Lecture 6: Examples of query problems

Andrew J. Landahl, alandahl@unm.edu
(**DRAFT**: September 19, 2007)

1 Query complexity

The question I want to address in today's lecture is, "How hard is it to compute a function $g : X \to Y$?" This sounds like a question with no objective answer. For example, your buddy Larry might be able to compute every $g(x)$ in a flash, whereas your other buddy Moe might take an eternity just to figure out a single $g(x)$. Well, I've got good news and bad news. The good news is that one *can* objectively quantify how hard it is to compute g. The bad news is that there are *many* different ways of doing so! So is that progress? I don't know. But what I do know is that today's lecture is going to focus on just one of these objective measures called the *query complexity* of g.

To make our task a bit simpler, let's restrict our attention to functions between *finite* sets. Moreover, let's restrict our attention further to functions whose input is a subset of N-letter strings. In other words, for today's lecture we'll consider functions $g : X \to Y$ where the following is true:

$$\begin{aligned} &N \in \mathbb{N} := \{1, 2, 3, \ldots\}, \\ &X \subseteq \Sigma^N, \text{ where } \Sigma \text{ is a finite set of } \textit{symbols} \text{ or } \textit{letters}, \\ &Y \text{ is a finite set.} \end{aligned}$$

We'll give special names to g when its input or output set satisfies special properties. In particular, we'll use the following nomenclature:

$$\begin{aligned} X = \Sigma^N &: g \text{ is a } \textit{total} \text{ function.} \\ X \subsetneq \Sigma^N &: g \text{ is a } \textit{partial} \text{ function.} \\ Y = \mathbb{B} &: g \text{ is a } \textit{Boolean} \text{ function.} \end{aligned}$$

A measure of how hard it is to compute g is called a measure of its *computational complexity*, or briefly, its *complexity*. Most such measures express how hard it is to compute g for the *worst-case* input. For example, if $X = \mathbb{B}$ and $g(0)$ takes an hour to compute but $g(1)$ takes ten hours to compute, then a typical measure would say that g has a computational complexity of ten hours. That's pretty darned conservative! In many practical situations it is the *average-case* complexity that is more relevant. For example, if one knew that $g(0)$ was needed 90% of the time and $g(1)$ was only needed 10% of the time, then a more practical complexity measure would report $90\%(1 \text{ hour}) + 10\%(10 \text{ hours}) = 1.9$ hours. The problem with average-case measures, discovered by computer scientists long ago, is that they are extremely sensitive to the whims of their priors—sometimes even a minor change will change the average-case complexity dramatically. A safer, and perhaps more lawyerly, approach is

to stick to worst-case complexity. We're going to continue this fine cover-your-butt tradition here.

* * *

As I mentioned, the computational complexity measure we're going to focus on today is the *query complexity*. Let's denote the query complexity of g by $D(g)$ and define it as follows:

$$D(g) := \min_{q \in \mathbb{N}} \max_{x \in X} \{q \mid \ q \text{ letters of } x \text{ determine } g(x)\}.$$

In words, $D(g)$ is the minimum number of input letters needed to evaluate $g(x)$ for the worst-case input x to g. When N is considered as a variable, so that g describes a family of functions indexed by N, we will express $D(g)$ as a function of N.

There are at least two strange things about the notation and language I've used so far. The first is my choice of the letter D for something called the "query complexity." It turns out that $D(g)$ is just one of several query complexity measures that we will encounter, and this one is conventionally called the "deterministic classical query complexity," hence the D. The second is my use of the word "query" in naming the measure—this word doesn't appear anywhere in its definition! The reason for using this word is that we can imagine a process whereby letters of an input x are obtained by a *physical* process called a *query*. This is a prime example of where physics and information collide—the input x must be stored in a physical system, so accessing its letters must be a physical process!

If the introduction of the word 'query' doesn't convince you that computer scientists were crying out for help from their physicist friends, the introduction of the word 'oracle' should. You see, some computer scientists have gone so far as to describe they querying process as follows. The input string x is held by an "oracle," and individual letters of x are accessed by making queries to the oracle. So a query exchange might go something like the following:

"Oh great oracle, what is the first letter of your input?"

