A decision problem is a computational problem with yes/no answers, for example: Given a graph on k vertices, is the graph 3-colorable? In complexity theory we often describe such a problem by a function family $\{f_n \colon \{0,1\}^n \to \{0,1\}\}$, one for every input length n, which associate to each input a 0/1 answer, where 0 means no and 1 means yes. In the 3-coloring example, G could be described by its adjacency matrix, which we can view as a string of length $n = \binom{k}{2}$, and $f(G)$ takes value one if G is 3-colorable, and zero if not.

We usually want to know how much time it takes for the best possible algorithm to solve such a problem f. This depends not only on the algorithm but also on the choice of input. In worst-case complexity we ask that the algorithm solves f within some time bound t on all inputs. Average-case complexity takes the point of view that not all inputs are relevant, so it asks that we solve f in time t on some fraction of all 2^n possible inputs.

Some natural problems turn out to be easier on average than in the worst case. For instance, the 3-coloring problem is NP-hard in the worst case, but most graphs on k vertices are not 3-colorable; so while we do not expect to have an efficient worst-case algorithm for 3-coloring, the algorithm that ignores its input and outputs "no" all the time works correctly on most inputs.

Could it in fact be the case that all "reasonable" decision problems are easy to solve on average. The answer is not known, but there are many natural examples of decision problems that we do not know how to solve efficiently even on average, although we usually cannot explain why this is so. Impagliazzo and Wigderson showed examples of decision problems that are almost as hard to solve on the average as they are in the worst case.

1 The Impagliazzo-Widgerson theorem

To describe this result we need a bit of complexity theory; for our purposes we can keep the discussion at a fairly informal level. There are two reasonable models of computation in complexity theory: algorithms and circuits. An algorithm is a single program that you can run (at least in principle) on arbitrarily large inputs. A circuit family is an infinite sequence of programs, one for every input length. As the input length grows, not only is the program given more time to run but its size can also become larger.

A reasonable measure of the complexity of a computation is the size of the program *plus* the time it takes the program to terminate on the given input. One intuition for this is that in many computations, we have two extreme approaches. At one end, we look for the solution by brute force. This takes a lot of time but a very short program. At the other end, we hard-code all possible answers (for all inputs we can expect to get) into the program. Then the computation gives an answer immediately but the program is very long. This suggests to define the *complexity* of a computation as the sum of program size and running time.

Since the program size of an algorithm is fixed for all input lengths, the asymptotic complexity of algorithms is determined by the running time (up to additive constant).

A decision problem $\{f_n\}$ has *worst-case complexity* $t(n)$ if there is a computation A of complexity

$t(n)$ on inputs of length n such that $A(x) = f_n(x)$ for every input x of length n. We say $\{f_n\}$ is *has average-case complexity* $t(n)$ with error $\delta(n)$ if there is a computation A that runs in time on inputs of length n

$$\Pr_{x \sim \{0,1\}^n}[A(x) = f_n(x)] \geq 1 - \delta(n).$$

Thus an average-case computation is allowed to fail on a $\delta(n)$-fraction of inputs of length n.

Theorem 1. *For every $\varepsilon > 0$ there is an $\varepsilon' > 0$ so that the following holds. Suppose there exists a decision problem $\{f_n\}$ that admits worst-case algorithms of time $2^{O(n)}$, but requires circuit families of worst-case complexity $2^{\varepsilon n}$. Then there exists a decision problem $\{g_n\}$ that admits worst-case algorithms of time $2^{O(n)}$, but requires circuit families of average-case complexity $2^{\varepsilon' n}$ even with error $1/2 - 2^{-\varepsilon' n}$.*

Let me try to explain this in words. Say you have a decision problem $\{f_n\}$ whose hardness is roughly exponential in the worst case: It requires circuits of exponential complexity, but has algorithms of (slightly larger) exponential complexity. For all we know, problems like boolean formula satisfiability (SAT) are exactly like that. While this theorem doesn't tell us much about the average-case hardness of SAT, it says there is a related problem $\{g_n\}$ that has not only similar hardness in the worst case, but is in also very hard in the average case: No circuit family of size $2^{\varepsilon' n}$ can give the correct answer on more than $1/2 + 2^{-\varepsilon' n}$ fraction of inputs. Notice that we can always get the correct answer on half the inputs: Either the circuit that always outputs 0 or the circuit that always outputs 1 is bound to be correct at least half the time. The Impagliazzo-Wigderson theorem says that we cannot do much better for $\{g_n\}$.

