Lecture 24, Tues April 18: Collision and Other Applications of Grover

We've seen the application of Grover's algorithm to searching game trees. Now let's see another important application, to…

The Collision Problem

In the simplest version of this problem, we're given quantum black-box access to a function

$f:\{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$

where N is even, and we're promised that f is two-to-one. The problem is to find x and y such that $x \neq y$, and $f(x) = f(y)$. So, there are lots of "collisions" to be found, and the challenge is just to find one of them.

In another version of the problem, we're promised that *either* f is one-to-one or it's two-to-one, and the problem is to decide which. Clearly, if we could solve the "search" version, then we could also solve the "decision" version, by simply outputting that f is two-to-one if a collision is found, or that f is probably one-to-one if not. Conversely, any lower bound for the decision version implies the same lower bound for the search version.

The collision problem often arises in cryptography, when we're trying to break collision resistant hash functions. You can think of the collision problem as being a lot like Simon's Problem but with less structure---or alternatively, as being like the Grover search problem but with *more* structure.

With a classical randomized algorithm, if we have black-box access to f, then $\Theta(\sqrt{N})$ queries are necessary and sufficient to solve the collision problem. Why? The upper bound follows from the famous "birthday attack": if there are N days in the year, then you only need to ask about $\sqrt{N}$ people before there's an excellent chance that you'll find two with the same birthday, because what matters is the number of *pairs* of people. The lower bound can be proven using the union bound: with a random two-to-one function, each pair has only a $\sim 1/N$ chance of being a collision, so to find a collision with constant probability you need to look at $\sim \sqrt{N}$ pairs or more.

What about quantumly? Well, we could of course simulate the above randomized algorithm to get $\sim \sqrt{N}$. But there's also a completely different way to get $\sim \sqrt{N}$: namely, we could first query $f(1)$, and then do a Grover search for an $x \neq 1$ such that $f(x) = f(1)$. So a question arises: can we combine the two approaches to do even better than $\sim \sqrt{N}$?

(Brassard, Hoyer, Tapp 1997) showed how to do exactly that. Here's their algorithm:

First, pick $\sqrt[3]{N}$ random inputs to f, query them classically, and sort the results for fast lookup.

Next, run Grover's algorithm on $N^{2/3}$ *more* random inputs to f (inputs that weren't queried in the first step). In this Grover search, count each input x as "marked" if and only if $f(x)=f(y)$ for one of the $\sqrt[3]{N}$ inputs y that was already queried in the first step. (This requires lookups to our sorted list, but no additional queries to f.)

How many pairwise comparisons do we make this way?

$$N^{2/3} \times N^{1/3} = N$$

What's the runtime?

$$N^{1/3} + \sqrt{N^{2/3}} = O(N^{1/3})$$

The centerpiece of Professor Aaronson's PhD thesis was showing that you can't improve on this by much. It was later shown by Yaoyun Shi that you can't improve on it at all.

The BHT algorithm gives a good illustration of how quantum algorithms can end up with weird running times. You have two or more phases of the algorithm that you try to balance against each other, make about equally time-consuming, in order to minimize the total time,.

At a high level, you can see why the BBBV proof that we used to prove the optimality of Grover's algorithm doesn't work for the collision problem. In the BBBV proof, we changed a single element from 0 to 1, then showed that it would take many iterations for the algorithm to notice. With the collision problem, by contrast, the key issue is that turning a one-to-one function into a two-to-one function requires changing half the elements.

Instead, Aaronson and Shi used polynomial approximation theory (a branch of math) to rule out super-fast quantum algorithms for the collision problem.

In some sense, proving a quantum lower bound for the collision problem *should* be harder than proving one for the Grover problem, because if the lower bound for collision did too much, then it would rule out things like Simon's algorithm or Shor's algorithm. What the proof does is take advantage of the symmetry of the collision problem.

Symmetry in the sense that you can arbitrarily permute the function in the collision problem, and it's still a valid input, which isn't the case for something like Simon's problem.

More broadly, after Grover published his algorithm, there was a ten-year flood of people realizing you can use it for speedups in all sorts of problems.

For example, in addition to the collision problem, there's also the closely related problem of …

Element Distinctness

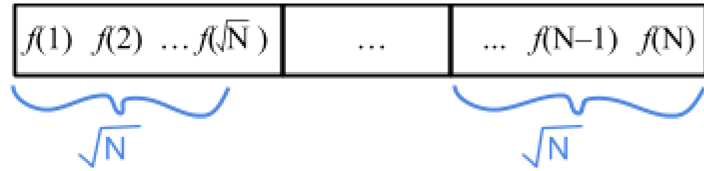
Given black-box access to the function $f: \{1, \dots, N\} \rightarrow \{1, \dots, N\}$, with no promises about f. Determine if f is one-to-one.

In other words: Are there any duplicates/collisions?

Classically, you'd hash the elements (or sort them, or use a binary search tree, etc.), which would take N queries plus the amount of computation required for sorting (say, N log N steps).

Quantumly, there's an algorithm that takes only $O(N^{3/4})$ queries, as shown in a paper of Buhrman et al. from 2000.

