QuIC Seminar 11

Simon's Algorithm

Contents

While Bernstein-Vazirani gave us a linear speedup in the query model, Simon's algorithm will give us a more impressive speedup: exponential. Indeed, Simon's algorithm (1994) was the first quantum algorithm with such an exponential speedup. The story goes that Simon was unimpressed by recent papers (like Deustch-Jozsa and Bernstein-Vazirani) and set out to prove there couldn't be any *real* quantum speedup.

11.1 Simon's Problem

The problem Simon considered[1] was the following.

Definition 11.1 (Simon's Problem). Let f be a function from n-bit strings to n-bit strings

$$f : \{0,1\}^n \longrightarrow \{0,1\}^n \tag{11.1}$$

such that, for all $x, y \in \{0,1\}^n$

$$f(x) = f(y) \iff x = y \oplus s. \tag{11.2}$$

Here, $s \in \{0,1\}^n$ is a "secret string" and $\oplus$ denotes bitwise addition module two. **Simon's problem** is to find s by querying f (as few times as possible).

The bitwise addition module two (also bitwise XOR) is best understood by vector addition. The following example should remove any ambiguity.

Example 1: Bitwise XOR

Consider $s_1, s_2, s_3 \in \{0,1\}^3$ defined by

$$s_1 := 010 \qquad s_2 := 110 \qquad s_2 := 001.$$

Then,

$$s_1 \oplus s_2 = 100, \qquad s_1 \oplus s_3 = 011, \qquad s_2 \oplus s_3 = 111.$$

[1] Or rather *found*. These quantum algorithms are always discovered by seeing what the quantum computer can do well, then framing the problem around that.

Here, we take the first bits of each string, add them together, and take the result modulo two. We continue in this fashion for all bits in the bit strings.

One may see what's going on even clearer by representing the bit strings as vectors in $\mathbb{Z}_2$. Then, for example

$$\begin{aligned} s_1 \oplus s_2 &= (0,1,0) \oplus (1,1,0) \\ &= (0+1 \mod 2, 1+1 \mod 2, 0+0 \mod 2) \\ &= (1,0,0). \end{aligned}$$

One property that should be clear from the vector representation is the following distribution law:

$$(z_1 \oplus z_2) \cdot z_3 = z_1 \cdot z_2 \oplus z_1 \cdot z_3 \tag{11.3}$$

Here, each z_i is a length n bit string and the $\cdot$ denotes the inner product (modulo two). We'll use this property during the analysis of Simon's algorithm.

Exercise 52: Prove (11.3) by direct computation.

First, we consider examples of functions f with the "Simon property" (11.2).

Example 2: Functions with Simon's Property

Consider the function $f : \{0,1\}^3$ with the secret string $s = 110$ defined by the following table of values.

x	f(x)
000	101
001	010
010	000
011	110
100	000
101	110
110	101
111	010

It's not hard to see that the property (11.2) holds. For example, we have $f(000) = 101 = f(110)$. By taking the input of the first (000) and applying the XOR mask s, we indeed get the second input 110, since $000 \oplus s = 000 \oplus 110 = 110$. Thus, Simon's property hold's here. Note that Simon's property is symmetric—i.e., we could have applied the XOR mask s to the second bit string to get the first. Indeed, $110 \oplus s = 110 \oplus 110 = 000$.

Exercise 53: Find the other three pairs of "matching inputs" and verify that Simon's property holds for those inputs as well.

Thus, functions f with Simon's property are two-to-one functions,[2]meaning that there are exactly two inputs which produce the same output. What are these two inputs? Simon's property gives no restrictions on what they can be, only how they are related. Namely, by flipping the bits of one input at all locations where the secret string s has a one, we obtain the other input.

11.2 Classical Query Complexity

In the classical setting, we send a bit string x_1 to our query as input and learn the output $f(x_1)$. We then do this for another input $x_2 \mapsto f(x_2)$. Before we ask how many queries it takes us to solve Simon's problem (i.e., compute s), let's first ask the simpler question: how do we compute s in the first place?