2 Codes and average-case hardness

We show an elegant proof of the Impagliazzo-Wigderson theorem due to Sudan, Trevisan, and Vadhan. Their proof is based on an interesting connection between average-case hardness and error-correcting codes. Let's pretend that we have somehow managed to construct $\{g_n\}$ from $\{f_n\}$ and now we need to argue that if $\{f_n\}$ is worst-case hard for circuits, then $\{g_n\}$ is average-case hard for circuits. It is easier to work with the contrapositive of this statement: We need to argue that if there is a circuit family $\{C_n \colon \{0,1\}^n \to \{0,1\}\}$ computing g_n in time $t(n) = 2^{\varepsilon' n}$ with error $1/2 + 1/t(n)$, then there is another circuit family that computes f_n without any errors.

This suggests a connection between converting average-case algorithms into worst-case algorithms and error correction in codes. To make this connection explicit, we look at the truth-table representation of functions from $\{0,1\}^n$ to $\{0,1\}$. Every such function can be specified as a collection of 2^n values, one for every input, so we can view such a function as a string of length 2^n. Then the statement "C_n computes g_n with error $\delta(n) = 1/2 + 1/t(n)$" just says that (the truth table of) C_n is at relative distance $\delta(n)$ from (the truth table of) g_n. So if we think of g_n as a codeword obtained from the "message" $f_{k(n)}$, then the truth-table of the circuit C_n would be a corrupted codeword. If we managed to "decode" this codeword, we would get back a new function $D_{k(n)}$ that equals $f_{k(n)}$. This $D_{k(n)}$ is the desired circuit for $f_{k(n)}$. We only need to arrange things so that the complexity of this circuit is not too large.

Let's set up this construction in the coding-theoretic framework and see what kind of code may be helpful here. We want a family of error-correcting codes $\{\mathcal{C}_N\}$, where $\mathcal{C}_N$ maps "messages" $f_{k(n)}$ of length $K(n) = 2^{k(n)}$ into codewords g_n of length $N(n) = 2^n$ so that:

- The encoding $\mathcal{C}_N$ is computable in time polynomial in $N = 2^n$: To compute $g_n(x)$, we need to compute a codeword that depends on possibly all $2^{k(n)}$ values $f_{k(n)}$. If $\mathcal{C}_N$ is computable in polynomial time, then $g_n(x)$ can be computed in time $2^{k(n)+O(n)} = 2^{O(n)}$.
- What distance should the code $\mathcal{C}_N$ have? Since the "corrupted codeword" C_n can differ from g_n in as many as $\delta(n)$-fraction of inputs, it appears we need $\mathcal{C}_N$ to have relative distance at least $2\delta(n)$. However, this is impossible when $\delta(n) = 1/2 + 1/t(n)$, which is close to $1/2$. Let us ignore this issue for now and try to make $\mathcal{C}_N$ have as large distance as possible.
- How can we make $D_{k(n)}$ have circuit complexity $t(n) \leq 2^{\varepsilon' n}$? To obtain $D_{k(n)}$ we must decode the "corrupted codeword" C_n. But C_n has length $N = 2^n$ so merely inspecting the codeword will take too much time. But we do not need to compute all of $D_{k(n)}$ at once; we only need to find the value $D_{k(n)}(x)$ at the specific input x we are given. In coding-theoretic terms, what we want is not to decode the whole message, but merely to find the value of a specific codeword entry. This leads us to the notion of local decoding.

3 Local decoding

Let $\mathcal{C}$ be an $[N, K, D]$ code. An *ℓ-local decoding algorithm* for $\mathcal{C}$ for t errors is a randomized algorithm that on input $i \in [K]$ and given access to any corrupted codeword $y \in \{0,1\}^N$ with at most t corruptions, inspects at most ℓ positions of y and outputs the value of the closest codeword to y at position i with probability $2/3$. (The exact value of this probability won't matter too much for us as long as it is bounded away from $1/2$.)

To illustrate the concept let's give a local decoding algorithm for the Hadamard code. Recall that the Hadamard code is an $[N, K = \log N, N/2]$ code where the encoding Had_a of $a \in \{0,1\}^K$ consists of the values $\langle a, x\rangle$ for all $x \in \{0,1\}^K$. We can view Had_a as the truth-table of the linear function $Had_a(x) = \langle a, x\rangle$.

Suppose we have a corrupted codeword within distance $N/8$ from the code. We view this corrupted codeword as the truth-table of a function $f\colon \{0,1\}^k \to \{0,1\}$. Then the fact that f is within distance $N/8$ of the code says that

$$\Pr_{x\sim\{0,1\}^k}[f(x) \neq Had_a(x)] \leq N/8 \quad \text{for some } a \in \{0,1\}^k.$$

To decode the i-th bit of a, we choose a random $x \sim \{0,1\}^k$ and inspect the values $f(x)$ and $f(x + e_i)$, where e_i is the vector that is all zero except at position i. If there were no errors, we would get $f(x) = \langle a, x\rangle$ and $f(x + e_i) = \langle a, x + e_i\rangle$; adding these two values yields $a_i = \langle a, e_i\rangle$. If there are few errors in f, this is still usually going to work, unless we are unlucky in our choice of x:

$$\begin{aligned}\Pr_{x\sim\{0,1\}^k}[f(x) + f(x + e_i) \neq a_i] &\leq \Pr_{x\sim\{0,1\}^k}[f(x) \neq \langle a, x\rangle] + \Pr_{x\sim\{0,1\}^k}[f(x + e_i) \neq \langle a, x_i\rangle] \\ &\leq 1/8 + 1/8 < 2/3.\end{aligned}$$

So the Hadamard code is 2-locally decodable for $N/8$ errors by the following simple algorithm: On input i and given access to a corrupted codeword f, choose a random $x \sim \{0,1\}^k$ and output $f(x) + f(x + e_i)$.

However the rate of the Hadamard code is too low for our purposes. Recall that we managed to get codes of better rate by concatenating the Reed-Solomon code and the Hadamard code. The

Reed-Solomon code gave us good rate, while we used the Hadamard code to reduce the alphabet size.

This suggests looking at local decoding algorithms for the Reed-Solomon code. However the Reed-Solomon code does not have good local decoding properties: If we allow for t errors, there can be no t-local decoding algorithm. The intuitive reason is that if the t errors are random, then any t positions of the corrupted codeword will look random and so they won't give any information about the codeword.

4 Reed-Muller codes

In the Reed-Solomon code, the message is a univariate polynomial and the codeword consists of evaluations of this polynomial at various points. In the Hadamard code, we can view the message $a \in \mathbb{F}_2^K$ as a linear function $\ell_a(x) = \langle a, x \rangle$. Then the codeword consists of the evaluations of this linear function at all points of $\mathbb{F}_2^K$. This offers a unifying perspective on the two codes: In the Reed-Solomon code, we evaluate a polynomial of fairly large degree but in one variable, while in the Hadamard code we evaluate a polynomial of as low degree as possible (a linear function) but in many variable.

There is a natural spectrum of intermediate alternatives, known as Reed-Muller codes. In the (m, d) Reed Muller code over finite field $\mathbb{F}$, the messages are m-variate polynomials $p(x_1, \ldots, x_m)$ of total degree d and their encoding consists of the evaluation of p over all points in $\mathbb{F}^m$. The number of coefficient in an m-variate, degree k polynomial is $\binom{m+d}{d}$. The distance of this code is given by the following important lemma:

Lemma 2 (Schwarz-Zippel)**.** *Let p be a degree-d multivariate polynomial over $\mathbb{F}$. Then*

$$\Pr_{x \sim \mathbb{F}^m}[p(x) = 0] \leq d/|\mathbb{F}|.$$

Here is a fake "proof" of this lemma. If p is univariate, then the statement follows from the fact that p has at most d zeros. Otherwise look at the restriction of p on a ℓ in $\mathbb{F}^m$. Such a line can be parametrized as $\ell(t) = a + bt$, so $p(\ell(t))$ is a degree d *univariate* polynomial in t. By the univariate case, $\Pr_{t \sim \mathbb{F}}[p(\ell(t)) = 0] \leq d/|\mathbb{F}|$. Averaging over all lines, we get that

$$\Pr_{\ell, t}[p(\ell(t)) = 0] \leq d/|\mathbb{F}|. \tag{1}$$

but a random point t on a line $\ell(t)$ is just a random point in $\mathbb{F}^m$, so $\Pr[p(x) = 0] \leq d/|\mathbb{F}|$.

The problem with this proof is that p could vanish completely along some lines, in which case (1) is not true. I think it can be fixed with some additional ideas if instead of looking at lines, we consider $(m-1)$-dimensional affine subspaces of $\mathbb{F}^m$.

In any case, this idea inspires the following strategy for decoding Reed-Muller codewords. Say we are given a corrupted version f of some codeword p. Take a random line $\ell(t)$ and inspect the values of f along the line $\ell(t)$. On average, we expect $f(\ell(t))$ to agree with $p(\ell(t))$ about the same fraction of times that f agrees with p. But $p(\ell(t))$ is a univariate polynomial in t, so we may be able to recover $p(\ell(t))$ from $f(\ell(t))$ using the Reed-Solomon decoding algorithm. If we can recover p along sufficiently many such random lines, we may be able to piece this information together and recover p itself.

A careful implementation of these ideas gives not only a decoding algorithm, but a local reconstruction algorithm. The difference between decoding and reconstruction is somewhat technical: While the objective of decoding is to recover the message from a corrupted codeword, the objective of reconstruction is to recover the original codeword.

5 Local reconstruction of Reed-Muller codes

Let f be a corrupted codeword of the Reed-Muller code with above parameters and p its closest polynomial. Suppose f and p are at relative distance δ (i.e. they disagree on $\delta|\mathbb{F}|^m$ points). Let $B \subseteq \mathbb{F}^m$ denote the set of points where they disagree.

We are interested in the fraction of points on which f and p disagree along a random line ℓ, namely the size of the intersection between ℓ and B. Let X_t be an indicator random variable for the event that the t-th point along ℓ is in B. Since the t-th point on a random line is a random point in $\mathbb{F}^m$, we have $\mathrm{E}[X_t] = \delta$. By linearity of expectation, the expected intersection size of ℓ and B is $\delta|\mathbb{F}| \leq d$.

We now want to say that *most* lines have this intersection size with B. We can apply Markov's inequality, but it is possible to do better. Notice that any pair of points on a random line is independent, so the random variables X_t are pairwise independent. So we can apply Chebyshev's inequality:

Lemma 3. *Let $X_1, \ldots, X_q$ be pairwise independent $\{0,1\}$-valued random variables with $\mathrm{E}[X_i] = \delta$. Then*

$$\Pr[X_1 + \cdots + X_q \geq \delta q + \lambda\sqrt{\delta q}] \leq 1/\lambda^2$$

for every $\lambda \geq 1$.

Specifically for $\lambda = \sqrt{4d}$, we get that at most a $1/4d$ fraction of random lines have intersection size with B exceeding $\delta|\mathbb{F}| + \sqrt{4d\delta|\mathbb{F}|} = 3d$. The decoding algorithm for the Reed-Solomon code works for up to $(|\mathbb{F}| - d)/2$ errors. If we choose $\mathbb{F}$ to have size at least $8d$, then $(|\mathbb{F}| - d)/2 > 3d$, and we can handle disagreements as large as $\delta = 7/16$.

