A survey on hardness magnification

Sanjana Das

May 6, 2024

Contents

§1 Introduction

Hardness magnification is the surprising phenomenon where weak-looking lower bounds against certain computational problems — e.g., bounds of the form $n^{1+\varepsilon}$ for any $\varepsilon > 0$ (for problems we believe should satisfy much better bounds) — would imply much stronger-looking lower bounds — e.g., separations such as $\mathsf{NP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$. Several theorems along these lines have been proven in recent years. In this survey, we explain the proofs of some subset of these theorems.

§1.1 Overview of results

In this subsection, we describe the results we'll prove in this survey. We'll use the following notation:

- For a function $t\colon \mathbb{N} \to \mathbb{N}$, we use $\mathsf{Circuit}[t(n)]$ to denote the class of problems which can be solved by a circuit family $\{C_n\}$ (where C_n is the circuit corresponding to inputs of length n) with $\mathsf{size}(C_n) = O(t(n))$. (We work with circuits using **AND** and **OR** gates of fan-in 2 and **NOT** gates of fan-in 1; we measure the size of a circuit by the number of gates.) We use $\mathsf{Circuit}[\mathsf{poly}]$ to denote $\bigcup_k \mathsf{Circuit}[n^k]$.
- We define $\mathsf{Formula}[t(n)]$ and $\mathsf{Formula}[\mathsf{poly}]$ similarly for Boolean formulas — we work with De Morgan formulas consisting of **AND** and **OR** gates of fan-in 2, where the leaves are all literals (i.e., x_i or $\overline{x_i}$) or constants, and we measure the size of a formula by its number of leaves. For example,
$$x_1 \vee ((\overline{x_2} \wedge 0) \vee (x_3 \wedge \overline{x_1}))$$
is a Boolean formula of size 5.
- We'll also work with a slightly extended formula model — we say a **XOR**-formula is a formula again consisting of **AND** and **OR** gates of fan-in 2, but where the leaves are allowed to be arbitrary parity functions (i.e., functions of the form $x_{i_1} \oplus \cdots \oplus x_{i_k} \oplus b$ for some indices $i_1, \ldots, i_k \in [n]$ and some bit $b \in \{0, 1\}$). We still measure the size of such a formula by its number of leaves — for example,
$$(x_1 \oplus x_2) \wedge ((x_1 \oplus x_5 \oplus x_8 \oplus x_{10}) \vee (x_2 \oplus x_4 \oplus 1))$$
is a **XOR**-formula of size 3 (the leaves are $x_1 \oplus x_2$, $x_1 \oplus x_5 \oplus x_8 \oplus x_{10}$, and $x_2 \oplus x_4 \oplus 1$). We define the classes $\mathsf{Formula\text{-}XOR}[t(n)]$ and $\mathsf{Formula\text{-}XOR}[\mathsf{poly}]$ similarly for this formula model.

  Note that $\mathsf{Formula\text{-}XOR}[t(n)] \subseteq \mathsf{Formula}[t(n) \cdot n^2]$, since any parity function on n bits can be computed by a De Morgan formula of size $O(n^2)$. But it turns out that several hardness magnification results work especially nicely with this formula model, which is why we use it.

Oliveira and Santhanam (who coined the term 'hardness magnification') first prove several hardness magnification theorems in [OS18], especially for variants of the 'meta-computational' problems MCSP and MKtP. Roughly speaking, MCSP is the problem of determining whether a function can be computed by a small circuit, and MKtP is the problem of determining whether a string has low Kt-complexity, i.e., it can be computed reasonably quickly by a Turing machine with a short description. (The precise definitions of these problems are given in Section 2.)

For example, they consider an 'average-case' variant of MCSP, denoted by $(1, 1-\delta)$-$\mathsf{MCSP}[s]$ — the promise problem where we're given a function $f\colon \{0,1\}^m \to \{0,1\}$, presented as a truth table of length $n = 2^m$, and we wish to distinguish between the case where f can be computed by a circuit of size s and the case where it cannot even be $(1-\delta)$-approximated by a circuit of size s (where δ and s may depend on n). They prove several hardness magnification theorems for this problem; here is one such statement, whose proof we will discuss in Subsection 3.1.

Theorem 1.1 ([OS18, Theorem 17])

Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ with $(1, 1-n^{-\beta})$-$\mathsf{MCSP}[n^\beta] \not\in \mathsf{Formula}[n^{1+\varepsilon}]$. Then $\mathsf{NP} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

In a later paper, Oliveira, Pich, and Santhanam [OPS19] prove a hardness magnification theorem for a 'gap' variant of MCSP — Gap-MCSP$[s_1, s_2]$ is the promise problem where we want to distinguish between the case where f can be (exactly) computed by a circuit of size at most s_1 and the case where it cannot be computed by a circuit of size at most s_2 (for $s_1 \leq s_2$). We'll explain this proof in Section 4.

Theorem 1.2 ([OPS19, Theorem 4])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ with $\mathsf{Gap\text{-}MCSP}[n^\beta, n^\beta \log n] \notin \mathsf{Circuit}[n^{1+\varepsilon}]$. Then $\mathsf{NP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$.

Oliveira and Santhanam [OS18] and Oliveira, Pich, and Santhanam [OPS19] also proved several hardness magnification theorems for a gap variant of MKtP — Gap-MKtP$[p_1, p_2]$ is the promise problem where we want to distinguish between the cases $\mathsf{Kt}(x) \leq p_1$ and $\mathsf{Kt}(x) > p_2$. (We'll define Kt-complexity in Subsection 2.2.) One such theorem, which we'll explain in Subsection 3.2, is the following.

Theorem 1.3 ([OPS19, Theorem 1])
There is an absolute constant $c > 0$ such that the following holds: suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ with $\mathsf{Gap\text{-}MKtP}[n^\beta, n^\beta + c \log n] \notin \mathsf{Formula\text{-}XOR}[n^{1+\varepsilon}]$. Then $\mathsf{EXP} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

Remark 1.4. For several of the magnification theorems we'll discuss, including Theorem 1.3, the authors considered a variety of computational models — for example, circuits, formulas over various different bases, AC^0 circuits, and branching programs (where the bound needed for magnification typically depends on the model).

For simplicity, throughout this survey we'll focus on formula models for all such results. Specifically, we state Theorem 1.1 just in the case of Formula, and for the remainder of such results (Theorems 1.3, 1.9, 1.11, and 1.12), we state them just in the case of Formula-XOR. (For these theorems, we can also prove magnification results with ordinary formulas, but from lower bounds of $n^{3+\varepsilon}$ rather than $n^{1+\varepsilon}$; one reason Oliveira, Pich, and Santhanam [OPS19] consider Formula-XOR is that there exist problems for which we know how to prove $n^{2-\varepsilon}$ lower bounds in this model (e.g., see [Tal16]), providing some hope of actually being able to use such magnification theorems to prove strong lower bounds.)

McKay, Murray, and Williams [MMW19] then consider the *exact* versions of MCSP and MKtP, rather than 'average-case' or 'gap' variants. In fact, they even consider *search* versions of MCSP and MKtP, where we need to *find* the small circuit or Turing machine (respectively) that computes the given input, rather than just checking whether one *exists*. (We'd expect that these changes make the problems harder, therefore making the hypotheses needed for magnification easier to prove.) They prove several magnification theorems for Circuit models; we'll explain the proofs of the following ones in Subsections 5.1 and 5.3, respectively.

Theorem 1.5 ([MMW19, Theorem 1.4])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that $\mathsf{Search\text{-}MCSP}[n^\beta]$ does not have circuits of size $n^{1+\varepsilon}$ and depth n^ε. Then $\mathsf{NP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$.

Theorem 1.6 ([MMW19, Theorem 1.7])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that $\mathsf{Search\text{-}MKtP}[n^\beta]$ does not have circuits of size $n^{1+\varepsilon}$ and depth n^ε. Then $\mathsf{EXP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$.

They also prove magnification theorems from lower bounds against low-space (deterministic) streaming algorithms; we'll explain the proof of the following theorem for MCSP in Subsection 5.2.