[2]Unless s is the all zero bit string, in which case f is a one-to-one function. For simplicity, we'll exclude this case in our analysis and assume the more general case of $s \neq \mathbf{0}$.

Suppose you send an input x_i into the query and get out $f(x_i)$. After some more inputs, you send another input x_j to learn $f(x_j)$. By keeping record of the function values, you remember that $f(x_i) = f(x_j)$. Can you determine s from this? If so, how?

Example 3: Computing s from collisions.
Consider the function f given in Example 11.1. Suppose you first send in the input $x_1 = 010$ and then the input $x_2 = 100$. As can be seen in the table, both inputs produce the same output. Verify that the secret string s satisfies $s = x_1 \oplus x_2$. Do this for all other "matching inputs" (i.e., pairs of inputs that produce the same output).

It's not hard to see that this is always the case—i.e., we can always find s by finding a "collision pair" or "matching pair" of inputs x and y.

Exercise 54: Prove that $z \oplus z = \mathbf{0}$ (the all zero bit string) for all $z \in \{0,1\}^n$. Use this fact to prove that

$$s = x \oplus y \tag{11.4}$$

for x, y, and s in (11.2).

We now know that we can compute s, thereby solving Simon's problem, by finding a collision pair. The question now is:

How many queries does it take to find a collision pair?

One thing we could do is just query *every value* of the function f. Then we could look through and surely find a matching pair![3]How many queries does this take? Well, f is a function on bit strings of length n, and there are 2^n total such bit strings. Thus, we can solve Simon's problem in *at most* 2^n queries. Can we do any better than this?

The answer is yes, but not significantly better. I claim we can do quadratically better than 2^n queries by the same reasoning (or lack thereof) in the famous *birthday paradox*.

Example 4: The birthday paradox
Suppose you're in a room with 23 people. What's the probability that *any pair of people* share a birthday? Are you surprised to hear its about 50%?

One may naively expect we'd need on the order of 365 people to get a match, since there are 365 days in the year. The key here is that we care about *any pair*. Why is this important?

Suppose we have n people. How many pairs can we form? There are n choices for the first person, then $n-1$ choices for the second person (since we've already chosen one). Of course "Alice and Bob" is the same pair as "Bob and Alice" as far as we're concerned, so we'd better divide the total number by two to account for this. Thus, the total number of pairs of n people is

$$\frac{n(n-1)}{2}. \tag{11.5}$$

Combinatorically, we could have wrote down the answer immediately as

$$\binom{n}{2} = \frac{n(n-1)}{2}. \tag{11.6}$$

So, we've learned that the *number of pairs* doesn't grow linearly in the number of people. It actually grows quadratically, to leading order:

$$\frac{n(n-1)}{2} = O(n^2). \tag{11.7}$$

For the birthday paradox, the key realization is that once the number of pairs is on the order of 365, we have a good probability of finding a collision pair.

[3]Recall we're excluding the case that s is the all zero bit string, for simplicity. Though, the analysis really isn't much different here. If we queried everything and didn't find a match, we could be pretty sure (i.e., positive) that $s = \mathbf{0}$.

The same is true for Simon's algorithm. Suppose we send in t input values to our query at random. After t queries, we have roughly t^2 pairs. We can expect to find a matching pair once $t^2 = O(2^n)$, i.e., $t = O(2^{n/2})$.

Since we want to prove a quantum speedup, we have to consider the best case classical query complexity. It's possible to rigorously show that the best query complexity of a randomized classical algorithm cannot be better than $O(2^{n/2})$. We'll present a "semi-rigorous"[4] argument as to why this is the case.

Theorem 11.1. The best query complexity for a randomized classical algorithm solving Simon's problem is at least $O(2^{n/2})$.

