Randomized Algorithms 2016/7A
Lecture 1

Min Cut Algorithm, Closest Pairs, (Multi)-Set Equality [*]

Moni Naor

The lecture introduced randomized algorithms. Why are they interesting? They may solve problems faster than deterministic ones, they may be essential in some settings, especially when we want to go to the sublinear time complexity realm[1], they are essential in distributed algorithms e.g. for breaking symmetry, they yield construction of desirable objects that we do not know how to build explicitly and are essential for cryptography[2] and privacy[3]. Another type of study is to analyze algorithms when assuming some distribution on the input, or some mixture of worst case and then a perturbation of the input (known as *smoothed analysis*). But our emphasis would be worst case data where the randomness is created independently of it. That is we assume the algorithm or computing device in addition to the inputs gets also a random 'tape' (like the other tapes of the Turing Machine, but this one with truly random symbols). One nice feature that some randomized algorithms have is that they are simple. We demonstrated this in three algorithms in three different scenarios.

Randomized algorithms existed for a long time, since the dawn of computing (for instance the numerical "Monte Carlo Method"[4] from the 1940's or Shannon's work [9], also from that time.

Later, when people started arguing rigorously about the running time of programs, the idea of complexity classes of probabilistic machines emerged. We mentioned three such classes: RP, $Co-RP$ and BPP. The common conjecture is that $P = BPP$ and it is known to hold under the assumption that there are sufficiently good pseudo-random generators, that is a function $G: \{0,1\}^* \mapsto \{0,1\}^*$ that stretches its input significantly and the output cannot be distinguished from random. The weakest assumption under which the latter exist is that E (the class of deterministic $2^{O(n)}$) has a problem with *circuit complexity* $2^{\Omega(n)}$ [4].

One of the proponents of the power of randomized algorithms in the 1970s was Michael Rabin who published a very influential paper [7] on randomized algorithms. In that paper he gave a

[*]These notes summarize the material covered in class, usually skipping proofs, details, examples and so forth, and possibly adding some remarks, or pointers. In the interest of brevity, most references and credits were omitted.

[1]For instance, the famed PCP Theorem, that states that every NP statement can be verified using a few queries *must* use randomness for picking the queries. Another area is property testing.

[2]Where no task is possible without good randomness

[3]Differential privacy is a notion for sanitizing data that involves necessarily randomization, e.g. adding noise to an aggregate of a population.

[4]Do not confuse with the term "Monte Carlo Algorithm" which is a general name for an algorithm whose running time is deterministic (usually polynomial) but may err.

randomized algorithm for testing the primality of a number (based on Miller's deterministic test which assumed the 'Extended Riemann Hypothesis') that ran in polynomial time as well as a linear expected time algorithm for finding the closest pair of points from a given set. The primality test was (one directional) 'Monte Carlo' - it would always output 'prime' on a prime input and would output 'non-prime' with high probability on a composite. Since then several randomized algorithms for primality have been discovered as well as a deterministic one (see Schoof [8]). The fact that fast algorithms for primality exist made the RSA cryptosystem (suggested not long after and based on picking two random primes P and Q and making public their product $N = P \cdot Q$) feasible.

A simple example we mention was for checking matric multiplication: given three $n \times n$ matrices A, B and C how do you check that $A \cdot B = C$, say over the finite field $GF[2]$? To recompute the product $A \cdot B$ is relatively expensive (the asymptotic time it takes is denoted as $O(n^{\omega})$ where the current (as of 2014) best value for ω is ≈ 2.3728639), but in 1977 Freivalds suggests and $O(n^2)$ algorithm for verification: picking at random a vector $r \in \{0,1\}^n$ and compute $A(Br))$ and Cr and compare the two resulting vectors. If $AB = C$ then the algorithms always says 'yes' and if $A \cdot B \neq C$ then the algorithm says 'no' with probability at least $1/2$.

This algorithm is a good example for the theory of program checking.

The Minimum Cut Problem

The algorithm we saw demonstrates simplicity in a spectacular way. No need for flows, just pick a random edge and contract! The min-cut algorithm is due to Karger from SODA 1993. There is a faster version with Stein where the repetition is done in a clever way (i.e. not starting from scratch each time), yielding a near $O(n^2)$ algorithm [2]

Question: What happens if instead of picking a random edge you pick at random a pair of vertices and contract? Is the resulting algorithm a good min-cut algorithm?

The analysis of the algorithm had the form of analyzing the probability of a bad event in set i of the algorithm, given that a bad event had not occurred so far (the bad event was picking an edge from the cut). IF that probability has an upper bound of P_i, then the probability of a bad event ever occurring is bounded by $\Pi_{i=1}^{n} P_i$.

Another important idea we discussed is amplification. Given an algorithm which has some small probability of success, but running it many times, as a function of the probability, we can get a high probability of success. In this case the basic algorithm had probability $1/n^2$ of finding the min-cut, so after running it n^2 time and taking the best (smallest cut) we have probability $(1-1/n^2)^{n^2} \approx 1/e$. Repeating it a few more times gets us high probability of success.