Theorem 1.7 ([MMW19, Theorem 1.3])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that Search-MCSP$[n^\beta]$ cannot be solved by a n^ε-space streaming algorithm with n^ε update time. Then $\mathsf{P} \neq \mathsf{NP}$.

This statement is perhaps illustrative of why hardness magnification is surprising — we don't believe that MCSP$[n^\beta]$ is even in P, and the statement that it can't be solved by a n^ε-space n^ε-update time streaming algorithm is *much* weaker than this. But even this extremely weak-looking statement is enough to separate P from NP.

So far, all the results we've discussed involve the 'meta-computational' problems MCSP and MKtP (Oliveira and Santhanam also consider a few other problems, such as Vertex-Cover, in [OS18], but most hardness magnification results up to this point considered these meta-computational problems). It's natural to wonder what's special about these problems — can we prove hardness magnification for other, more general, classes of problems? One property of MCSP$[n^\beta]$ and MKtP$[n^\beta]$ is that they are *sparse* — for example, there are only roughly $2^{n^\beta \log n}$ circuits of size n^β, and therefore only this many **YES** instances to MCSP$[n^\beta]$ of length n. It turns out that this sparsity alone is enough to get hardness magnification — Chen, Jin, and Williams [CJW19] prove hardness magnification results for *any* NP language of similar sparsity. In some sense, this is quite surprising — for example, the proofs in [MMW19] use the fact that MCSP and MKtP are in some sense about string compression in a crucial way.

Definition 1.8. We say a language $L \subseteq \{0,1\}^*$ is $\psi(n)$-*sparse* if for all n, the number of **YES** instances to L of length n is at most $\psi(n)$.

Theorem 1.9 ([CJW19, Theorem 1.1])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that there is a 2^{n^β}-sparse language $L \in \mathsf{NP}$ with $L \notin$ Formula-XOR$[n^{1+\varepsilon}]$. Then $\mathsf{NP} \not\subseteq$ Formula$[n^k]$ for all $k \in \mathbb{N}$.

We'll prove this in Subsection 6.1.

(The conclusion here is slightly different from the conclusions of the previous theorems, in that we get that for every k there is some problem in NP that does not have n^k-sized formulas, rather than that there is some problem in NP that does not have n^k-sized formulas for any k — but this would still be a very strong result. We'll touch on the reasons for the difference in Subsubsection 1.2.4.)

Chen, Jin, and Williams [CJW19] also prove several magnification results for such languages from bounds against uniform models with a small amount of advice. One such result, which we'll prove in Subsection 6.2, is the following.

Theorem 1.10 ([CJW19, Theorem 1.2])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that there is a 2^{n^β}-sparse language $L \in \mathsf{NP}$ that cannot be computed by a $n^{1+\varepsilon}$-time n^ε-space deterministic algorithm with n^ε bits of advice. Then $\mathsf{NP} \not\subseteq$ Circuit$[n^k]$ for all $k \in \mathbb{N}$.

Chen, Jin, and Williams [CJW19] also use the ideas behind Theorem 1.9 to extend magnification results for Search-MCSP and Search-MKtP (such as Theorems 1.5 and 1.6) to many more computational models. For example, they prove the following theorems, which we'll explain in Subsections 6.4 and 6.3 (respectively).

Theorem 1.11 ([CJW19, Theorem 1.6])

Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ for which $\mathsf{Search\text{-}MCSP}[n^\beta] \notin \mathsf{Formula\text{-}XOR}[n^{1+\varepsilon}]$. Then $\mathsf{PSPACE} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

In fact, [CJW19] proves that we can replace PSPACE with either $\oplus\mathsf{P}$ or PP; this strengthening requires additional technical ingredients, so we will only prove it for PSPACE for simplicity.

Theorem 1.12 ([CJW19, Theorem 1.6])

Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ for which $\mathsf{Search\text{-}MKtP}[n^\beta] \notin \mathsf{Formula\text{-}XOR}[n^{1+\varepsilon}]$. Then $\mathsf{EXP} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

These theorems in some sense combine the best features of Theorems 1.1 and 1.3 (which apply to more restricted models — in which we have a better chance of being able to prove lower bounds — but for more relaxed variants of MCSP and MKtP for which it's probably harder to prove lower bounds) and Theorems 1.5 and 1.6 (which apply to the harder problems Search-MCSP and Search-MKtP, but require bounds on the more powerful model of Circuit).

§1.2 Overview of ideas

Before we get into the details of the proofs, we'll explain the main ideas. All these theorems are proven by contrapositive — we assume that the strong lower bound in the theorem statement doesn't hold (e.g., $\mathsf{NP} \subseteq \mathsf{Formula}[\mathsf{poly}]$), and we use this assumption to obtain an *extremely* efficient algorithm (in the relevant model of computation) for the problem at hand, contradicting the weak lower bound in the theorem hypothesis.

At a very high level, the idea behind obtaining this extremely efficient algorithm is to reduce to a problem on a much smaller input. Specifically, we'll show *unconditionally* that there exists an extremely efficient algorithm for our problem that makes queries to an oracle solving some auxiliary problem, where the length of each query is tiny compared to n (e.g., $\mathsf{poly}(n^\beta)$). Then we'll show that under the given assumption, we can implement this oracle with a *decently* efficient algorithm (for example, if the oracle problem is in NP, then the assumption $\mathsf{NP} \subseteq \mathsf{Formula}[\mathsf{poly}]$ allows us to implement the oracle with a polynomial-sized formula). But since each input to the oracle is tiny, even a decently efficient algorithm for the oracle is *extremely* efficient compared to n (e.g., if we have an oracle query with input length $\mathsf{poly}(n^\beta)$ and we implement the oracle with a polynomial-sized formula, this formula only has size $\mathsf{poly}(\mathsf{poly}(n^\beta)) = \mathsf{poly}(n^\beta)$), so this gives us an extremely efficient algorithm for the original problem.

We'll now discuss the ideas that go into producing such a reduction (or oracle algorithm), which vary across the different theorems.

§1.2.1 Magnification via random sampling

The first idea (from [OS18]), used to prove Theorem 1.1 (regarding $(1, 1-\delta)$-MCSP), is randomly sampling the input — given the truth table of a function f, instead of trying to figure out whether there's a small circuit computing the *entire* truth table of f, we'll imagine taking a small random sample of input-output pairs $(z_1, f(z_1)), \ldots, (z_t, f(z_t))$ and trying to figure out whether there's a small circuit computing just these input-output pairs. This is a problem on a much shorter input length, and it's what we'll use as our oracle — note that it's in NP, since we can solve it by nondeterministically guessing a circuit and then computing it on all the given inputs.

If f is a **YES** instance of $(1, 1-\delta)$-MCSP, then there's a small circuit C that computes f, and of course the same circuit computes any sample of input-output pairs from f. On the other hand, if f is a **NO** instance, then each small circuit C fails to compute f on at least a δ-fraction of inputs, so with probability at least $(1-\delta)^t$ it fails to compute f on one of the input-output pairs in the sample. And since there aren't too many small circuits, if we choose t appropriately, then we can union-bound over all such circuits to get that with high probability, *every* small circuit fails to compute f on the sample.

Right now, this gives us a randomized oracle algorithm for $(1, 1-\delta)$-MCSP$[s]$, rather than a formula. But to turn it into a formula, we can first imagine performing n independent trials of this form, and finding the **AND** of their results. This makes our error probability (over the randomness used to produce the sample) so small that we can union-bound over all inputs to find a choice of samples that works for *all* inputs, and then hardcode these samples into our formula — so in the end, our formula is a **AND** of n short oracle calls (where to obtain the inputs $(z_1, f(z_1))$, …, $(z_t, f(z_t))$ to each oracle call, we hardcode each z_i into the formula, and each $f(z_i)$ is a bit of the given input).

