Advanced Algorithms.
Probability and Randomized Algorithms
Continued

3-SAT: How to Satisfy Most Clauses

The Satisfiability problem (SAT) asks to assign truth values to the variables in a boolean formula so as to make the formula true. Specifically, the formula is given as a conjunction of clauses, where each clause is a disjunction of literals, i.e., unnegated or negated boolean variables. SAT appears directly in many real problem settings where logical variables have to satisfy certain constraints. In 3-SAT, every clause has 3 literals. 3-SAT is a classical NP-complete problem. MAX 3-SAT is the following natural relaxation of 3-SAT: If the formula is not satisfiable, find an assignment of truth values that satisfies as many clauses as possible. By an obvious reduction from 3-SAT we see that MAX 3-SAT is also NP-complete.

On the other hand, if any conjunction of k clauses with exactly 3 literals is given, we can easily find an assignment that satisfies most of the clauses, namely $0.875k$ in expectation. An extremely simple randomized algorithm will do: Assign truth values 0 or 1, each with probability 1/2, to all variables independently. The analysis is very simple, too: Every clause is satisfied with probability 7/8, hence, by linearity of expectation, an expected number of 7/8 of all clauses is satisfied.

We can conclude more from this result: Since an expected number of 7/8 of all clauses is satisfied, there must always exist some truth value assignment that actually satisfies at least 7/8 of the clauses. This easily follows from a general argument: Consider the random variable X indicating the number of satisfied clauses. The expected value $E[X]$ is the average value of X, weighted by the probabilities of values. Hence any random variable X can take on some value greater than or equal to $E[X]$.

This reasoning is the famous Probabilistic Method: When we look for a certain combinatorial structure (here: a truth assignment satisfying many clauses), we may apply some simple randomized algorithm and show that the desired structure is produced with some positive probability. Hence this structure must exist. Of course, the approach does not work for any such problem (due to lack of a simple randomized algorithm), and it proves only the existence of the thing we are looking for, but it does not say how we can find it efficiently. These questions must be studied for any specific search problem at hand.

In the case of MAX 3-SAT, how difficult is it to actually find an assignment that satisfies at least 7/8 of the clauses? The obvious idea is to iterate the above algorithm until success. We analyze the expected number of iterations needed. Let p_j be the probability that exactly j clauses are satisfied. Since the expected value of j is $7k/8$, we have the following equality, where the sum is already split in two cases:

$$7k/8 = \sum_j jp_j = \sum_{j<7k/8} jp_j + \sum_{j\geq 7k/8} jp_j.$$

As an abbreviation define $p := \sum_{j\geq 7k/8} p_j$, and let k' be the largest integer with $k' < 7k/8$. We upperbound the sum generously and obtain

$$7k/8 \leq \sum_{j<7k/8} k'p_j + \sum_{j\geq 7k/8} kp_j = k'(1-p) + kp \leq k' + kp.$$

Thus we have $kp \geq 7k/8 - k'$, which is at least $1/8$ due to the definition of k'. Thus, a random assignment succeeds with probability $p \geq 1/8k$, and the expected waiting time for success is at most $8k$ iterations.

Note that this is a Las Vegas algorithm. Furthermore, note that it does not solve the actual MAX 3-SAT problem. It guarantees only $0.875k$ satisfied clauses in every input. But what if, for example, $0.95k$ clauses are satisfiable ...? In fact, it has been shown that, for any small $\epsilon > 0$, it is already NP-complete to decide whether a MAX 3-SAT instance allows to satisfy $(0.875 + \epsilon)k$ clauses. In this sense, running the simple randomized algorithm is already the best one can do in general.

Hashing

(This part may be skipped if you know hashing already very well from Data Structure courses. But make sure that you also understand the probability theory behind it.)