After we've made one query on the input x, the probability of getting a collision after one more input y is

$$P(x = y) = \frac{1}{2^n - 1}. \tag{11.8}$$

Why is this? There are 2^n total outcomes and we've already queried one (hence the minus one). Now, the probability of a collision after t queries is[5]

$$P(\text{collision after } t \text{ queries}) \leq \frac{\binom{t}{2}}{2^n - 1}. \tag{11.9}$$

Thus, if we want this probability to be "good" (say 1/2), then it's easy to see we must take $t = O(2^{n/2})$.

11.3 Quantum Query Complexity

Now's the fun (quantum!) part: I claim that there's a quantum algorithm that solves Simon's problem with $O(n)$ queries, an exponential improvement! What's the proof of this? Simon's algorithm!

11.3.1 Simon's Algorithm

Simon's algorithm, or at least the quantum part, is shown in the circuit diagram below.

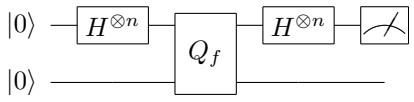

Figure 11.1: Quantum circuit implementing the quantum part of Simon's algorithm.

The circuit uses $2n$ qubits arranged in two registers of n qubits each. Note that we're using a single line in the top register to represent n qubits, and similarly in the bottom register.

Exercise 55: Write out the circuit in Fig. 11.1 to explicitly show all $2n$ qubits and wires.

This circuit represents the "quantum part" of Simon's algorithm, which consists of the following high-level steps.

Definition 11.2 (Simon's algorithm). Simon's algorithm is a hybrid quantum-classical algorithm that implements the following two steps.

(1) Run the circuit shown in Fig. 11.1 $m = O(n)$ times.

[4] cf. *semiclassical*.

[5] Here we're using the *union bound*, a fundamental postulate of probability theory, which states that the probability of a union of events is at most the probability of the sum of events. Mathematically,

$$P\left(\bigcup_i E_i\right) \leq \sum_i P(E_i).$$

The expression is an equality for mutually exclusive events.

(2) Perform classical postprocessing on the output that scales efficiently in n to determine s.

Let's break down these steps in more detail, first examining what Simon's circuit does for us. We start out with two registers in the zero state and perform Hadamard gates on all qubits in the top register:

$$|0\rangle^{\otimes n} \otimes |0\rangle^{\otimes n} \longmapsto |+\rangle^{\otimes n} \otimes |0\rangle^{\otimes n} = \sum_{x\in\{0,1\}^n} |x\rangle \otimes |0\rangle^{\otimes n} \tag{11.10}$$

(Here, as in other places, we omit normalization factors for simplicity.) We then make a query, using the first type

$$Q_f|x, a\rangle = |x, f(x) \oplus a\rangle. \tag{11.11}$$

Doing so yields the state

$$\sum_{x\in\{0,1\}^n} |x\rangle \otimes |0\rangle^{\otimes n} \longmapsto \sum_{x\in\{0,1\}^n} |x\rangle \otimes |f(x)\rangle. \tag{11.12}$$

Note that although the terms inside the summand are written as tensor products, the resulting state is (most generally) entangled. If this confuses you, consider the following exercise.

Exercise 56: To see that the sum of tensor product states (11.15) can be entangled, show that the state

$$|\psi\rangle = \sum_{j=0}^{1} |j\rangle \otimes |j\rangle \tag{11.13}$$

is entangled. Which entangled state is it?

Note that at this point (11.15) we're done with our bottom register of qubits—no more operations happen on it. This is somewhat non-intuitive—why did we make a query in the first place, if we're not going to do anything with the second register $|f(x)\rangle$. The reason we do this is the *effect* it has on the first register. These registers are entangled, so the simple act of making the query affects our top register in a beneficial (computationally speaking) way. We'll see the particular affect this query has shortly.

Simon's circuit now tells us to Hadamard all qubits in the top register again. To write this down, we'll use the following circuit identity.