The idea of random sampling, together with one more ingredient, can be used to prove Theorem 1.3 (regarding Gap-MKtP) as well. Given an input x, we can't randomly sample from x directly — unlike with $(1, 1-\delta)$-MCSP, it's no longer true that all **YES** and **NO** instances of Gap-MKtP differ in a constant fraction of indices (which was the crucial property that allowed the above proof to work for $(1, 1-\delta)$-MCSP). But we can get such a property by using *error-correcting codes* — if x has low Kt-complexity and E is an efficient error-correcting code, then $E(x)$ must *also* have low Kt-complexity (since a Turing machine can produce $E(x)$ by first producing x and then encoding it). Meanwhile, if x has *high* Kt-complexity, then $E(x)$ can't even be *close* to a string y with low Kt-complexity (because we could produce x by first producing y and then decoding it to x). (This is what we need the gap for.)

Now in order to decide Gap-MKtP on x, we can imagine first computing $E(x)$, and then taking a random sample of indices $(d_1, E(x)_{d_1})$, …, $(d_t, E(x)_{d_t})$ and checking whether this sample is consistent with a string of low Kt-complexity — this is again a problem on a much smaller input size (and it's in EXP, since we can enumerate over low Kt-complexity strings by enumerating over short Turing machine descriptions and running each for the appropriate amount of time), and the rest of the proof is very similar to that of Theorem 1.1. The place where working with the model Formula-XOR becomes convenient is that if E is also *linear*, then each bit $E(x)_d$ is a fixed parity function of the bits of x, which means we don't have to do any computation to obtain $E(x)_d$ (for any fixed d) — we can simply take it to be a leaf of the formula.

§1.2.2 Magnification via anticheckers

The next idea (from [OPS19]), used to prove Theorem 1.2 (regarding Gap-MCSP), is that of anticheckers. Informally, if a function $f\colon \{0,1\}^m \to \{0,1\}$ can't be computed by a circuit of size s, then an *antichecker* for f against circuits of size s is a collection of strings $\mathcal{S} \subseteq \{0,1\}^m$ such that every circuit of size s in fact fails to compute f on one of those strings.

Using a combinatorial argument, it's possible to show that if f can't be computed by a circuit of size sm, then there exists a 'small' antichecker for f against circuits of size s. (Here s is n^β and m is $\log n$, and this is what we need the gap for.) Furthermore, a 'greedy' construction works — specifically, we'll imagine constructing $\mathcal{S}$ one string at a time, and the combinatorial argument will show that at every step, there exists some string z such that adding z to $\mathcal{S}$ would substantially shrink the number of 'surviving' circuits (i.e., small circuits that correctly compute f on $\mathcal{S}$).

In order to prove Theorem 1.2, we'll show that we can implement this construction using an efficient circuit that makes short queries to a NP oracle. The main idea behind this is that if we've placed strings z_1, …, z_t into $\mathcal{S}$ so far, then finding a good string z to choose next in the greedy combinatorial construction amounts to going through all possible strings z, counting the number of surviving circuits that would result from adding each to $\mathcal{S}$, and choosing the one that minimizes this number. But this is an approximate counting problem — we really only need to *approximate* this number of surviving circuits, and the quantity we're trying to count is essentially the number of witnesses to a NP-verifier (specifically, the verifier checking that a circuit C (which is the witness) computes f on $\mathcal{S} \cup \{z\}$). And approximately counting such a quantity can be done with a NP oracle (where the input to the oracle is the input-output pairs of f corresponding to $\mathcal{S} \cup \{z\}$, which does have short length).

This allows us to turn the combinatorial proof of the *existence* of an antichecker into an efficient circuit (with a NP oracle) for *constructing* one. And if this circuit successfully constructs an antichecker, we immediately know f is not a **YES** instance to Gap-MCSP (if there exists an antichecker for f, then f certainly doesn't have a circuit of size s), while if it fails, then we know f is not a **NO** instance (since the circuit is guaranteed to work if f doesn't have a circuit of size sm).

§1.2.3 Magnification via compression

The next idea (from [MMW19]), used to prove Theorems 1.5, 1.6, and 1.7 (regarding Search-MCSP and Search-MKtP), is that MCSP and MKtP are essentially problems about string compression — for example, in MCSP$[n^\beta]$, we're given a truth table of length n and we're trying to figure out whether we can describe it by a circuit of the much smaller size n^β — and we can use this to compress the input we feed to our oracle.

For now, we'll explain the idea in the context of Theorem 1.7 (on an efficient streaming algorithm for Search-MCSP); the remaining theorems use the same idea, but implemented by a circuit rather than a streaming algorithm. (For simplicity, we'll implement this idea in a way that's a bit less efficient than the actual argument in [MMW19].)

Imagine that we're a streaming algorithm being fed one entry in the truth table of f at a time (in order), and we're trying to space-efficiently figure out whether there's a small circuit computing f. We certainly don't have enough space to store all the entries of the truth table we've seen so far *directly*. Instead, we store them *using a small circuit* — if there is a small circuit computing f, then there's certainly a small circuit computing the portion of the truth table we've seen so far. More precisely, once we've read the entries of f corresponding to the interval of strings $[0^m, z] \subseteq \{0,1\}^m$ (by 'interval' we mean the lexicographical order on $\{0,1\}^m$, which we assume is the order our truth table is presented in), we store z and the lexicographically minimal small circuit C that computes f correctly on this interval (or that no such circuit exists). Then when we read the entry of f corresponding to the next string z', we use an oracle to 'merge' this new information into C — specifically, we give the oracle z, C, z', and $f(z')$ (this input has short length), and it gives us a small circuit C' that matches C on $[0^m, z]$ and matches $f(z')$ on z' (or tells us that no such circuit exists, in which case we immediately know f is a **NO** instance). Then once we've read through the entire input, we automatically have a small circuit for f, which means we've solved Search-MCSP.

Unlike in the previous arguments, this oracle isn't necessarily in NP, but we *can* show that it's in the polynomial hierarchy — and if P = NP then the polynomial hierarchy collapses, so it's still true that the assumption P = NP (in the contrapositive of Theorem 1.7) allows us to efficiently implement the oracle.

(To be more precise, for the statement that the oracle is in PH to make sense, we need it to only output a single bit. So we actually define the oracle so that we also give it an index j, and it returns the jth bit of the *lexicographically first* circuit matching our specifications.)

§1.2.4 Magnification via hashing

The final idea we'll discuss (from [CJW19]), used to prove Theorems 1.9 and 1.10 (on hardness magnification for arbitrary sparse NP languages) and Theorems 1.11 and 1.12 (on Search-MCSP and Search-MKtP in more restrictive models), is hashing. Specifically, [CJW19] construct a family of linear hash functions $h_v\colon \{0,1\}^n \to \{0,1\}^t$, with outputs and seeds of length $t = \Theta(n^\beta)$, such that for any 2^{n^β}-sparse language L, there exists some seed v for which h_v hashes all **YES** instances of L to different values.

We'll first explain the idea behind the proof of Theorem 1.9 (which we again implement in a slightly less efficient way than [CJW19] does for simplicity.) We first fix such a seed v (which we'll hardcode into our formula). Then on an input x, imagine that we first compute $h_v(x)$. Then we go through the indices of x one at a time, and for each index $i \in [n]$, we ask our oracle, is there a string $y \in \{0,1\}^n$ such that $h_v(y) = h_v(x)$, $y_i = x_i$, and $y \in L$? If x is itself in L, then the oracle will answer yes for every i (as we can take $y = x$). Meanwhile, if x is not in L, then there's at most one string $y \in L$ whose hash agrees with that of x, and this string y won't agree with x on all indices $i \in [n]$, so the oracle will answer no for some i.

Finally, the fact that h_v is linear means that we can implement this with an efficient **XOR**-formula.

Note that the information we need to give the oracle is essentially just v, $h_v(x)$, i, and x_i, so as usual, the input to the oracle is short (specifically, of length $\Theta(n^\beta)$). Also note that the problem we're asking the oracle to solve is in NP (we could solve it by nondeterministically guessing y, computing $h_v(y)$, and nondeterministically checking that $y \in L$). So the assumption $\mathsf{NP} \subseteq \mathsf{Formula}[n^k]$ allows us to implement this oracle efficiently, as usual.