How we can reconstruct $p(x)$ at an arbitrary point x of our choice? The trick is to choose a random line ℓ passing through x and choose any $d+1$ points $x_1, \ldots, x_{d+1}$ along this line. For each of these points x_i, choose a random line ℓ_i through x_i. Since x_i is a random point, ℓ_i will be a random line so p can be reconstructed along ℓ_i with probability $1/4d$. In particular, this gives the value $p(x_i)$. By a union bound, the probability that any of the values for $p(x_i)$ obtained in this reconstruction is incorrect is at most $(d+1)/4d < 1/3$, so with probability $2/3$ we get hold of the values $p(x_1), \ldots, p(x_{d+1})$. But $p(\ell(t))$ is a degree k polynomial, so knowing its values at $d+1$ points allows us to obtain $p(x)$.

To summarize, let's describe the algorithm one more time:

Algorithm $RMLocal(f, x)$ where f is a function from $\mathbb{F}^m$ to $\mathbb{F}$ and $x \in \mathbb{F}^m$:
1. Choose a random line ℓ through x so that $x = \ell(0)$ and points $x_1, \ldots, x_{d+1}$ along this line.
2. For every $1 \leq i \leq d+1$:
3. Choose a random line ℓ_i through x_i so that $x_i = \ell_i(0)$.
4. Apply the Reed-Solomon decoding algorithm to find the closest

5. degree d polynomial $p_i(t)$ for $f(\ell_i(t))$.
6. Interpolate the unique degree d polynomial $p(\ell(t))$ such that $p(x_i) = p_i(0)$.
7. Output the value of $p(\ell(0))$.

Here is a summary of what we proved about this algorithm.

Theorem 4. *For $|\mathbb{F}| \geq 8d$ and d sufficiently large, algorithm RMLocal is a $O(d|\mathbb{F}|)$-local decoding algorithm for the (m, d) Reed-Muller code for up to $1/2 - 1/2d$ fraction of errors.*

In fact this algorithm is an even stronger decoding algorithm than what is required in the definition; you can check that for every f within relative distance $7/16$ of some codeword p,

$$\Pr[RMLocal(f, x) = p(x) \text{ for all } x \in \mathbb{F}^m] \geq 2/3.$$

6 Proof sketch of the Impagliazzo-Wigderson theorem

To prove the Impagliazzo-Wigderson theorem, we will construct $\mathcal{C}$ by concatenating a suitable Reed-Muller code with the Hadamard code. Recall that the Reed-Muller code is supposed to give us reasonably good rate, but also local decoding. However, it is not possible to get such a code over binary alphabet, and for this reason we concatenate with the Hadamard code.

We saw that the (m, d) Reed-Muller code maps messages of length $K = \binom{d+m}{m} \geq (d/m)^m$ into codewords of length $N = |\mathbb{F}|^m$ and has a $O(d|\mathbb{F}|)$-local decoder up to relative distance $7/16$, for $|\mathbb{F}| \geq 8d$. For $d = (\log K)^2$, $m = O(\log K / \log\log K)$, and the block length is at most $N \leq K^2$.

The concatenation of the (m, d) Reed-Muller code and the Hadamard code maps messages of length at least $K = 2^k$ into codewords of length at most $N|\mathbb{F}| = 2^{O(k)}$. What about decoding? The local decoder for the Reed-Muller code works up to relative distance $7/8$, and we saw a local decoder for the Hadamard code for relative distance $1/8$. Composing these two, we get an $O(\text{poly}\log K) = \text{poly}(n)$ local decoder for the concatenated code up to relative distance $(1/8)(7/16) = 7/128$.

This falls well short of what we wanted to do, which was to decode up to relative distance $1/2 + 2^{-\varepsilon' n}$. We observed that unique decoding is impossible at this distance, so we have to turn to list-decoding. Fortunately, the local decoding algorithm *RMLocal* for the Reed-Solomon code can be converted into a local list-decoding algorithm by replacing the Reed-Solomon decoding algorithm in step 3 with Sudan's list-decoding algorithm. This local list-decoding algorithm works even at distances close to 1. For the inner Hadamard code, it turns out that finding all the codewords in the list by brute-force is sufficient for our purposes. With some modification in the choice of parameters and a few additional ideas, we can prove Therorem 1 along these lines.