Notes on Communication Complexity

Communication complexity is a modern subject of theoretical computer science that is extremely relevant to various practical settings. The basic concepts are pretty simple to understand, and it is relatively easy to prove interesting results on the topic. We'll introduce this model, show you some basic properties of it, and then talk about how the model ties in nicely with DFAs and Streaming Algorithms.

The set-up for communication complexity is a theoretical model of distributed computing with two computers. In general, we want to compute some interesting function

$$f : \{0,1\}^\star \times \{0,1\}^\star \to \{0,1\}.$$

That is, our function f takes two binary strings, and outputs a bit. (For simplicity, we will stick to the binary alphabet throughout these notes.)

So f requires an input $x \in \{0,1\}^\star$ and an input $y \in \{0,1\}^\star$. We will assume $|x| = |y| = n$, where n is, in general, a **huge** integer. Furthermore, we assume the input (x, y) is stored across two computers **Alice** and **Bob**. Alice holds x, and Bob holds y. Let's suppose Alice and Bob are very far apart – for example, they may be on different planets, or different sides of the earth. Their goal is to cooperate together in a *communication protocol* for computing $f(x, y)$, by *communicating bits about their inputs to each other*. To keep the discussion as simple and focused as possible, we will even assume that Alice and Bob are super-powerful computers, so we don't even care about the computational cost of f. We will only care about the number of bits communicated between Alice and Bob in order to determine $f(x, y)$.

1 The Model and Definitions

Let's be a little more formal, and first define what is a protocol.

Definition 1 *A* **communication protocol** *is a pair of functions* $A, B : \{0,1\}^\star \times \{0,1\}^\star \to \{0, 1, \texttt{STOP}\}$.

So we think of Alice and Bob as implementing functions A and B, respectively, and both functions take two inputs. The first input to A is the input x, and the second input to A is the *communication history* H: the sequence of bits sent back and forth so far in the communication. Similarly, the first input to B is the input y, and the second input is the communication history H. The output of $A(x, H)$ is the bit that

Alice would send, if it was her turn to speak. Similarly, the output of $B(y, H)$ is the bit that Bob would send.

In general, Alice and Bob will speak one bit each, in rounds. Alice will always start the communication, and speak one bit in odd-numbered rounds. Bob will always speak a bit in even-numbered rounds. When one party says `STOP`, the communication protocol ends, and the previous bit spoken is taken to be the output of the protocol.

Example 2 *Suppose Alice holds x and Bob holds y. The protocol starts with Alice, who computes a bit $b_1 := A(x, \varepsilon)$ and sends b_1 to Bob. (Since the communication has just started, there is no history, so the history string is ε.) Bob receives b_1, computes a bit $b_2 := B(y, b_1)$ (because b_1 is part of the history), and sends b_2 to Alice. Alice receives b_2 and computes $b_3 := A(x, b_1 b_2)$ (because the history is now $b_1 b_2$). Now, let's suppose b_3 is not a bit, but rather the symbol* `STOP`*. When Alice sends* `STOP` *to Bob, the communication ends, the output of the protocol is the bit b_2, and we would say that this protocol took two rounds on (x, y): two bits were communicated.*

Let's write down the above requirements in pseudocode. Given a pair of functions $A, B : \{0,1\}^\star \times \{0,1\}^\star \to \{0, 1, \texttt{STOP}\}$, the function they compute is determined by the following process:

```
On input (x, y),
 Let r := 0, b_0 := ε.
 While (b_r ≠ STOP),
  r := r = 1
  If r is odd, Alice sends b_r := A(x, b_1 ··· b_{r−1})
       else, Bob sends b_r := B(y, b_1 ··· b_{r−1}).
Output b_{r−1}.
```

We say that a protocol (A, B) **computes the function** f if for all $(x, y) \in \{0,1\}^\star \times \{0,1\}^\star$, the output bit b_{r-1} of the protocol equals $f(x, y)$.