Let be U a universe (a huge set) of elements. A dictionary is a data structure that keeps track of a set $S \subset U$ and supports the following operations: insert, delete, lookup. That is, a dictionary enables us to quickly insert elements into a set, delete elements from a set, or retrieve elements of the set. Hash tables are among the most well known implementations of dictionaries. In the following, n is always some fixed size bound much smaller than $|U|$. A hash table H is an array of size n, with indices $0, \ldots, n-1$, where $n \geq |S|$. That is, H allocates enough space for storing sets S of at most n elements. However, several elements may be stored in the same entry of H, for example as a list. Then we speak of collisions.

A hash function h maps U onto this index set. In order to execute any of the dictionary operations for an element, we compute the index of that element and access the corresponding entry of H. Of course, h must be easily computable, and it is essential that our hash function keeps collisions to a minimum: If many elements are stored in the same entry, we still have to search for the desired element there, and this would slow down the dictionary operation. Since U is much larger than n, collisions cannot be avoided, but with a good randomized approach we can keep their expected number small. In the following, note again that randomness is only in the algorithm (here: in the design of our hash function h), but we do not make any assumptions on the set S we want to store, other than $|S| \leq n$.

Here is a classical simple hashing scheme, along with a rigorous analysis of its performance. We will choose h at random from a certain class of easily computable functions. We call a function class "universal" if for any pair $u, v \in U$ the probability of $h(u) = h(v)$ is at most $1/n$. This is a good property for hashing because, if we pick a random h from a universal class then, for any fixed element u, the expected number of other elements $s \in S$ with $h(s) = h(u)$ is at most 1, and we barely get large bags of elements in the same entry of H. Thus our dictionary will be able to do any operation in $O(1)$ expected time.

But do such universal classes of functions exist? Trivially, the class of all functions from U into the index set has this property. But what would it mean to choose a random h from the class of all functions? Since the values of such h are random and independent, h has "no structure", and we can "compute" the values of h for given elements only by looking them up, in a table of size $|U|$, which is against the very idea of hashing. We need a restricted class of functions which are easily computable but still "shake

well" the elements of any subset with at most n elements. One construction comes from elementary number theory.

We choose a prime number p slightly larger than our n. (Prime numbers are "dense enough" in the set of integers, we will always find such p. We do not go into details of this preprocessing step.) We represent the elements of U as vectors $x = (x_1, \ldots, x_r)$ with $0 \leq x_i < p$ for all i. The dimensionality we need is clearly $r \approx \log |U| / \log p$. (This may look complicated, but note that these vectors can be seen as arbitrary "names" of the elements.) For every $a = (a_1, \ldots, a_r)$ we define a hash function $h_a(x) = (\sum_{i=1}^{r} a_i x_i) \bmod p$. For any given $x \in U$ these values are really easy to compute. It remains to analyze the collisions. We will see that the class of all functions h_a is universal. Very little help from number theory is needed: If p is a prime and $z \neq 0 \bmod p$, then $az = bz \bmod p$ implies $a = b \bmod p$ for any two numbers a, b. (The proof is straightforward.)

Using this fact we show, for any two $x, y \in U$, that $h_a(x) = h_a(y)$ happens with probability at most $1/p$. (Recall where this probability comes from: We took some random a.) Since $x \neq y$, their vectors must differ somewhere. Hence, let j be some position where $x_j \neq y_j$. A nice trick makes the probability calculation extremely simple: Instead of considering a random a, we fix all a_i, $i \neq j$, and choose only a_j randomly, where $0 \leq a_j < p$. Then the probability result applies also to a random vector a. (Why?) By the construction of h_a, a collision $h_a(x) = h_a(y)$ appears if and only if $a_j(y_j - x_j) = \sum_{i \neq j} a_i(x_i - y_i) \bmod p$. Since we have fixed the right-hand side, we can treat it as a constant, say m. Now define $z := y_j - x_j$. Due to the above number-theoretic fact, there exists exactly one a_j with $a_j z = m \bmod p$. Hence the probability of collision is $1/p \leq 1/n$, and our hash table can execute dictionary operations in $O(1)$ expected time.