Theorem 11.2 (Effect of Hadamard on a computational basis state). Let $|x\rangle = |x_1 \cdots x_n\rangle$ denote a computational basis state on n qubits. That is, $x \in \{0,1\}^n$ and each $x_i \in \{0,1\}$ denotes a particular bit value. Then,

$$H^{\otimes n}|x\rangle = \sum_{z\in\{0,1\}^n} (-1)^{x\cdot z}|z\rangle. \tag{11.14}$$

Here, the dot product $x \cdot z$ denotes the dot product modulo two.

Exercise 57: Prove Theorem 11.2. *Hint:* $H|x_i\rangle = |0\rangle + (-1)^{x_i}|1\rangle$ *for all* $x_i \in \{0,1\}$.

We can now write the resulting state after the second round of Hadamard gates:

$$\sum_{x\in\{0,1\}^n} |x\rangle \otimes |f(x)\rangle \longmapsto \sum_{z\in\{0,1\}^n} \sum_{x\in\{0,1\}^n} (-1)^{x\cdot z}|z\rangle \otimes |f(x)\rangle \tag{11.15}$$

Now, we're going to condition on a particular output of the function—this collapses the sum over x and gives us a particular value of $f(x)$, say $f(x) = w$[6]. Note that, because of Simon's promise, there are two such inputs that produce this output, call them x_1 and x_2.

[6]Formally, we're taking a partial trace over the second (bottom) register of $|f(x)\rangle$.

$$f(x_1) = f(x_2) \equiv w. \tag{11.16}$$

The resulting state after conditioning on this particular outcome is then

$$\sum_{z\in\{0,1\}^n} \sum_{x\in\{0,1\}^n} (-1)^{x\cdot z}|z\rangle \otimes |f(x)\rangle \longmapsto \sum_{z\in\{0,1\}^n} \left[(-1)^{x_1\cdot z} + (-1)^{x_2\cdot z}\right]|z\rangle \otimes |w\rangle \tag{11.17}$$

Now Simon's circuit tells us to measure the top register. We're going to get some output string z, but the question is, which possible z in the above sum could we measure? If we measure it, then the amplitude must have interfered constructively—that is,

$$(-1)^{x_1\cdot z} = (-1)^{x_2\cdot z} \tag{11.18}$$

We can rewrite this condition to say

$$x_1 \cdot z = x_2 \cdot z \pmod 2. \tag{11.19}$$

Now, we use the criteria from Simon's problem, which states that $x_2 = x_1 \oplus s$, where s is the secret string we're trying to find. Substituting this expression in and using (11.3), we arrive at the condition

$$\boxed{x_1 \cdot s = 0.} \tag{11.20}$$

Thus, the result of running Simon's circuit is that we measure a bit string $x_1 \in \{0,1\}^n$ such that it's inner product modulo two with the secret string s is zero. This doesn't tell us s exactly, but it tells us "something about s."

Simon's algorithm then tells us to run this circuit many times, in particular $m = O(n)$ times. What does this give us? Well, it gives us m bit strings $x_1, ..., x_m$ such that

$$x_1 \cdot s = 0$$

$$x_2 \cdot s = 0$$

$$\vdots$$

$$x_m \cdot s = 0$$

This is a system of m equations in n variables, which we can solve efficiently using classical methods, for example Gaussian elimination[7]. This is the second step of Simon's algorithm—the classical post-processing. After we solve this, we have found the secret string s, thereby solving Simon's problem!

Why do we require that we run Simon's circuit $m = O(n)$ times, instead of just exactly n times? There's a small chance that we could measure the same output string, say $x_i = x_j$, at two different runs of the circuit, and we need at least n equations in order to solve the system.

Exercise 58: What's the probability of measuring the same bit string twice in two independent runs of Simon's circuit?

Exercise 59: Prove that there is a high probability of being able to solve the system of equations after $m = O(n)$ independent runs of Simon's circuit.

[7]Gaussian elimination has runtime $O(n^3)$ for an $n \times n$ system of equations.