This is another cute application of Grover search.



Given a list of the N values of a function, we split them into $\sqrt{N}$ blocks of $\sqrt{N}$ values each. Then, by using Grover's algorithm over these blocks, we produce a quantum algorithm that makes O($\sqrt{N}$) queries to f and finds a collision (if there is one) with probability $\frac{1}{\sqrt{N}}$.

So, how can we do that?

Pick a block at random and query all elements in it.

If you find a collision in the block, you're already done! If you don't, then sort the elements in the block for fast lookup. Next, do a Grover search on the other items in the list, counting an item as "marked" if and only if it equals an element from the collision block.

As long we were lucky enough to pick a block that contains at least one element of a collision pair, this algorithm will find such a pair with constant probability. Hence it succeeds with $\frac{1}{\sqrt{N}}$ probability.

We can improve on this (after all, we *do* have access to a quantum computer), by doing an outer layer of Grover search that searches through the $\sqrt{N}$ blocks, counting a block as "marked" if and only if the "inner" algorithm above finds a collision involving that block.

Our final runtime is $\sqrt{\sqrt{N}}$ * $\sqrt{N}$ = **O($N^{3/4}$)**

Outer Grover   Inner collision search   **Element Distinctness in $N^{3/4}$**

What's the lower bound on Element Distinctness?

As a baseline, we know the query complexity has to be at least that of searching a list for a given i, which is ~ $\sqrt{N}$. Pinning down the complexity of Element Distinctness between ~ $\sqrt{N}$ and ~$N^{3/4}$ was an open problem for several years.

As it turns out, the answer is ~$N^{2/3}$.

Yaoyun Shi noticed that an ~$N^{2/3}$ lower bound follows from the ~$N^{1/3}$ lower bound for the collision problem.

That is, suppose for a contradiction that we could solve Element Distinctness in $t << N^{2/3}$ queries.

This would let us solve the collision problem in $\sqrt{t} << N^{1/3}$ queries.

How?

Given a 2-to-1 function f, pick ~ $\sqrt{N}$ inputs uniformly at random. Since, by the birthday paradox, we can expect to find a collision within that set of inputs (with constant probability), we now simply run our hypothesized Element Distinctness algorithm on that subset.

Matching this lower bound, in 2003 Andris Ambainis found a quantum algorithm that solves Element Distinctness with $O(N^{2/3})$ queries. His algorithm used "quantum walks," which are vaguely like Grover's algorithm but more sophisticated. It also required a huge amount of workspace qubits, on the order of $N^{2/3}$. Whether this large number of workspace qubits is necessary remains open to this day.

OK, how about one more vignette on the quantum query complexity of fundamental problems from computer science, which is now something we know a lot about.

Parity of an *n*-bit String

Given $x \in \{0,1\}^n$, suppose we just want to determine $x_1 \oplus x_2 \oplus \ldots \oplus x_n$ (i.e., whether the total number of 1 bits is odd or even).

Classically, of course, this requires n queries. Quantumly, we've seen that we can do it in $\frac{n}{2}$ queries, by splitting x into $\frac{n}{2}$ pairs and then applying the Deutsch-Jozsa algorithm separately to each pair.

A beautiful result shows that this is optimal: $\frac{n}{2}$ queries are needed by any quantum algorithm for Parity. This can be shown using the polynomial method (Beals et al., 1998). Just to give you a brief taste:

Suppose that we have quantum algorithm A, which makes t queries to an input string x. We can study the probability that A accepts x, call it $p(x)$.

For simplicity we'll assume that A is trying to compute a Boolean function, so it either accepts or rejects.

A critical fact proven by Beals et al. is that $p : \{0,1\}^n \rightarrow \mathbb{R}$ turns out to be a multivariate polynomial in the n bits of x. Furthermore, the degree of that polynomial is at most 2T.

So, suppose we can show that any polynomial p that can approximate a given Boolean function f must have $\deg(p) \geq D$. Then we can deduce that the quantum algorithm must have made at least $\frac{D}{2}$ queries.

This reduces questions about quantum query complexity to purely mathematical questions about the degrees of real polynomials, with no further CS or quantum computing needed!

In the case of Parity, it turns out one can show that any polynomial approximating the Parity function needs degree n. This implies that any quantum algorithm for Parity must make at least $\frac{n}{2}$ queries.

As a complement to Parity, it's also worth briefly discussing the n-bit Majority function, which outputs 0 or 1 depending on whether the input string has more 0's or 1's respectively. The quantum query complexity of Majority turns out to be order n--i.e., there is no asymptotic quantum speedup for this problem--and that can also be proved using the polynomial method.

However, there *is* a quantum speedup for a problem closely related to Majority.

For a poll to be accurate within x percent, how many people do you need to classically query?

Suppose you want to approximate the Hamming weight (i.e., number of 1's) in your n-bit input string, to within an additive error $\pm \varepsilon n$.