Each iteration of the while loop is one *round* of the protocol. In the pseudocode, the *total number of rounds is the value of $r-1$ at the end.* Observe that b_i is the bit sent in the i-th round. Note that the number of rounds is exactly the number the number of bits communicated between Alice and Bob; we want to minimize this. We say that the total number of rounds is the *communication complexity of the protocol (A, B) on the input (x, y)*. As is typical for complexity, we measure this communication complexity in terms of the worst-case pair of inputs (x, y) that make (A, B) communicate as much as possible.

Definition 3 *The* **cost of the protocol** (A, B) *on n-bit strings is*

$$\max_{x,y \in \{0,1\}^n} [\textit{number of rounds taken by } (A, B) \textit{ on } (x, y)] .$$

So the cost is a worst-case measure: we look at the largest number of rounds needed to compute $f(x, y)$ over all x, y of n bits.

Definition 4 *For each n, the* **communication complexity** *of f on n-bit strings, called $cc(f)$, is the minimum cost over all protocols computing f on n-bit strings. Alternatively, this is the minimum number of rounds used by any protocol computing $f(x, y)$, over all n-bit inputs x and y.*

As we have defined it, $cc(f)$ is a function from $\mathbb{N}$ to $\mathbb{N}$: for every n, there is some minimum communication cost to computing f on n-bit inputs. So technically, we could write it as $cc(f)(n)$, but the double parentheses notation can be confusing. To keep the notation simpler, we will assume that the variable n is always the input length to $cc(f)$, and write $cc(f)$ as a function of the variable n.

2 Some Examples

Let's start with the simplest possible example. Let $f : \{0,1\}^\star \times \{0,1\}^\star \to \{0,1\}$ be arbitrary.

There is always a "trivial" protocol for f that uses $2n$ rounds. Alice simply sends the bits of her x over to Bob, in odd-numbered rounds. Bob may send whatever bit he wants in even-numbered rounds (Alice can ignore his blather). After $2n-1$ rounds of this lopsided interaction, Alice has sent her entire n-bit x. So now, Bob knows x and (because we're assuming they're all-powerful) Bob can send $f(x, y)$ in the next round. Formally, this means:

Proposition 1 *For every f, $cc(f) \leq 2n$.*

However, recall our assumption that n is very large. A protocol using $\Theta(n)$ bits of communication is highly undesirable, just as a streaming algorithm using $\Theta(n)$ bits of space would be. We want to know: for what problems we can get protocols using much less than n bits of communication?

2.1 PARITY

The PARITY function is a good example of a problem that requires extremely low communication. We define

$$\text{PARITY}(x, y) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i \bmod 2.$$

So, the PARITY function just computes the sum of all bits in Alice and Bob's inputs, modulo 2. What's a good protocol for computing PARITY? Well, Alice could simply send $b_1 = \sum_i x_i \bmod 2$ to Bob. Then Bob can compute $b_2 = b_1 + \sum_i y_i \bmod 2$. This is the same as PARITY(x, y), so Alice can `STOP` after that!

Proposition 2 *$cc(PARITY) \leq 2$.*

It's also the case that $cc(\text{PARITY}) \geq 2$. Each party needs to send at least one bit to the other party, because PARITY depends on both party's inputs. If the communication complexity was 1, this means that Alice could simply send Bob the answer to PARITY(x, y) without even looking at Bob's input! But this is impossible: if Bob flipped one bit in his own input then Alice's answer should change, but her answer doesn't depend on Bob's input at all.

2.2 MAJORITY

The MAJORITY function, MAJORITY(x, y), computes the most frequent bit in the string xy. (And if there is an equal number of 1's and 0's in xy, then we say MAJORITY$(x, y) = 1$.) This is a natural voting problem for an election: suppose two polling places are far away from each other, they've both tallied up n votes for candidate 0 versus candidate 1, and they (only) want to determine the winner of the election.