A final remark: There is often confusion about the time complexity of hash table operations. $O(1)$ is the expected number of arithmetic operations. But the bit complexity is not constant, it grows logarithmically in the size of the sets we want to deal with. Thus, hash tables are asymptotically not faster than other dictionary implementations such as balanced search trees. The real advantage of hash tables is elsewhere: They are easy to implement (just evaluation of some simple functions) and use only arithmetic, which is physically faster than manipulations with pointers, etc., that would be needed to implement trees.

Closest Points

For the problem of finding a closest pair of n points in the plane there exists a divide-and-conquer algorithm running in $O(n \log n)$ time. It follows a simple idea but is a bit complicated when it comes to the implementation details. Here we show a Las Vegas algorithm that is not only simpler but also solves the problem already in $O(n)$ expected time plus $O(n)$ dictionary operations.

We can always assume that our n points are in a unit square. In our algorithm we maintain a real number d which is the smallest distance between two points known so far. We consider the n points in random order. For every new point p we test whether p has distance smaller than d to some earlier point, and in this case we update d. For an efficient test we have to avoid computing the distances to *all* earlier points. Therefore we divide the unit square into squares of side length $d/2$. Since d is the smallest distance, at most one earlier point can be located in each square. Moreover, those points which might have a distance smaller than d to p are in squares close to the square containing p, more precisely, they are in a 5×5 grid of squares. Thus we have to test at most 25 candidates in every step. Hence $O(n)$ computations are enough, for all n points. So far we have not even used the fact that points are processed in random order.

However, some complications begin here: We need to know which points are in the candidate squares! For this purpose we may use a hash table, with an entry for every point. But whenever d is diminshed, our partitioning into squares of side length $d/2$ changes totally, and we have to create a new hash table from scratch. How often do we have to insert our points into the various hash tables? Only here the randomized order of points becomes important.

Let X be a random variable for the total number of insertions. Let X_i be another random variable, with $X_i = 1$ if the ith point causes an update, and $X_i = 0$ else. Clearly, $X = n + \sum_i iX_i$. The key fact is that $X_i = 1$ with probability at most $2/i$: For each i, the first i points are randomly ordered as well, hence, the event that some of the two points in a closest pair is the ith point has probability $2/i$. Linearity of expectation gives $E[X] = n + \sum_i iE[X_i] \leq 3n$. Thus, the expected number of dictionary operations is $O(n)$, and each of them needs $O(1)$ expected time. From these two facts it follows that the total expected time is $O(n)$. Stop! The latter conclusion seems obvious at first glance. But referring to linearity of

expectation is not enough here, since the number of random variables to be added is a random variable itself. A real proof needs a careful analysis of conditional expectations, since we combine here two different sources of randomness. – However we omit this technical part of the proof. We only wanted to stress the efficiency and elegance of a randomized approach.

Chernoff Bounds

This is a very useful general tool to bound the probabilities that certain random variables deviate much from their expected values. Here we will derive one version of this bound and then apply it to a simple load balancing problem. (You do not need the proof when you *apply* the bound, but why not see it once? It is pretty nice and elegant.)

Let X be sum of n independent 0-1 valued random variables X_i taking value 1 with probability p_i. Clearly $E[X] = \sum_i p_i$. For any $\mu \geq E[X]$ and $\delta > 0$ we ask how likely it is that $X > (1+\delta)\mu$, in other words, that X exceeds the expected value by more than 100δ percent.

Since function exp is monotone, this inequality is equivalent to $\exp(tX) > \exp(t(1+\delta)\mu)$ for any $t > 0$. Exponentiation and this free extra parameter t seem to make things more complicated, but we will see very soon why they are useful.

For any random variable Y and any number $\gamma > 0$ we have $E[Y] \geq \gamma Pr(Y > \gamma)$. This is known as Markov's inequality and follows directly from the definition of $E[Y]$. For $Y := \exp(tX)$ and $\gamma = \exp(t(1+\delta)\mu)$ this yields $Pr(X > (1+\delta)\mu) \leq \exp(-t(1+\delta)\mu)E[\exp(tX)]$.

