ADVANCED ALGORITHMS I (SPRING 2020)

CHIHAO ZHANG

This course is mainly about the use of random bits to design and analyze algorithms. Today I will show you some classical examples that can, hopefully, demonstrate the power of randomness.

1. Polynomial Identity Testing

1.1. **Univariate Polynomials.** We are given two polynomials $F(x), G(x) \in \mathbb{F}(x)$ where $\mathbb{F}$ is some field. The problem is to decide whether $F(x) \equiv G(x)$. We consider $\mathbb{F} = \mathbb{R}$ or $\mathbb{F} = \mathbb{C}$. The case when $\mathbb{F}$ is a finite field is more subtle.

While formally defining a computational problem, one needs to explicitly specify how the input looks like. In our case, we have to first agree on how the input polynomials $F(x)$ and $G(x)$ are encoded: If both polynomials are represented by their coefficients, namely $F(x) = \sum_{i=0}^{d} f_i \cdot x^i$ and $G(x) = \sum_{i=0}^{d} g_i \cdot x^i$, then the identity testing is a trivial task as one scanning of all coefficients suffices. In practice, we might have following two ways to encode polynomials:

(1) a polynomial is represented as products of polynomials, e.g. $F(x) = (x-1)(x-2)(x+3)$; or
(2) a polynomial is treated as an *evaluation oracle*, namely given any number $s \in \mathbb{F}$, there is a black box who can tell you the value of $F(x)$.

Now we assume that $F(x)$ is given as $F(x) = \prod_{i=1}^{d}(x - a_i)$ and $G(x) = \sum_{i=0}^{d} b_i \cdot x^i$. How to decide whether $F(x) \equiv G(x)$? A basic strategy is to expand $F(x)$ and compute all the coefficients. If we assume that the multiplication and addition of two elements in $\mathbb{F}$ cost constant time, a straightforward implementation requires $O(d^2)$ time. One can use Fast Fourier Transform [3] to accelerate the polynomial multiplication, so that the overall running time can be reduced to $O(d \log d)$.

If random bits are allowed to use, we can decide whether $F(x) \equiv G(x)$ in linear time, at the cost of making mistakes with very low probability, say 0.01%. The idea is simple: randomly choose a number $r \in U \subseteq \mathbb{F}$ for some set U, and test whether $F(r) = G(r)$. If the equality holds, then output $F(x) \equiv G(x)$, otherwise output $F(x) \not\equiv G(x)$. The evaluation operation costs $O(d)$ time.

It is clear that if $F(x) \equiv G(x)$, our algorithm always outputs the correct answer. On the other hand, if $F(x) \not\equiv G(x)$, the algorithm might make mistakes. It errs whenever the random number r is a root of the polynomial $F(x) - G(x)$. To analyze the chance of this event, we need the *Fundamental Theorem of Algebra*.

Theorem 1 (Fundamental Theorem of Algebra)**.** *Every non-zero polynomial $F(x) \in \mathbb{C}(x)$ of degree d has, counted with multiplicity, exactly d roots in $\mathbb{C}$.*

Therefore, since $F(x) - G(x)$ is a polynomial of degree at most d, if the set U is chosen to be sufficiently large, say $|U| \geq 100d$, then

$$\Pr_{r \in_R U}[F(r) - G(r) = 0] \leq \frac{d}{|U|} \leq \frac{1}{100}.$$

There are at least two ways to further decrease the error probability above. We can either enlarge the set U or to independently run the algorithm many times. Suppose we independently sample t numbers $r_1, \ldots, r_t \in U$ and look at $F(r_1) - G(r_1), \ldots, F(r_t) - G(r_t)$. As long as one of $F(r_i) - G(r_i)$ is not zero, we can safely claim that $F(x) \not\equiv G(x)$. Therefore, our algorithm fails only when all $r_1, \ldots, r_t$ are roots of $F(x) - G(x)$. This happens with probability at most 100^{-t}. So t independent repetitions dramatically reduce the failure probability, at the cost of using $O(td)$ time.

Date: Last modified on Mar 18, 2020.

1.2. **Multivariate Polynomials.** The benefit of this simple randomized algorithm turns out to be more prominent in the multivariate case than the univariate case. We are given two polynomials $F(x_1, \ldots, x_n), G(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ encoded as product of small polynomials, e.g. $F = (x_1 - x_2)(x_2 + x_3)(2x_3 - x_4)$. We still fix a set $U \subseteq \mathbb{F}$ and independently sample n random numbers $r_1, \ldots, r_n \in U$ uniform at random. The algorithm outputs $F \equiv G$ if and only if $F(r_1, \ldots, r_n) = G(r_1, \ldots, r_n)$.