Remark 1.13. The reason we need a *fixed-polynomial* bound (e.g., $\mathsf{NP} \subseteq \mathsf{Formula}[n^k]$) rather than an *arbitrary-polynomial* bound (e.g., $\mathsf{NP} \subseteq \mathsf{Formula}[\mathsf{poly}]$) is that unlike in all the previous arguments, here our oracle *depends* on L — in particular, different values of β correspond to different oracles — and we need the same upper bound on all of these oracles (or at least, bounds with the same exponent). In contrast, in the previous arguments we used the same oracle for all values of β, so this followed even from arbitrary-polynomial bounds.

For Theorem 1.10, the assumption that $\mathsf{NP} \subseteq \mathsf{Circuit}[n^k]$ allows us to implement the above oracle with a small circuit, and we take our n^ε bits of advice to consist of this circuit together with the good seed v as above. Then our algorithm does the same thing as the above formula — we go through each index $i \in [n]$ one at a time, run the circuit for our oracle on v, $h_v(x)$, i, and x_i, and accept if and only if the circuit accepts on all i. (The only difference is that we now need h_v to be computable with low space, but this is true of the construction of [CJW19].)

The proofs of Theorems 1.11 and 1.12 are also similar to that of Theorem 1.9, but in order to get arbitrary-polynomial rather than fixed-polynomial bounds, we need to set up the oracle so that it doesn't depend on β. In particular, we can't show that the oracle is in NP by simply guessing y and checking that y is in our language anymore — $|y| = n$ is polynomial in the input length $\Theta(n^\beta)$ if we *fix* β, but not if we allow β to go to 0.

For Theorem 1.12 (for MKtP), this modification isn't difficult — we can show that the oracle is in EXP by enumerating over all low Kt-complexity strings y (by enumerating over short descriptions of Turing machines and running them for the appropriate amount of time, as with the oracle for Theorem 1.3).

For Theorem 1.11 (for MCSP), the modification is harder — we can enumerate over all small circuits C in PSPACE, but we can't write down the entire truth table that the circuit computes (which is y in this case), which means we can't directly compute $h_v(y)$. Instead, we note that h_v is linear, so for each index i, there is some string $w_{v,i}$ such that $h_v(y)_i$ can be written as $\langle y, w_{v,i}\rangle$. We think of y as a truth table of a function $\{0,1\}^m \to \{0,1\}$, so y is indexed by strings $z \in \{0,1\}^m$; we think of $w_{v,i}$ as being indexed by such strings too, and we write $w_{v,i}(z)$ to denote the zth coordinate of $w_{v,i}$. Then we can write this as

$$\langle y, w_{v,i}\rangle = \#\{z \in \{0,1\}^m \mid C(z) = w_{v,i}(z) = 1\} \pmod 2.$$

And [CJW19] give a construction of the hash family such that $w_{v,i}(z)$ is efficiently computable given v, i, and z, which lets us compute $\langle y, w_{v,i}\rangle$ in PSPACE (by enumerating over all z, computing $C(z)$ and $w_{v,i}(z)$ for each, and keeping a running count of the above quantity mod 2).

(In both cases, to solve the *search* versions, we also need the oracle to output the good Turing machine or circuit that it finds (rather than just accepting if it finds one), but this can be done in a similar way to the oracles for Theorems 1.5 and 1.6 — we also give the oracle an index j, and we instead ask it for the jth bit of the lexicographically smallest Turing machine or circuit meeting our specifications.)

§2 Preliminaries

Throughout this paper, we use log to denote the base-2 logarithm. We also omit floors and ceilings for notational simplicity. For $x, y \in \{0,1\}^n$, we use $\langle x, y\rangle$ to mean $\sum_{i=1}^n x_i y_i \pmod 2$. We assume a RAM model when we describe algorithms unless otherwise specified.

As described in Subsection 1.2, the proofs of most of the theorems in Subsection 1.1 will involve (unconditionally) constructing oracle circuits or formulas for the relevant problems, where the length of each oracle query is tiny. To formalize this, we'll always work with oracles that output a single bit. We say an $\mathcal{A}$-oracle circuit is a circuit whose gates are **AND**, **OR**, **NOT**, and calls to $\mathcal{A}$ (on inputs of a fixed length, determined by the number of wires feeding into the gate); we refer to this fixed length as the *fan-in* of the $\mathcal{A}$-oracle gate. We define $\mathcal{A}$-oracle formulas similarly (for both ordinary formulas and **XOR**-formulas); we say that an $\mathcal{A}$-oracle formula makes nonadaptive queries if there are no nested $\mathcal{A}$'s. For example,

$$\mathcal{A}(x_1 \oplus x_2, x_3) \wedge ((x_1 \oplus x_5) \vee \mathcal{A}(x_2 \oplus x_4, x_8, x_{10}))$$

is an $\mathcal{A}$-oracle **XOR**-formula of size 6 (with leaves $x_1 \oplus x_2$, x_3, $x_1 \oplus x_5$, $x_2 \oplus x_4$, x_8, and x_{10}) which makes nonadaptive queries of fan-in at most 3; in contrast,

$$\mathcal{A}(x_1 \oplus x_2, x_3) \wedge \mathcal{A}((x_1 \oplus x_5), \mathcal{A}(x_2 \oplus x_4, x_8, x_{10}))$$

makes adaptive queries, since it contains nested $\mathcal{A}$'s. (Nonadaptiveness matters in the formula setting because to prove the magnification theorems, we'll replace the oracle gates with polynomial-sized formulas; if the formula makes nonadaptive queries then this only blows up its size by a polynomial in the fan-in (which will be tiny by construction), but if it made adaptive queries then we'd have to compose these polynomials, which could potentially result in much greater blowup.)

§2.1 Preliminaries regarding MCSP

In this subsection, we'll define and establish a few conventions regarding MCSP and the variants we'll work with. Roughly speaking, MCSP (which stands for 'minimum circuit size problem') is the problem of determining whether a given function can be computed by a small circuit. The input to MCSP is a function $f\colon \{0,1\}^m \to \{0,1\}$, given as a truth table consisting of $n = 2^m$ bits (the values of f on all strings $\{0,1\}^m$, listed in lexicographical order). When referring to MCSP, we'll always use m to refer to the input length of f and n to refer to the input length to the instance of MCSP (so $n = 2^m$).

We work with a parametrized version of MCSP, where $s\colon \mathbb{N} \to \mathbb{N}$ is a parameter depending on n that defines what we mean by a 'small' circuit. (We think of s as a function of n, not of m — in particular, we use $s(n)$ to denote the size bound for inputs $f\colon \{0,1\}^m \to \{0,1\}$ of length $n = 2^m$ (though we write s rather than $s(n)$ in most places for notational simplicity). This is mainly for consistency with the notation we'll use for MKtP, where there is no analog of m.)

The ordinary version of MCSP is defined as follows.

Definition 2.1 (MCSP[s])

- Input: a function $f\colon \{0,1\}^m \to \{0,1\}$, presented as a truth table of length $n = 2^m$.
- Decide: whether there exists a circuit C (on m inputs) with $\mathsf{size}(C) \le s$ that computes f.

We'll always assume that $s \ge m = \log n$. We typically think of s as n^β for arbitrarily small β; in Subsection 1.1 we stated all magnification theorems in this regime for concreteness, but we'll prove them for general values of s (the results are meaningful for any $s = n^{o(1)}$).

We'll also work with a few variants of MCSP. Specifically, for Theorem 1.1 we consider a promise version of MCSP where we're promised that f is either computable by a small circuit or δ-far fom being computable by a small circuit (in some sense, this is a variant about the 'average-case circuit complexity' of f, while ordinary MCSP is about the 'worst-case circuit complexity').

Definition 2.2 ($(1, 1-\delta)$-MCSP$[s]$)

- INPUT: a function $f\colon \{0,1\}^m \to \{0,1\}$, presented as a truth table of length $n = 2^m$.
- YES CASE: there exists a circuit C (on m inputs) with $\mathsf{size}(C) \leq s$ that computes f.
- NO CASE: for every circuit C (on m inputs) with $\mathsf{size}(C) \leq s$, we have $C(z) \neq f(z)$ for more than a δ-fraction of inputs $z \in \{0,1\}^m$.

Here $\delta\colon \mathbb{N} \to (0,1)$ is another parameter that may depend on n (and again, we view δ as a function of n rather than m).

Remark 2.3. When we say an algorithm (or formula or circuit) decides a promise problem, we mean that it accepts all **YES** instances and rejects all **NO** instances; it doesn't matter what it does on the remaining inputs. By a statement such as $(1, 1-\delta)$-MCSP$[s] \in$ Formula$[n^{1+\varepsilon}]$ (in the statement of Theorem 1.1), we mean that there exists a formula of size $n^{1+\varepsilon}$ that decides the problem in this sense.

For Theorem 1.2, we'll work with a different promise version of MCSP, where we return to 'worst-case circuit complexity' but have a gap between the circuit size bounds for **YES** and **NO** instances.

Definition 2.4 (Gap-MCSP$[s_1, s_2]$)

- INPUT: a function $f\colon \{0,1\}^m \to \{0,1\}$, presented as a truth table of length $n = 2^m$.
- YES CASE: there exists a circuit C (on m inputs) with $\mathsf{size}(C) \leq s_1$ that computes f.
- NO CASE: there does not exist a circuit C (on m inputs) with $\mathsf{size}(C) \leq s_2$ that computes f.

Here s_1 and s_2 are parameters depending on n, with $m \leq s_1 < s_2$.

Throughout our discussion of MCSP, we'll frequently make use of the following fact.

Fact 2.5 — If $s \geq m$, we can encode any m-input circuit C with $\mathsf{size}(C) \leq s$ using a string of $64s \log s$ bits. In particular, the number of such circuits is at most $2^{64s \log s}$.

(The value of 64 is not important; it's just a sufficiently large constant that we use for concreteness.)

Finally, for Theorems 1.5, 1.7, and 1.11, we'll work with a *search* version of MCSP, where instead of asking the algorithm to decide whether a small circuit computing f *exists*, we ask it to *output* such a circuit. For convenience, we use an encoding of circuits such that the all-0's string does not correspond to any circuit, and we use this string to represent that no circuit exists.

Definition 2.6 (Search-MCSP$[s]$)

- INPUT: a function $f\colon \{0,1\}^m \to \{0,1\}$, presented as a truth table of length $n = 2^m$.
- OUTPUT: a string of length $64s \log s$ encoding a circuit C (on m inputs) with $\mathsf{size}(C) \leq s$ that computes f, or the all-0's string if no such C exists.

§2.2 Preliminaries regarding MKtP

In this subsection, we'll define MKtP and the variant we'll work with. Roughly speaking, MKtP (which stands for 'minimum Kt problem') is the problem of determining whether a given string has low Kt-complexity. We'll use the following definition of Kt-complexity (following [CJW19]; some of the other papers we discuss use slightly different definitions, but the details of the definition are not important).

Definition 2.7. For $x \in \{0,1\}^*$, we define $\mathsf{Kt}(x) = \min |\langle M \rangle| + \log \mathsf{time}(M)$, where the minimum is taken over all Turing machines M that, when run on empty input, eventually halt and output x. (We use $|\langle M \rangle|$ to denote the description length of M, and $\mathsf{time}(M)$ to refer to its runtime.)

As with MCSP, we work with a parametrized version of MKtP, where $p \colon \mathbb{N} \to \mathbb{N}$ is a parameter depending on n that defines what we mean by 'low' Kt-complexity. The ordinary version of MKtP is defined as follows.

Definition 2.8 (MKtP$[p]$)

- Input: a string $x \in \{0,1\}^n$.
- Decide: whether $\mathsf{Kt}(x) \leq p$ — i.e., whether there exists a Turing machine M such that $|\langle M \rangle| + \log \mathsf{time}(M) \leq p$ that outputs x.

As with the parameter s in MCSP, we always assume that $p \geq \log n$, and we typically think of p as n^β for arbitrarily small β.

For Theorem 1.3, we'll work with the following promise version of MKtP (where p_1 and p_2 are parameters depending on n, with $\log n \leq p_1 \leq p_2$).

Definition 2.9 (Gap-MKtP$[p_1, p_2]$)

- Input: a string $x \in \{0,1\}^n$.
- Yes case: $\mathsf{Kt}(x) \leq p_1$ — i.e., there is a Turing machine M such that $|\langle M \rangle| + \log \mathsf{time}(M) \leq p_1$ that outputs x.
- No case: $\mathsf{Kt}(x) > p_2$ — i.e., there is no Turing machine M such that $|\langle M \rangle| + \log \mathsf{time}(M) \leq p_2$ that outputs x.

Finally, for Theorems 1.6 and 1.12, we'll work with a search variant of MKtP, where we ask the algorithm to *output* such a Turing machine M (instead of just deciding whether one exists). As with MCSP, we use the all-0's string to denote that no Turing machine exists. We also assume an encoding of Turing machines such that $\langle M \rangle$ always ends with a 1, and when defining Search-MKtP, we pad the output with 0's to have length *exactly* p. (This is because we'll be trying to solve Search-MKtP with circuits or formulas, whose output is always of a fixed size.)

Definition 2.10 (Search-MKtP$[p]$)

- Input: a string $x \in \{0,1\}^n$.
- Output: a string of length exactly p which is the zero-padded description of a Turing machine M such that $|\langle M \rangle| + \log \mathsf{time}(M) \leq p$ and M outputs x, or 0^p if no such Turing machine exists.

§3 Magnification via random sampling

In this section, we'll prove Theorems 1.1 and 1.3, following the outline described in Subsubsection 1.2.1.

§3.1 Magnification for $(1, 1-\delta)$-MCSP

In this subsection, we'll prove the following slight generalization of Theorem 1.1.

Theorem 3.1 ([OS18, Theorem 17])
Suppose that there exist parameters $s\colon \mathbb{N} \to \mathbb{N}$ and $\delta\colon \mathbb{N} \to (0,1)$ with $s(n) \geq \log n$ for all n such that $(1, 1-\delta)\text{-}\mathsf{MCSP}[s] \notin \mathsf{Formula}[n \cdot \mathsf{poly}(s, \delta^{-1})]$. Then $\mathsf{NP} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

Remark 3.2. When we write $(1, 1-\delta)\text{-}\mathsf{MCSP}[s] \notin \mathsf{Formula}[n \cdot \mathsf{poly}(s, \delta^{-1})]$, we mean that there is no $k \in \mathbb{N}$ such that $(1, 1-\delta)\text{-}\mathsf{MCSP}[s]$ has formulas of size $ns(n)^k \delta(n)^{-k}$ on inputs of length n.

Remark 3.3. To see that the more general version of [OS18, Theorem 17] stated here implies the version in Theorem 1.1, assume that the hypothesis of Theorem 3.1 is *not* satisfied, so that there is some $k \in \mathbb{N}$ such that for all $s\colon \mathbb{N} \to \mathbb{N}$ and $\delta\colon \mathbb{N} \to (0,1)$ as described, $(1, 1-\delta)\text{-}\mathsf{MCSP}[s]$ has formulas of size $ns(n)^k \delta(n)^{-k}$. Then given any $\varepsilon > 0$, for all $\beta < \frac{\varepsilon}{2k}$, plugging in $s(n) = n^\beta$ and $\delta(n) = n^{-\beta}$ gives that $(1, 1-n^{-\beta})\text{-}\mathsf{MCSP}[n^\beta]$ has formulas of size $n^{1+2k\beta} \leq kn^{1+\varepsilon}$, which means the hypothesis of Theorem 1.1 is not satisfied either.

To prove Theorem 1.1, we define the following auxiliary problem to serve as our oracle.

Definition 3.4 (Succinct-MCSP)

- Input: $\langle 1^s, (z_1, b_1), \ldots, (z_t, b_t)\rangle$, where $z_1, \ldots, z_t \in \{0,1\}^m$ for some $m \leq s$ and $b_1, \ldots, b_t \in \{0,1\}$.
- Decide: whether there exists a circuit C with $\mathsf{size}(C) \leq s$ such that $C(z_i) = b_i$ for all $i \in [t]$.

Claim 3.5 — We have $\mathsf{Succinct\text{-}MCSP} \in \mathsf{NP}$.

Proof. We can decide Succinct-MCSP using the following nondeterministic algorithm:

- Nondeterministically guess a circuit C of size at most s (as a $64s\log s$-bit string).
- Loop through all $i \in [t]$. For each, run C on z_i and check whether it outputs b_i.
- If $C(z_i) = b_i$ for all i, then *accept*; otherwise *reject*.

This algorithm runs in polynomial time (its input length is at least $s + m + t$, and running C on a single string z can be done in $\mathsf{poly}(s)$ time), showing that $\mathsf{Succinct\text{-}MCSP} \in \mathsf{NP}$. □

We'll then (unconditionally) construct an oracle formula for $(1, 1-\delta)\text{-}\mathsf{MCSP}[s]$ with the following bounds, from which Theorem 3.1 immediately follows — the assumption $\mathsf{NP} \subseteq \mathsf{Formula}[\mathsf{poly}]$ allows us to replace the oracle queries with $\mathsf{poly}(s, \delta^{-1})$-size formulas.

Lemma 3.6
For any s, δ, and n with $s \geq \log n$, there is a Succinct-MCSP-oracle formula of size $n \cdot \mathsf{poly}(s, \delta^{-1})$ making nonadaptive queries of fan-in $\mathsf{poly}(s, \delta^{-1})$ that decides $(1, 1-\delta)\text{-}\mathsf{MCSP}[s]$ on length-n inputs.

Explicitly, Lemma 3.6 means that there exists $k \in \mathbb{N}$ such that there is a Succinct-MCSP-oracle formula for $(1, 1-\delta)\text{-}\mathsf{MCSP}[s]$ on length-n inputs with size $ns^k\delta^{-k}$ and where all queries have fan-in at most $s^k\delta^{-k}$ (the construction gives an explicit and small value of k, but the exact value is not important).

Remark 3.7. To see why this implies Theorem 3.1, if we assume $\mathsf{NP} \subseteq \mathsf{Formula}[\mathsf{poly}]$ (in proving Theorem 3.1 by contrapositive), then since $\mathsf{Succinct\text{-}MCSP} \in \mathsf{NP}$, there is some ℓ such that $\mathsf{Succinct\text{-}MCSP} \in \mathsf{Formula}[n^\ell]$, which means we can replace each oracle queries with a $(s^k\delta^{-k})^\ell$-sized formulas in its inputs. This multiplies the size of our resulting formula by a factor of at most $(s^k\delta^{-k})^\ell$, giving an ordinary formula for $(1, 1-\delta)$-$\mathsf{MCSP}[s]$ of size $ns^{k(1+\ell)}\delta^{-k(1+\ell)}$.

As stated in Subsubsection 1.2.1, the main idea is to randomly sample a small number of input-output pairs from f and run the Succinct-MCSP oracle on this small sample; the following claim allows us to do so.

Claim 3.8 — Let $t = 128\delta^{-1}s\log s$, and suppose that we choose $z_1, \dots, z_t \in \{0,1\}^m$ uniformly and independently at random.

- If f is a **YES** instance of $(1, 1-\delta)$-$\mathsf{MCSP}[s]$, then $\langle 1^s, (z_1, f(z_1)), \dots, (z_t, f(z_t))\rangle$ is always a **YES** instance of Succinct-MCSP.
- If f is a **NO** instance of $(1, 1-\delta)$-$\mathsf{MCSP}[s]$, then $\langle 1^s, (z_1, f(z_1)), \dots, (z_t, f(z_t))\rangle$ is a **NO** instance of Succinct-MCSP with probability at least $\frac{3}{4}$.

Proof. The first statement is clear — if there is a circuit C with $\mathsf{size}(C) \leq s$ that correctly computes the full truth table of f, then C certainly computes all the input-output pairs $(z_i, f(z_i))$.

For the second statement, first consider any fixed circuit C with $\mathsf{size}(C) \leq s$. Then $\mathbb{P}_z[C(z) = f(z)] < 1-\delta$ for $z \in \{0,1\}^m$ chosen uniformly at random (by the definition of $(1, 1-\delta)$-$\mathsf{MCSP}[s]$), so

$$\mathbb{P}[C(z_i) = f(z_i) \text{ for all } i] < (1-\delta)^t \leq e^{-\delta t} \leq 2^{-128 s\log s}.$$

But there are at most $2^{64s\log s}$ circuits of size at most s, so union-bounding over all such circuits, we get

$$\mathbb{P}[\text{exists } C \text{ with } C(z_i) = f(z_i) \text{ for all } i] \leq 2^{64s\log s} \cdot 2^{-128s\log s} \leq \frac{1}{4}. \qquad \square$$

Proof of Lemma 3.6. Imagine that we perform n independent trials of the form described in Claim 3.8 — for each $j \in [n]$, we choose $z_{j1}, \dots, z_{jt} \in \{0,1\}^m$ uniformly at random and feed $\langle 1^s, (z_{j1}, f(z_{j1})), \dots, (z_{jt}, f(z_{jt}))\rangle$ into a Succinct-MCSP-oracle gate — and take the **AND** of their results. By Claim 3.8, on any fixed input f, this gives the correct answer with probability 1 if f is a **YES** instance of $(1, 1-\delta)$-$\mathsf{MCSP}[s]$, and at least $1 - 2^{-2n}$ if f is a **NO** instance. So union-bounding over all 2^n possible inputs, this construction gives the correct answer on *all* inputs f with probability at least $1 - 2^n \cdot 2^{-2n}$ (over the random choices of all nt strings z_{ji}). This probability is positive, so we can find *some* choice of strings for which this is the case.

We then hardcode these strings into our formula — we take the oracle formula

$$\bigwedge_{j\in[n]} \mathsf{Succinct\text{-}MCSP}(\langle 1^s, (z_{j1}, f(z_{j1})), \dots, (z_{jt}, f(z_{jt}))\rangle),$$

where 1^s and all the strings z_{ji} are hardcoded constants, and each $f(z_{ji})$ is a specific bit of the input. (Each oracle query has fan-in $O(s + mt) = \mathsf{poly}(s, \delta^{-1})$, and we have n such queries in parallel.) $\square$

§3.2 Magnification for Gap-MKtP

In this subsection, we'll prove the following slightly more general version of Theorem 1.3.

Theorem 3.9 ([OPS19, Theorem 1])

There is an absolute constant $c > 0$ such that the following holds: suppose that there exists $p\colon \mathbb{N} \to \mathbb{N}$ with $p(n) \geq \log n$ for all n and such that $\mathsf{Gap\text{-}MKtP}[p, p + c\log n] \notin \mathsf{Formula\text{-}XOR}[n \cdot \mathsf{poly}(p)]$. Then $\mathsf{EXP} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

We'll define the following auxiliary problem (which is an analog of Succinct-MCSP) to serve as our oracle.

Definition 3.10 (Succinct-MKtP)

- Input: $\langle 1^p, n, (d_1, b_1), \ldots, (d_t, b_t)\rangle$, where n and $d_1, \ldots, d_t \in [n]$ are integers written in binary, $p \geq \log n$, and $b_1, \ldots, b_t \in \{0, 1\}$.
- Decide: whether there exists a string $y \in \{0, 1\}^n$ with $\mathsf{Kt}(y) \leq p$ such that $y_{d_i} = b_i$ for all $i \in [t]$.

(We use the somewhat nonstandard letter d for indices so that we can use letters such as i and j in the same way as in the proofs for MCSP.)

Claim 3.11 — We have $\mathsf{Succinct\text{-}MKtP} \in \mathsf{EXP}$.

Proof. To solve Succinct-MKtP, we can simply enumerate over all possible Turing machine descriptions $\langle M\rangle$ of length at most p (of which there are at most 2^p), run each such Turing machine M for 2^p steps, and check that $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$ (if M has not halted within 2^p steps, we automatically know this is not true) and that its output y meets the given specifications. (If we find some M that passes all these checks, then we *accept*; otherwise we *reject*.) $\square$

We'll then construct an oracle formula with the following bounds; this immediately implies Theorem 3.9 (in the same way as described in Remark 3.7).