"Z"

"Oh great oracle, what is the second letter of your input?"

"Z"

"Oh great oracle, what is the third letter of your input?"

"Z"

"I think he's sleeping!"

Okay, all melodrama aside, it is clear that computer scientists recognized something funny was going on when they introduced the words "query" and "oracle" to the computational complexity lexicon.

At this point, in a typical course covering quantum algorithms, the lecturer carefully describes a number of physical *query models* and the properties they can have, such as being reversible or irreversible, being deterministic, randomized, or quantum, or being exact, zero-error, or bounded-error. I started to go in that direction in the last lecture, but afterwards a number of students told me that they were having trouble getting what this whole querying business was about in the first place. So what I've decided to do is defer a rigorous description

of query models and their properties and instead go through a large number of examples of query problems first. In order to do that, I need to give at least a rough model for what a query is, so let's let it be the following: a query corresponds to submitting an $i \in \{1, \ldots, N\}$ to the oracle and obtaining the letter x_i in response. Without further ado, let the examples begin!

2 Examples of query problems

2.1 Deutsch's problem

Deutsch's problem[1] is the problem of computing the parity of two bits. In the notation we've developed, it's the problem of computing the total Boolean function with the following set of parameters:

$$N = 2, \qquad \Sigma = \mathbb{B}, \qquad Y = \mathbb{B}, \qquad X = \mathbb{B}^2, \tag{1}$$

$$g(x) := x_0 \oplus x_1. \tag{2}$$

To visualize this problem, we can construct a table I'll call a *query table.* In such a table, all of the possible values for $x \in X$ are written as N-letter strings in columns labeled by $x^{(0)}, x^{(1)}, \ldots, x^{(|X|)}$. The rows of the table are labeled by the possible queries $i = 0, 1, \ldots N-1$ that could be made. The final row of the table lists the values g takes for each possible input. For Deutsch's problem, the query table is the following.

i	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
0	0	0	1	1
1	0	1	0	1
g	0	1	1	0

Important point: There is no meaning to the order of the columns in a query table. For Deustch's problem I chose to order them by the binary value of the strings, but any other ordering would have worked just as well.

Using this table, the question we'd like to answer is, what is the query complexity of Deutsch's problem? Suppose one queried the first letter ($i = 0$). Then x becomes restricted to one of the two boxes below: (`N.B., The boxes still need to be drawn in the table.`)

i	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
0	0	0	1	1
1	0	1	0	1
g	0	1	1	0

Does this determine the value of g? No. Each box has two possible g values available, as the bottom row of the query table indicates, so still one more query is needed. How about if

[1] That's David Deutsch, not our own Ivan Deutsch here at UNM.

one had queried the second letter instead ($i = 1$)? Then x would have been restricted to one of the following two different "boxes" below: (`N.B., The boxes still need to be drawn in the table.`)

i	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
0	0	0	1	1
1	0	1	0	1
g	0	1	1	0

Each of these boxes also has two possible g values available, so g isn't determined by this strategy either. In a nutshell, evaluating the parity of two bits requires two queries, so we have

$$\boxed{D(g) = 2.} \tag{3}$$

The conclusion that the query complexity of this problem is two is so transparent that it came as quite a surprise when, in 1989, David Deutsch showed that there is a quantum algorithm that computes g using just *one* quantum query.

2.2 Two-bit AND on even-parity strings

After considering Deutsch's problem, you may get the impression that the query complexity of every function on N letters is N. Here's a quick little example to show you that this isn't the case. Consider the following query problem:

$$N = 2, \qquad \Sigma = \mathbb{B}, \qquad Y = \mathbb{B}, \qquad X = \{00, 11\}, \tag{4}$$

$$g(x) := x_0 \wedge x_1. \tag{5}$$

This is a partial function, representing the AND of two bits of even parity. The query table for this problem is

i	$x^{(0)}$	$x^{(1)}$
0	0	1
1	0	1
g	0	1

It is clear from this table that once one bit of the input is known, the function g is determined entirely. Hence the query complexity is one, not two, for this partial function:

$$\boxed{D(g) = 1.} \tag{6}$$

2.3 The N-bit OR, a.k.a. Grover's problem

For our first example of a query problem with a variable-sized input, consider the problem of computing the OR of a collection of N bits. The parameters of the problem are as follows:

$$N \in \mathbb{N}, \qquad \Sigma = \mathbb{B}, \qquad Y = \mathbb{B}, \qquad X = \mathbb{B}^N, \tag{7}$$

$$g(x) := x_0 \vee \cdots \vee x_N. \tag{8}$$

The query table for this problem is

i	$x^{(0)}$	$x^{(1)}$	$\ldots$	$x^{(2^N-1)}$
0	0	0	$\ldots$	1
1	0	0	$\ldots$	1
$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\vdots$
$N-1$	0	1	$\ldots$	1
g	0	1	$\ldots$	1

How many queries does it take to compute g? Well in the worst case, every query returns a 0. It's not until the Nth query that one can distinguish between $g(x) = 0$ and $g(x) = 1$. So the query complexity of this problem is N:

$$\boxed{D(g) = N.} \tag{9}$$

Although it's pretty obvious what the query complexity of this problem is, I want to point out some of features of N-bit OR before leaving it. The first is that because this is an N-letter problem, the indices i sent to the oracle range from 0 to $N-1$. Hence each index requires $\lceil \log_2 N \rceil$ bits to represent it, where $\lceil a \rceil$ denotes the largest integer greater than or equal to a. The key point I want to get across is that the size of this "query register" is totally independent of the size of the alphabet Σ—it only depends on N.

The second point I'd like to make is that the query complexity of this problem is N even when g is just the *partial* function defined on input strings having at most one '1' in them, *i.e.*, on the input set

$$X = \{0^N\} \cup \{0^k 1 0^{N-k-2} \mid k = 0, \ldots, N-2\} \tag{10}$$

A more compact way of representing this set is to use the notion of the *Hamming weight* or more briefly the *weight* of a bit string. The Hamming weight of a string $x \in \mathbb{B}^N$ is defined as

$$\mathrm{wt}(x) := \text{The number of occurrences of the letter '1' in } x. \tag{11}$$

So using this function, we could equally well define X as

$$X = \{x \in \mathbb{B}^N \mid \mathrm{wt}(x) \leq 1\}. \tag{12}$$

The reason that $D(g)$ is unchanged in this case is that $g = 0$ only for one input as before, and in the worst case every letter of x needs to be known to figure this out—a single 1 for any input letter will make the function evaluate to 1 instead of 0.

Sometimes this problem, or its restriction to weight-one-or-less strings, is called the UNORDERED SEARCH problem. The reason for this is that it is the same as the problem of trying to find an item in an unordered list $x = (x_0, \ldots, x_N)$ where each x_i is no longer necessarily a bit and the queries are not of the form "What is the ith letter of x?" but rather "Is x_i the same as the item I am looking for?" For example, finding the ace of spades

in an unshuffled deck is an instance of the UNORDERED SEARCH problem. Regardless of the nature of the items in the list, be they elephants, typewriters, or grapes, these kind of *comparison* queries yield only a single bit: 1 if it is a match and 0 if it isn't, so the N-bit Boolean OR function captures this process exactly.

A final comment about this problem: Why have I called it Grover's problem? Well the reason doesn't have anything to do with a lovable furry blue Muppet, in case you were wondering. No, it has to do with the fact that Lov Grover in 1996, then at Bell Labs, discovered a quantum algorithm for this problem that requires only on the order of $\sqrt{N}$ quantum queries. His quantum algorithm achieving this was within a constant multiple of a lower bound on the best-possible quantum algorithm for this problem, proved two years earlier in 1994 by Bernstein, Bennett, Brassard, and Vazirani.