Multiset equality

The problem we addressed can be viewed as a 'streaming' one. We have two multi-sets A and B and they are given in an arbitrary order. Once an element is given it cannot be accessed again (unless it is explicitly stored) and our goal is to have a low memory algorithm. We required a family of functions H that was incremental in nature, in the sense that for a function $h \in H$:

- Given $h, h(A)$ and an element x it is easy to compute $h(A \cup \{x\})$.
- For any two different multi-sets A and B the probability over the choice of $h \in H$ that $h(A) = h(B)$ is small.
- The description of h is short and the output of h is small.

The function we saw was based on treating the set A as defining a polynomial $P_A(x) = \Pi_{a \in A}(x - a)$ over a finite field whose size is larger than the universe from which the elements of A are chosen (say a prime $Q > |U|$). The member of the family of functions is called h_x for $x \in GF[Q]$ and defined as $h_x(A) = P_A(x)$. The probability that two sets collide (i.e. $h_x(A) = h_x(B)$, which in turn means that $P_A(x) = P_B(x)$) is $\max\{|A|, |B|\}/Q$, since this is the maximum number of points that two polynomials whose degree is at most $\max\{|A|, |B|\}$ can agree without being identical.

Storing h_x and storing $h(A)$ as it is computed requires just $O(\log Q)$ bits, so the resulting algorithm never needs to store anything close size to the original sets.

Closest Points in the Plane

Rabin's closest pair algorithm was a Las Vegas type algorithm, i.e. it never outputs a wrong result but the run time may take longer than expected. The algorithm we saw in class is much later and is Due to Golin et al. [1]. The analysis was in expectation and was based on the probability that we will need to rebuild the dictionary from the beginning, which was $2/i$ in the ith phase. There is a good description of it in Kleinberg and Tardos's book [3]. (you can read a description of Rabin's algorithm, which was also based on constructing a grid, in Lipton's blog [5]). The algorithm uses the floor ($\lfloor x \rfloor$) operation to find the square in the grid and hence does not fit the model used by most algorithms studied in computational geometry.

To complete the algorithm we need a *good hash table* to store the non-vacant grid cells. Thus, the algorithm yields yet another motivation for having dictionaries with $O(1)$ per operation. How to build the hash table will be discussed in future lectures.

Question: What happens if each time a new (potential) closest pair is discovered you pick at random a new order of the remaining nodes (including the old ones) and insert them to the data structure until a new closest point is discovered?

Nontransitive Dice

We saw (physical) dice that that are nontransitive (invented by Bradley Efron, see [6]): for two dice A and B and consider the event the outcome of A is larger than that of B (that we assume that the dice are fair in the sense that each side is equally likely and each side has a certain number of dots (pips)). If the probability that this happens is more than half we say that A dominates B. The point of the example is that even though if we look at the **expectation** as representing a

random variable, then the relation $E[A] > E[B]$ is clearly transitive. But the domination relation is not, as the following dice show:
A (purple): 4, 4, 4, 4, 0, 0
B (yellow): 3, 3, 3, 3, 3, 3
C (red): 6, 6, 2, 2, 2, 2
D (green): 5, 5, 5, 1, 1, 1
We have that A dominates B that dominates C that dominates D that dominates A. Also A dominates C and B and D are equally likely to win. So here if the first player picks a die the second player has a better die she can pick.

References

[1] Mordecai Golin, Rajeev Raman, Christian Schwarz and Michiel Smid, *Randomized Data Structures For The Dynamic Closest-Pair Problem*, SIAM J. Comput., vol. 26, no. 4, 1998.

[2] David Karger and Clifford Stein, *A new approach to the minimum cut problem.* Journal of the ACM 43 (4): 601, 1996.

[3] Jon Kleinberg and Eva Tardos, **Algorithm Design**. Addison Wesley, 2006. The relevant chapter: `http://www.aw-bc.com/info/kleinberg/assets/downloads/ch13.pdf`

[4] Russell Impagliazzo and Avi Wigderson, *P = BPP if E Requires Exponential Circuits: Derandomizing the XOR Lemma,* STOC 1997, pp. 220–229.

[5] Dick Lipton's blog, "Rabin Flips a Coin", March 2009
`https://rjlipton.wordpress.com/2009/03/01/rabin-flips-a-coin/`

[6] Ivars Peterson, *Tricky Dice Revisited*, Science News 2002.

[7] Michael Oser Rabin, *Probabilistic algorithms.* In Algorithms and complexity: New Directions and Recent Results, pages 21-39. Academic Press, New York.

[8] Rene Schoof, *Four primality testing algorithms*, `http://arxiv.org/abs/0801.3840`

[9] Claude Shannon, *A Mathematical Theory of Communication.* Bell System Technical Journal 27 (3): 379-423, (July/October 1948).
`http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf`