CS 294-2 Simon's Algorithm 9/28/04
Fall 2004 Lecture 7

0.1 Overviews

Recall that our basic primitive for designing quantum algorithms is fourier sampling: prepare some quantum state $\left|\psi\right\rangle = \sum_x \alpha_x \left|x\right\rangle$ on n qubits; perform a Hadamard transform, resulting in the superposition $\sum_x \beta_x \left|x\right\rangle$; now measure to sample x with probability $|\beta_x|^2$. The point is that classically it is difficult to simulate the effects of the quantum interference, and therefore to determine for which strings x there is constructive interference and are therefore output with high probability.

How do we set up the initial superposition $\left|\psi\right\rangle = \sum_x \alpha_x \left|x\right\rangle$? So far we have done so classically: for classically computable functions f and g, we can set up the superposition $\sum_x (-1)^{f(x)} g(x \cdot 0^m)$. A major innovation in Simon's algorithm is the use of quantum techniques to set up the initial superposition.

Suppose we're given a function $f : \{0,1\}^n \to \{0,1\}^n$, specified by a black box. (Note that the outputs of f are n-bit strings, rather than single bits.) We're promised the following about f: there exists a nonzero secret string $a \in \{0,1\}^n$ such that

- For all inputs $x \in \{0,1\}^n$, $f(x) = f(x \oplus a)$.
- For all inputs $x, y \in \{0,1\}^n$, if $x \neq y \oplus a$, then $f(x) \neq f(y)$.

What these conditions mean is that f is a 2-to-1 function, and that any two inputs mapping to the same output differ in exactly those positions i for which $a_i = 1$, where i is the i-th position in an n-bit string. For example, $f(x) = 2 * \lfloor x/2 \rfloor$. For any k, $f(2k) = f(2k+1)$. But for any i, j, if i and j are not like $(2k, 2k+1)$ pair, then $f(i) \neq f(j)$. In this example, a is 1.

0.2 Simon's Algorithm

Let $x \oplus y$ denote the bitwise mod 2 addition of x and y, and $x \cdot y$ denote the inner product of x and y, $\sum_{i=1}^n x_i y_i$ mod 2. We now present Simon's quantum algorithm for finding a. The algorithm uses two registers, both with n qubits. The registers are initialized to the basis state $|0 \cdots 0\rangle |0 \cdots 0\rangle$. We then perform the

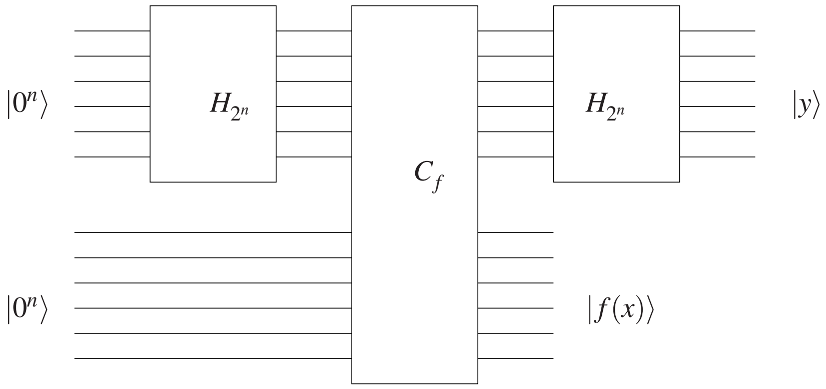

Figure 1: Simon's algorithm

Hadamard transform H_{2^n} on the first register, producing the superposition

$$\frac{1}{2^{n/2}} \sum_{x \in \{0,1\}^n} |x\rangle\, |0 \cdots 0\rangle\,.$$

Then, we compute $f(x)$ through the oracle C_f and store the result in the second register, obtaining the state

$$\frac{1}{2^{n/2}} \sum_{x \in \{0,1\}^n} |x\rangle\, |f(x)\rangle\,.$$

The second register is not modified after this step. Thus we may invoke the principle of safe storage and assume that the second register is measured at this point.

Let $f(z)$ be the result of measuring of the second register. There are exactly two x such that f(x)=f(z), according to the definition of f. one is z and the other is $z \oplus a$. The quantum state after measuring is

$$\left(\frac{1}{\sqrt{2}} |z\rangle + \frac{1}{\sqrt{2}} |z \oplus a\rangle \right) |f(z)\rangle$$

We're now done with the second register, so in the discussion to follow, we'll omit it from our notation. The state in the n-qubit first register

$$\frac{1}{\sqrt{2}} |z\rangle + \frac{1}{\sqrt{2}} |z \oplus a\rangle$$

clearly contains some information about a—the question is how to extract it. If we observed at this point, we will get z or $z \oplus a$. It is a state chosen uniformly at random from $\{0,1\}^n$, containing no information at all. Therefore some more computation is required. The key, once again, is to apply the Hadamard transform H_{2^n} to the register. Doing so, we obtain a superposition

$$\sum_{y \in \{0,1\}^n} \alpha_y |y\rangle$$

where

$$\alpha_y = \frac{1}{\sqrt{2}} \frac{1}{2^{n/2}} (-1)^{y \cdot z} + \frac{1}{\sqrt{2}} \frac{1}{2^{n/2}} (-1)^{y \cdot (z \oplus a)} = \frac{1}{2^{(n+1)/2}} (-1)^{y \cdot z} \left[1 + (-1)^{y \cdot a}\right].$$

There are now two cases. For each y, if $y \cdot a = 1$, then $\alpha_y = 0$, whereas if $y \cdot a = 0$, then

$$\alpha_y = \frac{\pm 1}{2^{(n-1)/2}}.$$

So when we observe the first register, with certainty we'll see a y such that $y \cdot a = 0$. Hence, the output of the measurement is a random y such that $y \cdot a = 0$. Furthermore, each y such that $y \cdot a = 0$ has an equal probability of occurring. Therefore what we've managed to learn is an equation

$$y_1 a_1 \oplus \cdots \oplus y_n a_n = 0 \tag{1}$$

where $y = (y_1, \ldots, y_n)$ is chosen uniformly at random from $\{0,1\}^n$. Now, that isn't enough information to determine a, but assuming that $y \neq 0$, it reduces the number of possibilities for a by half.

It should now be clear how to proceed. We run the algorithm over and over, accumulating more and more equations of the form in (1). Then, once we have enough of these equations, we solve them using Gaussian elimination to obtain a unique value of a. But how many equations is enough? From linear algebra, we know that a is uniquely determined once we have $n-1$ linearly independent equations—in other words, $n-1$ equations

$$\begin{aligned} y^{(1)} \cdot a &\equiv 0 \pmod 2 \\ &\vdots \\ y^{(n-1)} \cdot a &\equiv 0 \pmod 2 \end{aligned}$$

such that the set $\left\{y^{(1)}, \ldots, y^{(n-1)}\right\}$ is linearly independent in the vector space Z_2^n. Thus, our strategy will be to lower-bound the probability that any $n-1$ equations returned by the algorithm are independent.

Suppose we already have k linearly independent equations, with associated vectors $y^{(1)}, \ldots, y^{(k)}$. The vectors then span a subspace $S \subseteq Z_2^n$ of size 2^k, consisting of all vectors of the form

$$b_1 y^{(1)} + \cdots + b_k y^{(k)}$$

with $b_1, \ldots, b_k \in \{0,1\}$. Now suppose we learn a new equation with associated vector $y^{(k+1)}$. This equation will be independent of all the previous equations provided that $y^{(k+1)}$ lies *outside* of S, which in turn has probability at least $(2^n - 2^k)/2^n = 1 - 2^{k-n}$ of occurring. So the probability that any n equations are independent is exactly the product of those probabilities.

$$\left(1 - \frac{1}{2^n}\right) \times \left(1 - \frac{1}{2^{n-1}}\right) \times \cdots \times \left(1 - \frac{1}{4}\right) \times \left(1 - \frac{1}{2}\right).$$

Can we lower-bound this expression? Trivially, it's at least

$$\prod_{k=1}^{\infty} \left(1 - \frac{1}{2^k}\right) \approx 0.28879;$$

the infinite product here is related to something in analysis called a q-series. Another way to look at the constant $0.28879\ldots$ is this: it is the limit, as n goes to infinity, of the probability that an $n \times n$ random matrix over Z_2 is invertible.

But we don't need heavy-duty analysis to show that the product has a constant lower bound. We use the inequality $(1-a)(1-b) = 1 - a - b + ab > 1 - (a+b)$, if $a, b \in (0,1)$. We just need to multiply the product out, ignore monomials involving two or more $\frac{1}{2^k}$ terms multiplied together (which only increase the product), and observe that the product is lower-bounded by

$$\left[1 - \left(\frac{1}{2^n} + \frac{1}{2^{n-1}} + \cdots + \frac{1}{4}\right)\right] \cdot \frac{1}{2} \geq \frac{1}{4}.$$

We conclude that we can determine a with constant probability of error after repeating the algorithm $O(n)$ times. So the number of queries to f used by Simon's algorithm is $O(n)$. The number of computation steps, though, is at least the number of steps needed to solve a system of linear equations, and the best known upper bound for this is $O\left(n^{2.376}\right)$, due to Coppersmith and Winograd.

0.3 Classical solution

We are going to prove that any probabilistic algorithm needs an exponential time to solve this problem. Suppose that a is chosen uniformly at random from $\{0,1\}^n - \{0^n\}$. Now consider a classical probabilistic algorithm that's already made k queries, to inputs $x_1, \ldots, x_k$. We want to know how much information the algorithm could have obtained about a, given those queried pairs $(x_i, f(x_i))$.

On the one hand, there might be a pair of inputs x_i, x_j (with $1 \leq i, j \leq k$) such that $f(x_i) = f\left(x_j\right)$. In this case, the algorithm already has enough information to determine a: $a = x_i \oplus x_j$.

On the other hand, suppose no such pair $f(x_i), f(x_j)$ exists. Then the queried $f(x_i)$'s are distinct and a is none of $\binom{k}{2}$ values $x_i \oplus x_j$.

The probability that the next query will succeed is at most

$$\frac{k}{2^n - 1 - \binom{k}{2}}$$

because there are at least $2^n - 1 - \binom{k}{2}$ possible values of u for choosing at the $(k+1)$-th query. And $f(x_{k+1})$ should be equal to one of the prior observed $f(x_i)$, $i \in [1,k]$.

Taking the sum over all $k \in \{1, \ldots, m\}$. We get

$$\sum_{k=1}^{m} \frac{k}{2^n - 1 - \binom{k}{2}} \leq \sum_{k=1}^{m} \frac{k}{2^n - k^2} \leq \frac{m^2}{2^n - m^2}$$

In order to have an constant probability, we must choose $m = \Omega(2^{n/2})$. Hence, any deterministic algorithm has to run in exponential time to get a correct answer with probability larger than a constant.
