Randomized Algorithms
Solution for Tutorial Sheet 1

A randomized algorithm for deciding whether two positive integers, represented succinctly by arithmetic circuits, are equal or not.

Recall, we are asked to devise a randomized algorithm that, given as input two arithmetic circuits C_1 and C_2, of sizes n_1 and n_2, has the following properties:

(a.) The algorithm runs in time polynomial in the input size $n = n_1 + n_2$.

(b.) If $\texttt{val}(C_1) = \texttt{val}(C_2)$, then the algorithm always returns "YES".

(c.) If $\texttt{val}(C_1) \neq \texttt{val}(C_2)$, then the algorithm returns "NO" with probability at least $1/2$. (Actually, even better, with probability at least $(1 - \frac{1}{2^n})$.)

We need to describe our algorithm and prove it has the above properties. We are allowed to use the number-theoretic facts stated in the tutorial sheet without proof.

The algorithm we will devise is very simple.

This algorithm and the proof that it works are due to Arnold Schönhage ([4], 1979).

For any positive integer m, let $[m] = \{1, \ldots, m\}$.

We are given circuit C_1 consisting of gates $g_0, \ldots, g_{n_1}$, and circuit C_2 consisting of gates $g'_0, \ldots, g'_{n_2}$, and hence where C_1 and C_2 have sizes n_1 and n_2, respectively. Let $n = n_1 + n_2$. We can assume without loss of generality that $n = n_1 + n_2 \geq 7$, because otherwise, if $n \leq 6$, we can easily (deterministically) compute the values $\texttt{val}(C_1)$ and $\texttt{val}(C_2)$ by evaluating both circuits in constant time, and then check whether or not $\texttt{val}(C_1)$ is equal to $\texttt{val}(C_2)$.

Our randomized algorithm, parametrized by a positive integer k (which is the number of repetitions we will use to amplify the success probability to a "sufficiently high" level, to be determined later), is as follows:

Algorithm EQUAL-INTEGERS(C_1, C_2, k)

1. **for** $j = 1, \ldots, k$ **do**
2. Generate a fresh random number $N_j \in [2^{2n}] = \{1, 2, \ldots, 2^{2n}\}$, uniformly at random;
3. Calculate both $\texttt{val}(C_1) \pmod{N_j}$ and $\texttt{val}(C_2) \pmod{N_j}$;
4. **if** $\texttt{val}(C_1) \pmod{N_j} \neq \texttt{val}(C_2) \pmod{N_j}$
5. **return** "NO";
6. **return** "YES";

Let us now explain how and why this algorithm works:

Step 2: Choosing a number, $N \in [2^{2n}]$, uniformly at random can be done easily, by flipping a fair coin $2n$ times, to generate $2n$ independent random "bits" of the binary representation of a random number $B \in \{0, \ldots, 2^{2n} - 1\}$. We then add 1 to B, to obtain $N = B + 1 \in \{1, \ldots, 2^{2n}\}$. Clearly, each of the 2^{2n} possible distinct values in $[2^{2n}] = \{1, \ldots, 2^{2n}\}$ has equal probability $1/2^{2n}$ of being generated.

Step 3: To evaluate $\texttt{val}(C_1) \pmod N$ and $\texttt{val}(C_2) \pmod N$, we can inductively evaluate each gate of C_1 and C_2, starting from the trivial base case of gates g_0 and g_0', where $\texttt{val}(g_0) \pmod N = \texttt{val}(g_0') \pmod N = 1 \pmod N$. Inductively, we calculate $\texttt{val}(g_i) \pmod N$, for $i = 1, \ldots, n_1$, and $\texttt{val}(g_i') \pmod N$, for $i = 1, \ldots, n_2$, using modular arithmetic, and the fact (described on the tutorial sheet), that the congruence relation $\equiv \pmod N$ is "compatible" with integer addition and multiplication. So, for example, if $g_i := g_{j_i} * g_{k_i}$, then

$$\texttt{val}(g_i) \pmod N \;=\; ((\texttt{val}(g_{j_i}) \pmod N) * (\texttt{val}(g_{k_i}) \pmod N)) \pmod N.$$

Here, the value on both sides of the equality should be understood as the unique value in $\{0, \ldots, N-1\}$ congruent modulo N to that value. Likewise if $g_i := g_{j_i} + g_{k_i}$.