What's a good protocol for computing MAJORITY? The natural thing to do is to have Alice and Bob send the vote counts! Over multiple rounds, when it's Alice's turn, she sends bits of the integer N_x which is the number of 1s in x, Bob computes N_y which is the number of 1s in y, and Bob sends 1 to Alice if and only if $N_x + N_y$ is greater than n. (Note the total number of voters is $2n$.) If Alice sends N_x encoded in binary, this takes $O(\log n)$ rounds. We therefore have:

Proposition 3 *$cc(MAJORITY) \leq O(\log n)$.*

2.3 EQUALS

The function EQUALS, defined by

$$EQUALS(x, y) = 1 \iff x = y,$$

is a very significant function. It's often the case that databases have "mirrors" that hold exactly the same information but are geographically far apart, to minimize their access times for users. A common problem is that errors could happen on one database, but not the other, or maybe some updates occur on one database but not

the other. So one would like to regularly check if the two database mirrors are still holding exactly the same information. This is precisely what EQUALS models!

What's a good protocol for EQUALS? Hmm. Somehow Alice and Bob have to figure out where their strings differ, but they could differ in only one bit. We could try the following: Alice sends the odd-numbered bits of her input in odd-numbered rounds, and Bob sends the even-numbered bits of his input in even-numbered rounds. After these bits are communicated in n rounds, Alice knows all of Bob's *even-numbered* bits, and Bob knows all of Alice's *odd-numbered* bits. If the two strings are not equal, then at least one of the two parties now knows that! (If x and y are different, they're either different in an even-numbered bit, or they're different in an odd-numbered bit, or both.) Let's suppose n is even, so that after n rounds, it's Alice's turn. (If n is odd, we can just exchange the roles of Alice and Bob below.) Alice sends 0 if the strings differ in an even-numbered bit, and sends 1 if they do not. Bob can reply `STOP` if Alice sent 0. Otherwise, he replies 0 if the strings differ in an odd-numbered bit, and 1 otherwise. This takes $n + 2$ rounds.

Proposition 4 *$cc(EQUALS) \leq n + 2$.*

Can we do better than this? They are basically sending their inputs to each other!

To be continued...

3 Connection to Streaming Algorithms and DFAs

We can use communication complexity as a tool to understand the space usage of a streaming algorithm, and to prove that some languages are not regular. To understand how, we first show how to re-interpret every language $L \subset \{0,1\}^\star$ as a communication problem $f_L : \{0,1\}^\star \times \{0,1\}^\star \to \{0,1\}$. The idea is to give Alice one half of the input, give Bob the other half, and ask them to determine whether or not the concatenation of their inputs is in L. Formally, for inputs $x, y \in \{0,1\}^\star$ where $|x| = |y|$, we define

$$f_L(x,y) = \begin{cases} 1 & \text{if } xy \in L \\ 0 & \text{otherwise.} \end{cases}$$

Let us start with a simple example that may be instructive:

Example 5 *$L = \{z \mid z$ begins with a $0\}$. Recall this is a regular language. Observe that $cc(f_L) = 1$: Alice holds the first half of the input xy, so she only has to communicate to Bob the answer!*

Example 6 *Here are some languages L, where the corresponding functions f_L are communication problems that we defined earlier in these notes.*

- $L = \{z \mid z$ *has an odd number of* $1s\}$.
  Then, $f_L(x, y)$ *equals the PARITY function that we defined earlier!*
- $L = \{z \mid z$ *has at least as many* $1s$ *then* $0s\}$.
  Then, $f_L(x, y)$ *equals the MAJORITY function.*
- $L = \{z \mid z = xx$ *for some* $x\}$.
  Then, $f_L(x, y)$ *equals the EQUALS function.*

It is not hard to imagine that there are many more examples beyond these; we have picked these to be illustrative.