Lemma 3.12

There is an absolute constant $c > 0$ such that for any p and n with $p \geq \log n$, there is a Succinct-MKtP-oracle **XOR**-formula of size $n \cdot \mathsf{poly}(p)$ and making nonadaptive queries of fan-in $\mathsf{poly}(p)$ that decides $\mathsf{Gap\text{-}MKtP}[p, p + c\log n]$ on length-n inputs.

In order to prove Lemma 3.12, as stated in Subsubsection 1.2.1, we'll combine the ideas from Subsection 3.1 with error-correcting codes. We'll need codes with the following properties. There are several constructions of such codes, often with much stronger guarantees; for example, see [Spi96].

Theorem 3.13

There exist linear error-correcting codes with constant rate and relative distance that can be encoded and decoded by polynomial-time algorithms. More precisely, there are constants $a \in \mathbb{N}$ and $\gamma \in (0, \frac{1}{2})$ and a function $E\colon \{0, 1\}^* \to \{0, 1\}^*$ such that:

- The restriction of E to inputs of length n is a linear map $\{0, 1\}^n \to \{0, 1\}^{an}$ for each n (i.e., for each index $d \in [an]$, we can write $E(x)_d$ as $\langle x, w_d\rangle$ for some $w_d \in \{0, 1\}^n$).
- There is a polynomial-time algorithm that, given x, computes $E(x)$.
- For all n and all $y \in \{0, 1\}^{an}$, there is at most one $x \in \{0, 1\}^n$ such that $E(x)$ and y differ in at most a γ-fraction of indices. Furthermore, there is a polynomial-time algorithm that, given y (of length an for any n), computes the string x with this property (if one exists).

The main new observation is that applying such an error-correcting code doesn't affect the Kt-complexity of a string too much, and it allows us to get the 'distance' property we need for a random sampling argument to work (i.e., that 'good' and 'bad' inputs differ on a constant fraction of indices).

Claim 3.14 — There is an absolute constant c for which the following statements hold (where $n = |x|$):

(i) $\mathsf{Kt}(E(x)) \leq \mathsf{Kt}(x) + \frac{1}{2}c\log n$.

(ii) For all y which differ from $E(x)$ in at most a γ-fraction of indices, $\mathsf{Kt}(y) \geq \mathsf{Kt}(x) - \frac{1}{2}c\log n$.

Proof. For (i), if we have a Turing machine that prints x, then we can obtain one that prints $E(x)$ by first running the original machine and then the polynomial-time encoder for E. This increases the description length of the machine by a constant (since the encoder is a *fixed* algorithm) and the runtime by $\mathsf{poly}(n)$, so it increases Kt-complexity by $O(\log n)$.

Similarly for (ii), if we have a Turing machine that prints y, then we can obtain one that prints x by first running the original machine and then the polynomial-time *decoder* for E (which increases Kt-complexity by $O(\log n)$ for the same reason). □

Using this, we can get an analog of Claim 3.8 for $\mathsf{Gap\text{-}MKtP}$.

Claim 3.15 — Let $q = p + \frac{1}{2}c\log n$ and $t = 4\gamma^{-1}q$, and suppose we choose indices $d_1, \ldots, d_t \in [an]$ uniformly and independently at random.

- If x is a **YES** instance of $\mathsf{Gap\text{-}MKtP}[p, p + c\log n]$, then $\langle 1^q, an, (d_1, E(x)_{d_1}), \ldots, (d_t, E(x)_{d_t})\rangle$ is a **YES** instance of $\mathsf{Succinct\text{-}MKtP}$.
- If x is a **NO** instance of $\mathsf{Gap\text{-}MKtP}[p, p + c\log n]$, then $\langle 1^q, an, (d_1, E(x)_{d_1}), \ldots, (d_t, E(x)_{d_t})\rangle$ is a **NO** instance of $\mathsf{Succinct\text{-}MKtP}$ with probability at least $\frac{3}{4}$.

The proof is essentially the same as that of Claim 3.8, except that for the second statement, we use the fact that there are at most 2^q strings y with $\mathsf{Kt}(y) \leq q$ (as there are only 2^q possible Turing machine descriptions of length at most q) to union-bound over all such strings (in place of the fact that there are at most $2^{64s\log s}$ circuits of size at most s). We omit the details.

With this, we can prove Lemma 3.12 in a way very similar to the proof of Lemma 3.6.

Proof of Lemma 3.12. Imagine performing n independent trials sd in Claim 3.15 — for each $j \in [n]$, we sample $d_{j1}, \ldots, d_{jt} \in [an]$ uniformly at random and feed $\langle 1^q, an, (d_{j1}, E(x)_{d_{j1}}), \ldots, (d_{jt}, E(x)_{d_{jt}})\rangle$ into a $\mathsf{Succinct\text{-}MKtP}$ oracle gate — and taking the **AND** of their results. By Claim 3.15, this gives the correct answer on any fixed input x with probability 1 if x is a **YES** instance of $\mathsf{Gap\text{-}MKtP}[p, p+c\log n]$ and at least $1 - 2^{-2n}$ if x is a **NO** instance. So by union-bounding over all 2^n possible inputs, it gives the correct answer on *all* inputs x with probability at least $1 - 2^{-2n} \cdot 2^n > 0$, which means there is some choice of nt indices d_{ji} for which this is the case. We then fix these indices and hardcode them into our formula — so the final formula we take is

$$\bigwedge_{j\in[n]} \mathsf{Succinct\text{-}MKtP}(\langle 1^q, an, (d_{j1}, E(x)_{d_{j1}}), \ldots, (d_{jt}, E(x)_{d_{jt}})\rangle),$$

where 1^q, an, and all the indices d_{ji} are hardcoded constants, and each $E(x)_{d_{ji}}$ is a parity function of x (by the fact that E is linear) and therefore a valid leaf of a **XOR**-formula. (So this formula consists of n parallel oracle queries, where each query directly takes in $O(q + t\log an) = \mathsf{poly}(p)$ leaves.) □

§4 Magnification via anticheckers

In this section, we'll prove the following slightly more general version of Theorem 1.2, following the outline described in Subsubsection 1.2.2.

Theorem 4.1 ([OPS19, Theorem 4])

Suppose that there exists $s\colon \mathbb{N} \to \mathbb{N}$ with $s(n) \geq \log n$ for all n and such that $\mathsf{Gap\text{-}MCSP}[s, s\log n] \not\in \mathsf{Circuit}[n \cdot \mathsf{poly}(s)]$. Then $\mathsf{NP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$.

The main idea behind the proof is the concept of anticheckers.

Definition 4.2. An *antichecker* for a function $f\colon \{0,1\}^m \to \{0,1\}$ against circuits of size s is a collection of strings $\mathcal{S} \subseteq \{0,1\}^m$ such that for every circuit C with $\mathsf{size}(C) \leq s$, there is some string $z \in \mathcal{S}$ for which $C(z) \neq f(z)$.

Of course, if an antichecker for f against circuits of size s exists, then f can't be computed by a circuit of size (at most) s.

In Subsection 4.1, we'll prove the following lemma on the *existence* of *small* anticheckers.

Lemma 4.3

Let $s \geq m$, and suppose that $f\colon \{0,1\}^m \to \{0,1\}$ cannot be computed by a circuit of size (at most) sm. Then there exists $\mathcal{S} \subseteq \{0,1\}^m$ of size $|\mathcal{S}| = O(s \log s)$ such that $\mathcal{S}$ is an antichecker for f against circuits of size s.

Specifically, we'll give a combinatorial proof of Lemma 4.3 via a 'greedy' construction — we'll show that if we construct $\mathcal{S}$ one string at a time, at every step we can find a new string z such that adding z to $\mathcal{S}$ would eliminate a constant fraction of the 'surviving' circuits (circuits of size at most s that compute f on the set $\mathcal{S}$ we've built so far), which means it takes $O(s \log s)$ steps to eliminate all circuits.