Note that since the number of bits required to represent any of the intermediate calculated values $\texttt{val}(g_i) \pmod N$ is at most $2n$ bits, and since integer $\{+, *\}$ arithmetic instructions on $2n$ bit integers can easily be carried out in time polynomial in n, the entire computation of $\texttt{val}(C_1) \pmod N$ and $\texttt{val}(C_2) \pmod N$ can be carried out in time polynomial in the size n of the input.

Total Running time: Since each iteration of the for loop can be carried out in time $q(n)$, for some polynomial $q(\cdot)$, the entire algorithm can be carried out in time $k \cdot q(n)$. We will later choose k itself to be a suitable polynomial in n, which will ensure that our success probability is "high enough". Hence, under such a choice of k, we will have established property (a.) holds for the algorithm.

We next have to show the correctness of the algorithm, by showing that properties (b.) and (c.) hold.

Property (b.): If $\texttt{val}(C_1) = \texttt{val}(C_2)$, then clearly for any positive integer N, $\texttt{val}(C_1) \pmod N = \texttt{val}(C_2) \pmod N$, and hence the algorithm will always return "YES".

Property (c.): This is the heart of the proof of correctness for this algorithm. We will exploit the (number-theoretic) facts described on the tutorial sheet.

The following lemma is the key:

Lemma 1 *Suppose* $\texttt{val}(C_1) \neq \texttt{val}(C_2)$. *Then*

$$|\{N \in [2^{2n}] \;\mid\; \texttt{val}(C_1) \pmod N \neq \texttt{val}(C_2) \pmod N\}| \geq \frac{2^{2n}}{12 \cdot n}.$$

Proof.

Note firstly that $\texttt{val}(C_1) \pmod N = \texttt{val}(C_2) \pmod N$, or equivalently $\texttt{val}(C_1) \equiv \texttt{val}(C_2) \pmod N$, holds if and only if $N|(|\texttt{val}(C_1) - \texttt{val}(C_2)|)$.

Now note that since $\texttt{size}(C_1) \leq n$ and $\texttt{size}(C_2) \leq n$, using Fact (4.) stated on the tutorial sheet we know that $1 \leq \texttt{val}(C_1) \leq 2^{2^n}$ and $1 \leq \texttt{val}(C_2) \leq 2^{2^n}$. Hence $|\texttt{val}(C_1) - \texttt{val}(C_2)| \leq 2^{2^n}$.

Next, using fact (3.) stated on the tutorial sheet, and the fact that $|\texttt{val}(C_1) - \texttt{val}(C_2)| \leq 2^{2^n}$, we know that there are at most 2^n distinct prime numbers that divide $|\texttt{val}(C_1) - \texttt{val}(C_2)|$.

Hence, the number of prime numbers in $[2^{2n}]$ that do NOT divide $|\texttt{val}(C_1) - \texttt{val}(C_2)|$ is at least:

$$\begin{aligned}
\pi(2^{2n}) - 2^n &\geq \frac{2^{2n}}{3 \cdot \ln(2^{2n})} - 2^n \quad \text{(using Fact (2.) on the tutorial sheet)}, \\
&\geq \frac{2^{2n}}{3 \cdot 2 \cdot n} - 2^n \quad \text{(because } 2n = \log_2(2^{2n}) \geq \ln(2^{2n})) \\
&= 2^{2n}(\frac{1}{3 \cdot 2 \cdot n} - \frac{1}{2^n}) \\
&\geq 2^{2n}(\frac{1}{2 \cdot 3 \cdot 2 \cdot n}) \quad \text{(because we assumed, w.l.o.g., } n \geq 7, \text{ and hence } 2^n \geq (2 \cdot 3 \cdot 2 \cdot n)) \\
&= \frac{2^{2n}}{12 \cdot n}.
\end{aligned}$$

□

It follows immediately from Lemma 1 that if $\texttt{val}(C_1) \neq \texttt{val}(C_2)$, then the probability that $\texttt{val}(C_1) \pmod N \neq \texttt{val}(C_2) \pmod N$ for a number $N \in [2^{2n}]$ chosen uniformly at random is at least $\frac{1}{12n}$. Hence the probability that $\texttt{val}(C_1) \pmod N = \texttt{val}(C_2) \pmod N$ is at most $(1 - \frac{1}{12n})$.

Now, to suitably amplify the success probability, let us fix the number of iterations of the for-loop in our algorithm to be $k := 12n^2$.

Now, assuming that $\texttt{val}(C_1) \neq \texttt{val}(C_2)$, let A_j denote the event that the $j'th$ iteration of the for loop fails, meaning that $\texttt{val}(C_1) \pmod{N_j} = \texttt{val}(C_2) \pmod{N_j}$.

Then the probability that the algorithm fails, meaning the probability that all k inde-

pendent iterations fail, is

$$\begin{aligned}
\Pr(\bigcap_{j=1}^{k} A_j) &= \prod_{j=1}^{k} \Pr(A_j) \quad \text{(by mutual independence of events } A_j \text{, for } j = 1, \ldots, k) \\
&\leq (1 - \frac{1}{12n})^k = (1 - \frac{1}{12n})^{12n^2} \quad \text{(because } \Pr(A_j) \leq (1 - \tfrac{1}{12n}) \text{, for all } j \in [k]) \\
&= ((1 - \frac{1}{12n})^{12n})^n \\
&\leq (\frac{1}{e})^n \quad \text{(because by fact (5.) on the Tutorial sheet, } \tfrac{1}{e} \geq (1 - \tfrac{1}{m})^m \text{ for all } m \geq 1)
\end{aligned}$$

Hence, the probability that the algorithm succeeds is at least $(1 - \frac{1}{e^n}) > (1 - \frac{1}{2^n})$.

This completes our proof that the algorithm satisfies property (c.), in the strong form.

Hence, we have established correctness of the algorithm, and furthermore since the number of iterations of the algorithm is $k := 12n^2$, note that it follows that property (a.) also holds, meaning the algorithm always runs in time polynomial (namely, $12n^2 \cdot q(n)$) in the size n of the input.

Food for thought question: can you devise a *deterministic* polynomial time algorithm for the same decision problem?

As mentioned in the hint on the tutorial sheet, you would immediately become famous if you could do so. This is because, as shown by Allender et.al. in [1], the problem of deciding whether two such arithmetic circuits describe equal numbers, known as `EquSLP`, is in fact equivalent to a famous open problem, namely whether the very general form of the *Polynomial Identity Testing* (`PIT`) problem, known as `ACIT` is decidable in deterministic polynomial time.

`ACIT` and hence `EquSLP` are pivotal problems for understanding the relative power of randomized polynomial time versus determinstic polynomial time, and have been studied for decades because of this. The `ACIT` problem is known to be decidable in **co-RP**, meaning that there is a randomized polynomial time monte carlo algorithm with one-sided error for the problem (where, if two given succinctly represented polynomials are equal, the answer the algorithms gives is always the correct answer "Yes", whereas if the polynomials are not equal, then the answer has small positive probability of being incorrect).

However, even deciding whether the `ACIT` or `EquSLP` problem is in **NP** is open. Moreover, there is a result by Kabanets and Impagliazzo ([3]) which means, roughly speaking, that an efficient deterministic algorithm for `ACIT`, or even just showing that `ACIT` is in **NP** (!!), would resolve some very long-standing open problems about "lower bounds" either for boolean or arithmetic circuits, and would hence constitute a major breakthrough in complexity theory. (If you are very interested to learn more about complexity-theoretic aspects of these problems, you may also wish to read Chapter 6 and Chapter 20 (particularly section 20.4) of the textbook [2] by Arora and Barak on Computational Complexity. This book is available digitally for free through the University Library.)

Despite these very serious difficulties, it is also true that many complexity theory researchers (myself included) believe that it will eventually (perhaps many years from now) be shown that `ACIT` (and hence equivalently `EquSLP`) does have a deterministic polynomial time algorithm. So, you still have a wide open opportunity to become famous!

Kousha Etessami

References

[1] E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the complexity of numerical analysis. *SIAM J. Comput.*, 38(5):1987–2006, 2009.

[2] S. Arora and B. Barak. *Computational Complexity: a Modern Approach.*, Cambridge University Press, 2009.

[3] V. Kabanets and R. Impagliazzo. "Derandomizing polynomial identity testing means proving lower bounds", in *Proceedings of the 35th ACM Symposium on Theory of Computing*, pp. 355-364, 2003.

[4] A. Schönhage. "On the power of random access machines", in *Proceedings of the 6th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 520-529, 1979.