As there is no direct generalization of the fundamental theorem of algebra in the multivariate case, we need the following theorem who directly implies the failure probability.

Theorem 2 (Schwartz-Zippel Theorem). *Let $Q \in \mathbb{C}[x_1, \ldots, x_n]$ be a non-zero multivariate polynomial of degree at most d. For any set $U \subseteq \mathbb{F}$, it holds that*

$$\Pr_{r_1,\ldots,r_n \in_R U} [Q(r_1, \ldots, r_n) = 0] \leq \frac{d}{|U|}.$$

Proof. We prove by induction on n. The case $n = 1$ corresponds to the univariate case and therefore follows from the fundamental theorem of algebra. Now assume the theorem holds for smaller n. Without loss of generality we assume the degree of x_1 is at least 1, then the polynomial Q can be viewed as a univariate polynomial in x_1 and written as

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i \cdot Q_i(x_2, \ldots, x_n) =: P_{x_2,\ldots,x_n}(x_1),$$

where k is the maximum degree of x_1 and $Q_i(\cdot)$ is the coefficient of x_1^i. Therefore, for any fixed $x_2, \ldots, x_n$, $P_{x_2,\ldots,x_n}(x_1)$ is a polynomial in x_1 of degree $k \geq 1$. To simplify the notation, we use $\mathcal{A}$ and $\mathcal{B}$ to denote the events "$Q = 0$" and "$Q_k = 0$" respectively. Then

$$\begin{aligned} \Pr[\mathcal{A}] &= \Pr[\mathcal{A} \cap \mathcal{B}] + \Pr[\mathcal{A} \cap \neg\mathcal{B}] \\ &\leq \Pr[\mathcal{B}] + \Pr[\mathcal{A} \mid \neg\mathcal{B}] \end{aligned}$$

Since Q_k is a polynomial of degree $d-k$ in $n-1$ variables, we can apply induction hypothesis and obtain $\Pr[\mathcal{B}] \leq \frac{d-k}{|U|}$. On the other hand, conditional on that $Q_k \neq 0$, for any fixed $x_2, \ldots, x_n$, $P_{x_2,\ldots,x_n}(x_1)$ is a univariate polynomial of degree k. So $\Pr[\mathcal{A} \mid \neg\mathcal{B}] \leq \frac{k}{|U|}$. In conclusion,

$$\Pr[\mathcal{A}] \leq \frac{d-k}{|U|} + \frac{k}{|U|} = \frac{d}{|U|}.$$

□

In fact, there is no known deterministic polynomial-time algorithm for polynomial identity testing for multivariate polynomials. The fact justifies the usefulness of random bits in the algorithm design.

2. Karger's Min-Cut Algorithm

Given an undirected connected graph $G(V, E)$, a cut is a set of edges $C \subseteq V$ whose removal disconnects the graph. The Min-Cut problem is to a cut with minimum cardinality.

You might have learnt that the problem can be solved in polynomial-time using a max-flow based algorithm: Fix a source s and enumerate all possible sinks t. For each pair of the source and the sink (s, t), one can compute the max-flow between s and t. Let S be the set of vertices connected to s after removing all edges with flow. Then a min-cut between s and t is those edges between S and its complement, denoted by $C_t = E(S, \bar{S})$. The *global* min-cut is therefore the minimum one over all C_t. The famous max-flow min-cut [4] theorem guarantees the correctness of the algorithm. The running time depends on the algorithm to compute max-flow between s and t. With the fastest algorithm so far [5], the total cost is $O(n^2 m)$ where $n = |V|$ and $m = |E|$.

Karger [1] presented a very simple randomized algorithm for the problem. The main operation used in the algorithm is the *contraction* of edges. Given a graph $G = (V, E)$ and an edge $e = \{u, v\} \in E$, the contraction of e merges u and v and removes all edges between u and v. The operation is illustrated in the Figure 1

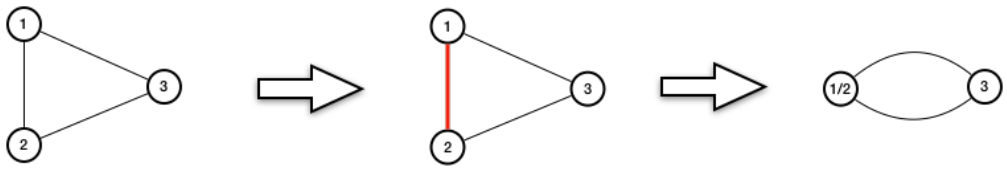

FIGURE 1. Contract an edge between vertex 1 and vertex 2.