2.4 The ordered search problem

In the previous problem, we saw that the UNORDERED SEARCH problem with comparison queries could be mapped onto the N-bit OR problem with ordinary letter queries. Can the same kind of mapping be done when the list is *ordered*? Yes! A query on an ordered list returns more information than a query on an unordered list. An ordered list query is of the form "Is x_i greater than, less than, or equal to the item I'm looking for?" This is a three-valued answer, but if our goal is simply to find the *first* instance of the item we are looking for in the list, then queries of the form "Is x_i greater-than-or-equal-to or less than the item I'm looking for?" will suffice. Of course, there is the possibility that the item is *nowhere* in the list. To account for this, two versions of the ORDERED SEARCH problem are usually put forward. In the first version, there are N items in the list, and one is promised that the item being sought is somewhere in the list. In the second version, there are $N-1$ items in the list, and the object is to *insert* the item into the list such that it preserves the list's order. Since an $(N-1)$-element list has N insertion points (including at the very beginning and the very end), this is the same problem as the first formulation, because one is guaranteed that there is an insertion point *somewhere*.

Because of the binary-valued comparison questions being used (1 = "greater-than-or-equal-to", 0 = "equal to") and the ordering of the list, one can map the ORDERED SEARCH problem onto the following problem using ordinary letter queries on the partial function g:

$$N \in \mathbb{N}, \qquad \Sigma = \mathbb{B}, \qquad Y = \mathbb{Z}_N, \tag{13}$$

$$X = \{0^k 1^{N-k} \mid 0 \leq k \leq N-1\}, \tag{14}$$

$$g(x) := \min_k \{x_k = 1\}. \tag{15}$$

Unlike our letter-query model for UNORDERED SEARCH, our letter-query model for ORDERED SEARCH returns an *index* from 0 to $N-1$ (*i.e.*, in $\mathbb{Z}_N$), not a bit. So this is our first example where Y isn't just $\mathbb{B}$, making g a non-Boolean function.

To determine the query complexity of this problem, we can construct a query problem

table as before:

i	$x^{(0)}$	$x^{(1)}$	...	$x^{(N-1)}$
0	0	0	...	1
1	0	0	...	1
⋮	⋮	⋮	...	⋮
$N-2$	0	1	...	1
$N-1$	1	1	...	1
g	N	$N-1$	...	1

It's relatively straightforward to see that $\lceil \log_2 N \rceil$ queries suffice to evaluate g. The algorithm that achieves this is the familiar *binary search* algorithm. One first queries at the middle of the string, ruling out half the possible strings, then queries at the middle of the remaining substring, ruling out half of the substrings, and so on. Since each query divides the space of possibilities by two, the query complexity is

$$\boxed{D(g) = \lceil \log_2 N \rceil.} \tag{16}$$

Proving that no algorithm can do any better is a little trickier. A rigorous argument involves information-theoretic reasoning and is beyond the scope of where we are currently in the course. For now we can just use casual reasoning and note that since there are N possible values of g that we need to determine, and it takes $\lceil \log_2 N \rceil$ bits to represent the answer, then at least that many queries need to be made.

The current status (ca. Sep. 2007) of the quantum query complexity of this problem is open. It is known that no quantum algorithm can solve this problem using fewer than $\frac{1}{\pi} \ln N$ queries, but the best known algorithm is one I developed with Andrew Childs and Pablo Parrilo that uses $4 \log_{605} N \cong 0.43 \log_2 N$ quantum queries. Almost certainly fewer queries suffice, but whether it is possible to attain the lower bound is unknown.

2.5 The Deutsch-Jozsa problem

Historically, the Deutsch-Jozsa problem was the first query problem scaling with N that was studied in a quantum context. It's named for David Deutsch and Richard Jozsa, who wrote about it in 1992. I should mention that Jozsa's last name is spelled "Jozsa" and not "Josza." I'm always getting that wrong. Richard's a great guy from Australia (currently a professor in the UK)—the kind of guy you'd like to have a beer with—but he's got this last name that vexes me every time! Anyway, the problem he and David Deutsch studied was the problem of determining whether an input N-bit string was "constant," meaning that it was either all 0s or all 1s, or "balanced," meaning that it had half 0s and half 1s, given that one or the other was promised to be the case. In the notation we've developed, their problem is as

follows:

$$N \in 2\mathbb{N}, \qquad \Sigma = \mathbb{B}, \qquad Y = \mathbb{B}, \tag{17}$$

$$X = \underbrace{\{0^N, 1^N\}}_{\text{constant}} \cup \underbrace{\{x \in \mathbb{B}^N \mid \text{wt}(x) = N/2\}}_{\text{balanced}}, \tag{18}$$

$$g(x) = \begin{cases} 0 & x \text{ is constant} \\ 1 & x \text{ is balanced.} \end{cases} \tag{19}$$