3.1 The Connection

Now that for every language L we have a communication problem f_L, we can state the connection:

Theorem 7 *If L has a streaming algorithm using $\leq s(n)$ bits of space on all inputs of length $\leq 2n$, then $cc(f_L) \leq 4s(n) + 4$.*

That is, if L has a good low-space streaming algorithm, then the communication complexity of f_L is similarly low: Alice and Bob don't have to communicate much to decide if $xy \in L$. (Note: you will improve this protocol in your next pset/pset, so don't worry if the below proof looks sub-optimal — it is!)

PROOF: Here is the general idea. Suppose we have a streaming algorithm A for L that uses $\leq s(n)$ bits of space on inputs of length n. We want an efficient protocol for computing f_L. How should Alice and Bob speak to each other?

Naturally, they should use the streaming algorithm A! Alice starts by running A on her input x, and when she is done reading x she has reached some memory state m of the algorithm. Then, she *passes the state m over to Bob as a bit string*, using $4s(n) + 3$ rounds of communication. Then, Bob runs A on his input y, starting from the memory state m. Eventually he gets an output bit, which he then sends back to Alice as the final output.

How can Alice send her memory state in $4s(n) + 3$ rounds? The tricky part is that, because A uses $\leq s(n)$ bits of space on inputs of length $\leq 2n$, the memory state as a bit string can have variable length: the length could be any integer between 0 and $s(n)$. We need to encode the memory state in a way that Bob can easily interpret it. One way of doing this is to re-encode m into a bit string m' of length $2|m| + 2$: each 0-bit of m is replaced with 00, each 1-bit of m is replaced with 01, and we put 11 at the end to indicate the end of the string. Now, if Alice sends this m', Bob will

be able to tell when the string encoding the memory state has ended, and can start running the streaming algorithm on his input.

Recall that Alice only speaks in odd-numbered rounds; we could have Bob just send back the bit 0 in even-numbered rounds, until Alice has sent the entire memory state. Using the above encoding, sending the memory state m takes in total $4s(n)+3$ rounds: Sending one bit of m takes four rounds each, and three more rounds (Alice-Bob-Alice) for Alice to send the 11 string at the end of m'. $\square$

We note that the above theorem extends to streaming algorithms that make multiple passes over the input as well.

Theorem 8 *If L has a k-pass streaming algorithm using $\leq s(n)$ space (in bits), then $cc(f_L) \leq O(k \cdot s(n))$.*

The idea behind this extension is simple: for a streaming algorithm that makes k passes, Alice and Bob can simply repeat the protocol above for k times, once for each pass over the input. (We have used big-O notation here to hide/forget the specific constants involved.)

We can use Theorem 7 to quickly get interesting communication protocols for known functions.

Corollary 9 *For every regular language L, $cc(f_L) \leq O(1)$.*

Corollary 9 follows because every regular language L can be modeled by an $O(1)$-space (constant-space) streaming algorithm! (In fact, given our definitions, DFAs are *exactly* the $O(1)$-space one-pass streaming algorithms.)

As an example, remember that $L = \{z \mid z \text{ has an odd number of 1s}\}$ is a regular language, and that its corresponding function $f_L = \text{PARITY}$. We already saw that $cc(\text{PARITY}) = 2$, so Corollary 9 is consistent with these observations.

Corollary 10 *$cc(MAJORITY) \leq O(\log n)$.*

This follows because there's a streaming algorithm for

$$L' = \{z \mid z \text{ has at least as many 1s then 0s}\},$$

and $f_{L'} = \text{MAJORITY}$.

Still, these theorems don't help us get a protocol for EQUALS, because the corresponding language $\{z \mid z = xx\}$ requires large space with streaming algorithms!

4 Lower Bounds on Communication Complexity

Earlier we saw that $cc(\text{EQUALS}) \leq n+2$ with what looked like a pretty bad protocol. We will show that is practically the best possible protocol one could have!