Note that the contraction operation may produce multiple edges between two vertices. Karger's algorithm is described in Algorithm 1.

Algorithm 1 Karger's Min-Cut Algorithm

Input: *An undirected graph* $G = (V, E)$.
Output: *The minimum cut of* G.
while G contains more that two vertices **do**
    Choose an edge $e \in E(G)$ uniformly at random;
    Contract e in G;
end while
Assuming $V = \{u, v\}$, $S \leftarrow$ vertices merged to u;
return $E(S, \bar{S})$;

We say a cut C survived after the algorithm if none of the edges in C has been contracted during the execution of the algorithm. We will show that any minimum cut of G will survive with a reasonable probability after one pass of the algorithm. The reason is that in each step of the algorithm, the contracted edge is picked uniformly at random, so the chance to hit the *min*-cut is small.

Let us fix a min-cut C with $|C| = k$ and assume the removal of C separates $S \subseteq V$ and $\bar{S} = V \backslash S$. Therefore C survives if and only if all contractions happen within $G[S]$ or $G[\bar{S}]$[1]. Since in each iteration, the algorithm merged two vertices and terminates when only two vertices remained, there are $n-2$ iterations in total. For every $i = 1, \ldots, n-2$, let A_i be the event that "i-th contraction avoids C". We consider the probability

$$\Pr\left[\bigcap_{i=1}^{n-2} A_i\right] = \prod_{i=1}^{n-2} \Pr\left[A_i \,\middle|\, \bigcap_{j=1}^{i-1} A_j\right]. \tag{1}$$

Note that in the i-th iteration, the graph contains $n - i + 1$ vertices and conditional on that C has never been hit before, each vertex is of degree at least k (otherwise, the minimum cut is less than k). Therefore, the number of edges in i-th iteration is at least $\frac{k \cdot (n-i+1)}{2}$. This implies

$$\Pr\left[A_i \,\middle|\, \bigcap_{j=1}^{i-1} A_j\right] \geq 1 - \frac{2k}{k \cdot (n-i+1)} = \frac{n-i-1}{n-i+1}.$$

Combining with Equation (1) yields

$$\Pr\left[\bigcap_{i=1}^{n-2} A_i\right] \geq \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \frac{2}{n(n-1)}.$$

So if we repeat the algorithm $50n^2$ times, the minimum cut survives with probability at least

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^{50n^2} \geq 1 - e^{-100}, \tag{2}$$

which is quite close to 1.

What is the time cost of this randomized algorithm? If we maintain the adjacency matrix of the graph, each contraction requires $O(n)$ operations. Therefore, to get the probability bound in Equation (2), one needs to cost $O(n^4)$ time.

[1]For any $S \subseteq V$, we use $G[S]$ to denote the subgraph of G induced by S.

The $O(n^4)$ is already comparable to the maxflow based algorithm with the most sophisticated flow algorithm in dense graphs! Moreover, Karger's algorithm is much simpler and easier to implement. In fact, Karger and Stein [2] improved the algorithm to $\tilde{O}(n^2)$[2]. I have left the development of the improvement as an exercise.

References

[1] D. R. Karger, *Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm.*, in SODA, vol. 93, 1993, pp. 21–30. 2

[2] D. R. Karger and C. Stein, *A new approach to the minimum cut problem*, Journal of the ACM (JACM), 43 (1996), pp. 601–640. 4

[3] Wikipedia contributors, *Fast fourier transform — Wikipedia, the free encyclopedia.* https://en.wikipedia.org/w/index.php?title=Fast_Fourier_transform, 2020. 1

[4] ———, *Max-flow min-cut theorem — Wikipedia, the free encyclopedia.* https://en.wikipedia.org/w/index.php?title=Max-flow_min-cut_theorem, 2020. 2

[5] ———, *Maximum flow problem — Wikipedia, the free encyclopedia.* https://en.wikipedia.org/w/index.php?title=Maximum_flow_problem, 2020. 2

[2] The notation $\tilde{O}(T)$ means $O(T \cdot \text{polylog}(T))$.