Classically, you can do this by sampling $\sim \frac{1}{\varepsilon^2}$ uniformly random bits and taking their average, and this is also tight. (This fact is extremely useful to know when, e.g., choosing the sample size for a political poll, to achieve a desired margin of error like ±3%.)

Quantumly, by contrast, via a clever application of Grover's algorithm, it turns out that we can solve this problem using only $\sim \frac{1}{\varepsilon}$ queries: a quadratic speedup.

To conclude this lecture, let's talk a bit about **Quantum Complexity Theory**, the generalization of computational complexity theory to the quantum realm, so we can understand the broader context of the quantum algorithms we've seen. Classically, we define complexity classes such as:

P (Polynomial-Time), the class of decision problems solvable by a standard, deterministic digital computer in polynomial time.

(examples: linear programming, connectivity of graphs)

NP (Nondeterministic Polynomial-Time), the class of decision problems for which there's a deterministic polynomial-time algorithm to *verify* yes-answers.

(example: factoring, when suitably phrased as a yes-or-no decision problem)

NP-hard problems are, roughly speaking, problems to which every **NP** problem can be reduced in polynomial time. So in particular, if you could solve any **NP**-hard problem in polynomial time, then you'd be able to solve everything in NP in polynomial time.

NP-complete problems are those that are both in **NP** and **NP**-hard. Informally, they're "the hardest problems in **NP**."

(examples: Traveling Salesman, 3SAT, Max Clique, Bin Packing, VLSI layout, Sudoku, Super Mario, and many other problems of practical and not-so-practical importance)

This picture already involves an enormous mathematical unknown: famously, no one has ruled out the possibility that **P** = **NP**, in which case all **NP** problems (so in particular, all **NP**-complete problems) would be solvable in polynomial time.

Where does quantum computing fit in?

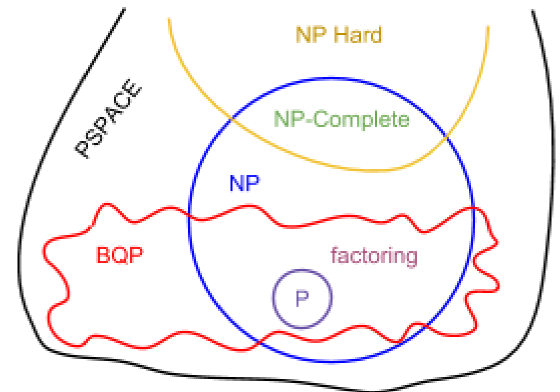
In 1993, Bernstein and Vazirani defined the complexity class **BQP** (Bounded-Error Quantum Polynomial-Time) as a quantum generalization of **P**. Loosely speaking, **BQP** contains all decision problems that can be solved in polynomial time with a quantum computer.

How does **BQP** relate to classical complexity classes?

We know that **P** ⊆ **BQP**, basically because Toffoli gates can simulate AND, OR, and NOT gates, and hence universal classical digital computation--and hence quantum computers can simulate classical ones. We also know, from Shor's algorithm, that Factoring (when suitably phrased as a decision problem) is in **BQP**, though it's not known (to put it mildly) whether Factoring is in **P**.

We don't know whether **NP** ⊆ **BQP**--that is, whether quantum computers can solve all **NP** problems (including **NP**-complete problems) in polynomial time. The BBBV Theorem does tell us that there isn't an easy proof of **NP** ⊆ **BQP** to be had that just treats the **NP** problem as a black box.

"Can quantum computers solve **NP**-complete problems in polynomial time?"
is one of the big open problems of Quantum Complexity Theory.



We could also ask the converse, "Is **BQP** ⊆ **NP**?" In other words: for every problem that a quantum computer can solve, is there a short proof of the answer that's easy to verify classically?

It's possible that there are problems that a quantum computer could solve easily which can't be classically solvable, but we don't have any present examples.

The last important question to ask here is, "If **BQP** doesn't seem to be contained in **P**, and maybe not even in **NP**, then what *is* it contained in?" In other words:
What classical class gives an upper bound on what a quantum computer can do?

Well, Bernstein and Vazirani showed that it's possible to simulate a quantum computer classically with exponential time and polynomial memory, basically by writing an amplitude of interest as a sum of exponentially many contributions, and then evaluating the contributions one by one, reusing the same memory each time, and adding them to a running total.

This gives us an upper bound: **BQP** ⊆ **PSPACE**, where **PSPACE** (Polynomial Space) is the class of problems solvable on a digital computer using a polynomial amount of memory, but possibly exponential time.

It's possible to get a better upper bound on **BQP**, but it involves other complexity classes that we won't define here.

So, what would it take to prove that **P** is different from **BQP**? Of course this would follow if Factoring wasn't in **P**, but proving the latter would require showing **P≠NP**! So, is there better hope for proving **P≠BQP** in the near future than there is for proving **P≠NP**?

Unfortunately, not so much. The reason is that **BQP** is sandwiched between **P** and **PSPACE**:

P ⊆ **BQP** ⊆ **PSPACE**.

For this reason, any proof of **P≠BQP** would also need to show that **P≠PSPACE**, which is a big unsolved problem in itself.