There are $\binom{N}{N/2} \sim 2^{N/2}$ possible balanced strings, so our query problem table for this problem looks like

i	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\ldots$	$x^{\left(2+\binom{N}{N/2}\right)}$
0	0	1	0	0	$\ldots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\vdots$
$N-3$	0	1	0	1	$\ldots$	1
$N-2$	0	1	1	0	$\ldots$	1
$N-1$	0	1	1	1	$\ldots$	1
g	0	0	1	1	$\ldots$	1

How many queries are needed to evaluate g in the worst case? Well, in the worst case, the first $N/2$ queries all yield the same result, leaving g undecided. However with one more query, g becomes completely known. Hence the query complexity of this problem is

$$\boxed{D(g) = \frac{N}{2} + 1.} \tag{20}$$

The amazing thing about this problem is that Deutsch and Jozsa showed that this function can be computed in just a single quantum query!

This problem can be stated in a more convoluted way that will make the subsequent query problem's we will discuss appear more transparent. We'll do it as follows. Let $s \in BB$ and let the input set to G be defined in terms of s:

$$X = \{x \in \mathbb{B}^N \mid \text{wt}(x) = \frac{N}{2^s} \bmod N\}.$$

The function g that the Deutsch-Jozsa problem considers is then $g(x) = s$.

2.6 The Bernstein-Vazirani problem

The Bernstein-Vazirani problem is to determine the query complexity of a partial non-Boolean function defined on bit strings whose length is a power of two. The formal definition of the problem is

$$n \in 0 \cup \mathbb{N}, \qquad N = 2^n, \qquad \Sigma = \mathbb{B}, \qquad Y = \mathbb{B}^n, \tag{21}$$

$$X = \{x = (x_0, \ldots x_{N-1}) \in \mathbb{B}^N \mid x_i = i \cdot s\} \tag{22}$$

$$g(x) = s. \tag{23}$$

As with the Deutsch-Jozsa problem, there is a parameter s—in this case, s is an n-bit string—that is used to define the structure of the input set, and the function g returns its value. The notation $i \cdot s$ indicates the *bitwise inner product* between the two n-bit strings:

$$i := (i_0, \ldots, i_n) \qquad s := (s_0, \ldots, s_n)$$
$$i \cdot s := i_0 s_0 \oplus \cdots \oplus i_n s_n.$$

What is the query complexity of this problem? Rather than writing out a query table, it's clearer to just reason about the problem directly. To learn s, it suffices to learn each of its n bits. Every time a query is made at an index that is a power of two, so that the index is a weight-one n-bit string, a single bit of s is obtained. This is because x_{2^k} equals the kth bit of s according to the definition of X and the bitwise inner product. So querying $i = 1, 2, 4, 8, 16, \ldots$ will allow g to be evaluated in n queries, which is the number of bits needed to describe s. Hence, the query complexity of the Bernstein-Vazirani problem is

$$\boxed{D(g) = \log_2 N.} \tag{24}$$

Why is this called the Bernstein-Vazirani problem, you ask? Well because it was studied by Bernstein and Vazirani of course! Seriously, Bernstein and Vazirani showed that there is a quantum algorithm for this problem that makes only a single quantum query. More significantly, they considered a generalized "recursive" version of this problem that requires on the order of $\log_2 \log_2 N$ quantum queries whereas the best possible classical algorithm requires $\log_2 N \log_2 \log_2 N$ queries (even when randomized algorithms are allowed, which we haven't talked about yet). Since the inputs to a query algorithm are indices, which require n bits to specify, sometimes query complexity is expressed as a function of n rather than N. Doing so reveals that the best possible classical query algorithm for the recursive Bernstein-Vazirani problem requires on the order $n^{\log_2 n}$ queries whereas the Bernstein-Vazirani quantum query algorithm uses only $\log_2 n$ queries. Historically, this was the first problem for which a super-polynomial separation in query complexity was demonstrated between classical and quantum query algorithms.

`Add boxes to Deutsch problem query tables.`	ToDo