Theorem 11 *$cc(EQUALS) \geq n$. That is, every communication protocol for EQUALS must communicate at least n bits between Alice and Bob.*

In other words, no communication protocol for EQUALS can do much better than essentially sending your input, no matter how Alice and Bob speak back and forth! This is a very powerful lower bound. The key to this lower bound is a simple concept and lemma:

Definition 12 *Let (A, B) be a communication protocol. The* **communication pattern** *of (A, B) on the input (x, y) is the communication history of Alice and Bob on (x, y), including the output bit.*

Example 13 *For example, suppose for A and B running on the input (x, y), A sends 0, B sends 1, A sends 1, B sends 0, and A sends* `STOP`*. Then the pattern on (x, y) is 0110.*

The following key lemma shows that if two distinct input pairs have the same communication pattern, then two *other* pairs also have the same pattern.

Lemma 14 (Key Lemma for Communication Complexity) *If the input pairs (x, y) and (x', y') both have the same communication pattern P on a protocol, then the input pairs (x, y') and (x', y) also have the communication pattern P.*

Proof: Remember how protocols work: each bit sent is a function of the player's input and the bits seen so far. Let Alice's protocol be A, and let Bob's be B. Suppose the input pairs (x, y) and (x', y') both have the same communication pattern $P = b_1 \cdots b_k$. So, Alice and Bob speak for k rounds on both (x, y) and (x', y'), and say exactly the same bits to each other, and conclude exactly the same output.

Then we have:

$$\begin{aligned} b_1 = A(x, \varepsilon) = A(x', \varepsilon), \\ \text{for all even } i, b_{i+1} = A(x, b_1 \cdots b_i) = A(x', b_1 \cdots b_i) \\ \text{for all odd } i, b_{i+1} = B(y, b_1 \cdots b_i) = B(y', b_1 \cdots b_i) \end{aligned}$$

From the above equations, it follows that the pattern for (x', y) and for (x, y') is also $b_1 \cdots b_k$. In particular, for $i = 1, \ldots, k$, the i-th bit of the pattern for (x', y) (and for (x, y')) is b_i.

To see this, note that since $b_1 = A(x, \varepsilon) = A(x', \varepsilon)$, we know the first bit of the pattern for (x', y) and the pattern for (x, y') is b_1. Then, since $b_2 = B(y, b_1) = B(y', b_1)$, we conclude that the second bit of the pattern for (x', y) and for (x, y') is b_2. Continuing in this way, we find that all four of (x, y), (x', y'), (x', y), and (x, y') must have the same communication pattern! (If you wanted to be super-formal, you could use induction on $i = 1, \ldots, k$ to prove this.) $\square$

Now we are ready to prove the communication lower bound for EQUALS:

Proof: Of Theorem 11

By contradiction. Assume $cc(\text{EQUALS}) \leq n - 1$. That means there is a protocol for computing EQUALS that uses at most $n-1$ bits of communication on all input pairs (x, y) where $|x| = |y| = n$.

For such a protocol, how many possible communication patterns can there be, on inputs x and y of length n? By assumption, each communication pattern in this protocol is a binary string of length at most $n - 1$, and there are

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

such binary strings. So this EQUALS protocol has at most $2^n - 1$ possible communication patterns, over all pairs (x, y).

Now consider the set of input pairs $S = \{(x, x) \mid x \in \{0, 1\}^n\}$. Clearly $|S| = 2^n$. By the pigeonhole principle, there must be two pairs (x, x) and (y, y) from S, where $x \neq y$, for which our protocol has the *same* communication pattern. Call this pattern P. By the Key Lemma, the pairs (x, y) and (y, x) must *also* have communication pattern P on this protocol. Since the communication pattern includes the output bit, Alice and Bob must output the same bit on all pairs (x, x), (y, y), (x, y) and (y, x). However,

$$\text{EQUALS}(x, x) = \text{EQUALS}(x, x) = 1, \text{ but } \text{EQUALS}(x, y) = \text{EQUALS}(y, x) = 0.$$

This is a contradiction! $\square$

In general, one can prove a communication complexity lower bound for other functions, using the following set-up. Take your favorite communication problem f that you want to prove a lower bound for. Then, your lower bound proof can have the following framework:

- Assume $cc(f) \leq k(n)$.
- Therefore, there is a protocol for f over n-bit strings, where Alice and Bob can compute f and exchange at most $k(n)$ bits.