Due to independence of the terms X_i we have:

$E[\exp(tX)] = E[\exp(\sum_i tX_i)] = E[\prod_i \exp(tX_i)] = \prod_i E[\exp(tX_i)] = \prod_i (p_i e^t + 1 - p_i) = \prod_i (1 + p_i(e^t - 1)) \leq \prod_i \exp(p_i(e^t - 1))$
$= \exp((e^t - 1)\sum_i p_i) \leq \exp((e^t - 1)\mu)$.

This gives us the bound $\exp(-t(1+\delta)\mu)\exp((e^t-1)\mu)$. We can arbitrarily choose t. With $t := \ln(1+\delta)$ our bound reads as $\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$.

The base depending on δ looks a bit complicated, however: Using $e^\delta \approx 1 + \delta$ one can see that the base is smaller than 1. For any fixed deviation δ the base is constant, and the bound decreases exponentially in μ. The more independent summands X_i we have in X, the smaller is the probability of large deviations. A direct application of the simple Markov inequality would be much weaker (therefore the detour via the exponential function).

For small δ one can use the Taylor expansion $e^\delta = 1+\delta+\ldots$ and simplify the bound to $e^{-\delta^2\mu/3}$, which is a more common form of a Chernoff bound, and nicer to use in most applications. (The detailed calculations are omitted here.)

In order to show at least one application, consider the following simple load balancing problem: m jobs shall be assigned to n processors, in such a way that no processor gets a high load. In contrast to the Load Balancing problem we studied earlier, no central "authority" assigns jobs to processors, but every job chooses a processor by itself. We want to install a simple rule yet obtain a well balanced allocation. (An application is distributed processing of independent tasks in networks.) To make the rule as light-weight as possible, let us choose for every job a processor randomly and independently. The jobs need not even "talk" to each other and negotiate places. How good is this policy?

We analyze only the case $m = n$. What would you guess: How man jobs end up on the same processor? To achieve clarity, consider the random variable X_i defined as the number of jobs assigned to processor i. Clearly $E[X_i] = 1$. The quantity we are interested in is $Pr(X_i > c)$, for a given bound c. Since X_i is a sum of independent 0-1 valued random variables (every job chooses processor i or not), we can apply the Chernoff bound. With $\delta = c - 1$ and $\mu = 1$ we get immediately the bound $e^{c-1}/c^c < (e/c)^c$.

But this is only the probability bound for one processor. To bound the probability that $X_i > c$ holds for *some* of the n processors, we can apply the union bound and multiply the above probability with n. Now we ask: For which c will $n(e/c)^c$ be "small"?

At least, we must choose c large enough to make $c^c > n$. As an auxiliary calculation consider the equation $x^x = n$. For such x we can say (1) $x \log x = \log n$ and (2) $\log x + \log\log x = \log\log n$, we have just taken the logarithm twice. Equation (2) easily implies $\log x < \log\log n < 2\log x$. Division by (1) yields $1/x < \log\log n/\log n < 2/x$. In other words, $x^x = n$ holds for some $x = \Theta(\log n/\log\log n)$.

Thus, if we choose $c := ex$, our Chernoff bound for every single processor simplifies to $1/x^{ex} < 1/(x^x)^2 = 1/n^2$. What this shows is that, with probability $1 - 1/n$, each processor gets $O(\log n/\log\log n)$ jobs. This answers our question: Under random assignments, the maximum load can be logarithmic, but it is unlikely to be worse.

For $m = \Theta(n \log n)$ or more jobs, the random load balancing becomes

really good. Then the load is larger than twice the expected value $\Theta(\log n)$ only with probability below $1/n^2$. Calculations are similar as above.

To see another (however very technical) example of the use of Chernoff bounds in computer science, it is recommended to have a look at the following article, where the sum of a huge set of numbers is approximately computed from a random sample.

B. Fu, W. Li, Z. Peng: Sublinear Time Approximate Sum via Uniform Random Sampling. Computing and Combinatorics, 19th International Conference, COCOON 2013, Lecture Notes in Computer Science (Springer), vol. 7936, pp. 713–720.