denote by h_i the expected number of steps to reach 0 for the first time starting from node i, we get: $h_0 = 0$, $h_n = h_{n-1} + 1$, and

$$h_i = 1/2h_{i-1} + 1/2h_{i+1} + 1$$

for all $i \in \{1, \ldots, n-1\}$. The solution to this recurrence is $h_i = n^2 - (n-i)^2$. Thus, no matter where we start in the graph, we expect to reach 0 in at most n^2 steps.

Exercise 4.3 Check.

Now, let us pick $\ell = 2n^2$ in the algorithm. Using Markov's inequality, the probability that we have not hit node 0 in ℓ steps is then

$$\Pr[h \geq 2n^2] \leq 1/2$$

Thus, the algorithm may err with probability at most $1/2$. By repeating the algorithm multiple times, or by increasing ℓ by a constant factor, we can reduce this probability.

Exercise 4.4 Does the above algorithm provide a uniform sample of satisfying assignments in case the formula is satisfiable?

4.3 Schöning's k-SAT Algorithm

Why would the same idea not work for 3SAT? The problem is that, for 3SAT, the probability that we make progress by picking the correct literal is $1/3$ and the probability of moving away from A is $2/3$. Thus, we have the following recurrences: $h_0 = 0$, $h_n = h_{n-1} + 1$, and

$$h_i = 2/3h_{i+1} + 1/3h_{i-1} + 1$$

By analyzing this random walk, where there is a bias to move away from 0, we find that the expected number of steps is exponential.

Exercise 4.5 Show that $h_i = 2^{n+2} - 2^{n-i+2} - 3i$ is a solution to the above recurrence.

Thus, if we start with a random assignment and perform a random walk, we expect to spend about 2^n steps. However, we can do better by the following observation: instead of one long walk, we can restart the walk after some time and

choose a new starting point. If we happen to pick a starting point close to zero, we are likely to hit zero with a short walk. This idea is made precise in Schöning's algorithm for k-SAT.

First, assume we pick an initial assignment uniformly at random. Then, the probability that we start in node i is given by

$$\Pr[X_0 = i] = \binom{n}{i}\left(\frac{1}{2}\right)^n$$

and the expectation is $E[x_0] = n/2$.

Schöning's algorithm works as follows.

```
repeat ℓ times (ℓ to be chosen appropriately)
  pick an initial assignment uniformly at random
  repeat for 3n steps
    perform literal flipping in an unsatisfied clause
    if satisfiable, return ''SAT''
return ''UNSAT''
```

Let us analyze this algorithm. In a random wal of $j+2k$ steps, the probability of exactly k left moves and $j+k$ right moves is

$$\binom{j+2k}{k}\left(\frac{2}{3}\right)^k\left(\frac{1}{3}\right)^{j+k}$$

Let q_j be a lower bound on the probability that a walk reaches 0 when started at j. We have

$$q_j \geq \max_{k\in\{0,\ldots,j\}}\binom{j+2k}{k}\left(\frac{2}{3}\right)^k\left(\frac{1}{3}\right)^{j+k}$$

Thus,

$$q_j \geq \binom{3j}{k}\left(\frac{2}{3}\right)^j\left(\frac{1}{3}\right)^{2j}$$

Let us apply Stirling's approximation:

$$\binom{3j}{j} \sim \frac{\sqrt{2\pi 3j}}{4\sqrt{2\pi j}\sqrt{2\pi(2j)}} \sim const\cdot\frac{1}{\sqrt{j}\left(\frac{27}{4}\right)^j}$$

Applying this approximation to q_j, we get

$$q_j \geq const\frac{1}{\sqrt{j}}\frac{1}{2^j}$$

and $q_0 = 1$.

The expected number of steps to reach node zero is then

$$q \geq \sum_{j=0}^{n}\Pr[X_0 = j]q_j \geq \frac{1}{2^n} + \sum\binom{n}{j}\left(\frac{1}{2}\right)^n const\frac{1}{\sqrt{j}\frac{1}{2^j}}$$
$$\sim const\frac{1}{poly(n)}\left(\frac{3}{4}\right)^n$$

Thus, the expected number of rounds is approximately $1/q \sim 1.34^n$.

4.4 Notes

There is a rich literature on deterministic and randomized algorithms for k-SAT. Exercise 4.2 is from lecture notes by Ryan Williams. Using a better branching heuristic, [**?**] show a better $2^{n(1-1/O(2^k)}$ bound for k-SAT.

The culmination of a number of advances in SAT are algorithms that run in time $2^{n(1-1/O(k)}$ time. A famous conjecture (*strong exponential time hypothesis*) states that this is the best dependence on k.

Definition 4.1 (SETH) For every $\epsilon > 0$ there exists a k such that k-SAT on n variables cannot be solved in $O(2^{(1-\epsilon)n})$ time.

SETH is a strengthening of ¶ ≠ NP: it states that k-SAT requires exponential time, not just super-polynomial time. Thus, proving SETH will show ¶ ≠ NP. On the other hand, refuting SETH will provide new and faster algorithms for SAT as well as a number of other results in combinatorial algorithms. *Fine-grained complexity* is a field of complexity that considers the relations between SETH and other algorithms, among other questions.

The randomized 2SAT algorithm is from [**?**].

References

1. Fan, C., Muller, M., Rezucha, I.: Development of sampling plans by using sequential (item by item) selection techniques and digital computers. Journal of the American Statistical Association **57**(298), 387–402 (1962)
2. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science. A foundation for computer science. Addison-Wesley (1994). URL https://books.google.de/books?id=cjgPAQAAMAAJ
3. Majumdar, R., Niksic, F.: Why is random testing effective for partition tolerance bugs? PACMPL **2**(POPL), 46:1–46:24 (2018). DOI 10.1145/3158134. URL https://doi.org/10.1145/3158134