- Find a set $S = \{(x, y)\}$ of devilish input pairs, where $|S| \geq 2^{k(n)+1}$.
- By pigeonhole, there must be two pairs (x, y) and (x', y') from S which have the *same* communication pattern.
- By the Key Lemma, all four pairs (x, y), (x', y'), (x', y) and (x, y') must have the same communication pattern.
- Therefore $f(x, y) = f(x', y') = f(x', y) = f(x, y')$. If your set S was devilish enough, you will have a contradiction.

You will work through other lower bounds in the pset/pests!

4.1 Applications to Streaming Lower Bounds

The contrapositives of Theorem 7 and Theorem 8, combined with our lower bounds on communication, can be used to prove powerful lower bounds on streaming algorithms, even with multiple passes:

Theorem 15 (Contrapositive of Theorem 8) *Let $L \subseteq \{0,1\}^\star$. Suppose $cc(f_L) \geq \Omega(k \cdot s(n))$. Then every k-pass streaming algorithm for L needs to use $\leq s(n)$ space (in bits).*

For example, we know $cc(\text{EQUALS}) \geq n$, and we know that for $L = \{z \mid z = xx \text{ for some } x\}$, $f_L = \text{EQUALS}$. Applying Theorem 15:

Corollary 16 *Every streaming algorithm for the language $L = \{z \mid z = xx \text{ for some } x\}$ must use $\geq cn$ bits of space, for some constant $c > 0$.*

In other words, the space lower bound for L is $\Omega(n)$. Even a streaming algorithm that used two passes, or ten passes, or a hundred passes, would need $\Omega(n)$ bits of space (the constant in that Ω would be decreasing as the number of passes goes up, but it would still be constant).

5 Some Good News (Optional Reading)

There is a bright side to all of the sad lower bounds for communication. While EQUALS needs at least n bits of communication in our *deterministic* model, there are very efficient protocols for computing EQUALS using *randomness*!

Theorem 17 *There is a randomized protocol for EQUALS(x, y) using only $O(\log n)$ bits of communication on x, y of length n. On all (x, y), the protocol outputs the correct answer with probability greater than 99.9%.*

The idea can be put in two words: **random hashing**. Alice and Bob think of their two n-bit strings x and y as *numbers* in the interval $[1, 2^n]$. Alice picks a random prime number p between 2 and n^2. She sends p and her number x modulo p. This is a number between 0 and n^2, so it takes $O(\log n)$ bits to send. Bob checks whether $y = x$ modulo p; if they are equal, he outputs 1, otherwise he outputs 0.

This protocol outputs the correct answer with (very) high probability! First, note that if $x = y$, then we always have $x = y \bmod p$. So if x and y are equal, this protocol always outputs the correct answer. Second, if $x \neq y$, how can this protocol fail? It would fail if we managed to pick a prime p such that $x - y \bmod p$ equals 0, i.e. p is a factor of $x - y$. (In such a case, the protocol would conclude $x = y$ when they actually aren't.) Now, $x - y$ is an integer in $[-2^{n+1}, 2^{n+1}]$; it is not hard to show that every such integer has at most $n + 1$ prime factors. Call these annoying primes "bad primes".

What are the chances we randomly picked a bad prime p? We are choosing p from $[2, n^2]$, and the prime number theorem says there are $\Theta(n^2/\log n)$ primes in that interval. So the chances we pick a "bad" prime is at most

$$\frac{n+1}{\Theta(n^2/\log n)} \leq \frac{O(\log n)}{n},$$

which is very low for large n.