In Subsections 4.2 and 4.3, we'll then show how to turn this existence proof into an efficient NP-oracle circuit for $\mathsf{Gap\text{-}MCSP}[s, sm]$ (note that $m = \log n$). Intuitively, our oracle circuit will try to compute such an antichecker for f; if it succeeds then we immediately know f can't be computed by a circuit of size at most s, while if it fails then we'll know f can be computed by a circuit of size at most sm. Implementing the greedy construction essentially amounts to repeatedly running an approximate counting problem on short inputs; in Subsection 4.2 we'll describe the specifics of this approximate counting problem and show how to solve it with a small (i.e., $\mathsf{poly}(s)$-sized) NP-oracle circuit. In Subsection 4.3 we'll use these $\mathsf{poly}(s)$-sized approximate counting oracle circuits to build a $n \cdot \mathsf{poly}(s)$-sized oracle circuit that implements the greedy construction, which will imply Theorem 1.2.

As a technical note, throughout this section, whenever we talk about counting a 'number of circuits' of size at most s (with some property), we really mean that we're counting the number of circuit *descriptions* (i.e., $2^{64s\log s}$-bit strings) — even if two strings represent the same circuit, we'll count them separately.

§4.1 The existence of anticheckers

In this subsection, we'll prove the following lemma, which immediately implies Lemma 4.3 (on the existence of small anticheckers).

Lemma 4.4

Suppose that $s \geq m$ and $f\colon \{0,1\}^m \to \{0,1\}$ does not have circuits of size at most sm, and $\mathcal{C}$ is a collection of circuits each with size at most s. Then there exists some $z \in \{0,1\}^m$ such that $C(z) \neq f(z)$ for at least a $\frac{1}{128}$-fraction of the circuits C in $\mathcal{C}$.

Proof. Let $\ell = \frac{1}{2}m$. The key observation is that for any ℓ circuits $C_1, \ldots, C_\ell \in \mathcal{C}$ (possibly with repetition), there must exist some $z \in \{0,1\}^m$ such that at least half of $C_1, \ldots, C_\ell$ fail to compute f on z — otherwise we would have $f(z) = \mathsf{Majority}(C_1(z), \ldots, C_\ell(z))$ for all $z \in \{0,1\}^m$, so we could obtain a circuit C that computes f by first computing $C_1, \ldots, C_\ell$ and then taking their majority. It's known that the $\mathsf{Majority}$ function on ℓ inputs can be computed by a circuit of size $O(\ell)$ (for example, see [DKKY10] for a reference), so the size of C would be at most $s\ell + O(\ell) \leq sm$, contradicting the fact that f does not have circuits of size at most sm.

Now let $|\mathcal{C}| = k$, and assume for contradiction that for every $z \in \{0,1\}^m$, there are at most $\frac{1}{128}k$ circuits C in $\mathcal{C}$ for which $C(z) \neq f(z)$. We'll obtain a contradiction by counting the quantity

$$(*) = \left\{(z, C_1, \ldots, C_\ell) \mid C_i(z) \neq f(z) \text{ for at least } \frac{1}{2}\ell \text{ indices } i \in [\ell]\right\}$$

in two different ways. On one hand, there are k^ℓ ways to choose $C_1, \ldots, C_\ell$, and for each such choice there is at least one way to choose z (by the above observation); this means $(*) \geq k^\ell$. On the other hand, suppose that we instead choose z first; this can be done in 2^m ways. Then there are at most 2^ℓ ways to choose a set of indices $i \in [\ell]$ (of size at least $\frac{1}{2}\ell$) for which to have $C_i(z) \neq f(z)$. Finally, for each index i in this set, there are at most $\frac{1}{128}k$ choices for C_i, while for each of the remaining indices i, there are at most k. So

$$(*) \leq 2^m \cdot 2^\ell \cdot \left(\frac{k}{128}\right)^{\ell/2} \cdot k^{\ell/2} = k^\ell \cdot 2^{m+m/2-7m/4} < k^\ell$$

(plugging in $\ell = \frac{1}{2}m$), which is a contradiction. □

Proof of Lemma 4.3. We'll build $\mathcal{S}$ one string at a time, keeping track of the collection $\mathcal{C}$ of 'surviving' circuits (i.e., circuits that compute f correctly on $\mathcal{S}$) as we go along. We initialize $\mathcal{S}$ as $\emptyset$ and $\mathcal{C}$ as the collection of *all* circuits of size at most s, so that $|\mathcal{C}| \leq 2^{64s\log s}$. Then at each step, by Lemma 4.4 there exists some $z \in \{0,1\}^m$ such that adding z to $\mathcal{S}$ shrinks $\mathcal{C}$ by a factor of at most $1 - \frac{1}{128}$. If we do this for $128 \cdot 64s\log s$ steps, then in the end $\mathcal{C}$ will have shrunk by a factor of at most

$$\left(1 - \frac{1}{128}\right)^{128\cdot 64s\log s} \leq e^{-64s\log s} < 2^{-64s\log s},$$

which means it must be empty (as its size is less than 1). This means *every* circuit C of size at most s disagrees with f on some $z \in \mathcal{S}$ (or else C would be in $\mathcal{C}$), so $\mathcal{S}$ is an antichecker for f. □

§4.2 Estimating the number of surviving circuits

In this subsection, we'll prove the following lemma, which essentially solves, using a NP oracle, the approximate counting problem that we need to implement the greedy construction in Subsection 4.1.

Lemma 4.5

There is some problem $\mathsf{Succinct\text{-}MCSP\text{-}with\text{-}Hash} \in \mathsf{NP}$ such that the following holds: let $s \geq m$, $t \in \mathbb{N}$, $\varepsilon > 0$, and $0 \leq k < 2^{64s \log s}$. Then there is a $\mathsf{Succinct\text{-}MCSP\text{-}with\text{-}Hash}$-oracle circuit of size $\mathsf{poly}(s, t, \varepsilon^{-1})$ that takes in inputs $z_1, \ldots, z_t \in \{0,1\}^m$ and $b_1, \ldots, b_t \in \{0,1\}$ and:

- *Accepts* if the number of circuits of size at most s with $C(z_i) = b_i$ for all $i \in [t]$ is at most k.
- *Rejects* if this number is at least $k(1+\varepsilon)$.

(When we apply Lemma 4.5 to produce our antichecker-constructing circuit, the quantities s, t, ε, and k will always be fixed values (not depending on the input), so it's fine for this circuit to depend on them.)

We'll prove Lemma 4.5 via pairwise independent hash families — for all $r, \ell \in \mathbb{N}$, let $\mathcal{H}_{r,\ell} = \{h_v \colon \{0,1\}^r \to \{0,1\}^\ell\}$ be an (efficiently computable) pairwise independent hash family, meaning that:

- If we choose $h \in \mathcal{H}_{r,\ell}$ uniformly at random, then for all distinct $x, y \in \{0,1\}^r$, the distribution of $(h(x), h(y))$ is uniform in $\{0,1\}^\ell \times \{0,1\}^\ell$.
- The elements of $\mathcal{H}_{\ell,r}$ are given by seeds v of length $\mathsf{poly}(r, \ell)$.
- There is a polynomial-time algorithm to compute $h_v(x)$ given r, ℓ, v, and $x \in \{0,1\}^r$.

There are several standard constructions of such families — for example, taking $h_{M,w}(x) = Mx + w$, where the seed v consists of a matrix $M \in \mathbb{F}_2^{\ell \times r}$ and vector $w \in \mathbb{F}_2^\ell$ chosen uniformly at random, works.

The main point is that if a set $S \subseteq \{0,1\}^r$ has size on the same 'order of magnitude' as 2^ℓ, then we can estimate $|S|$ by the probability (over a randomly chosen $h \in \mathcal{H}_{r,\ell}$) that some element of S hashes to 0^ℓ.

Claim 4.6 — Let $S \subseteq \{0,1\}^r$. Then for $h \in \mathcal{H}_{r,\ell}$ chosen uniformly at random, we have

$$\frac{|S|}{2^\ell}\left(1 - \frac{|S|}{2^{\ell+1}}\right) \leq \mathbb{P}_h[\text{exists } x \in S \text{ with } h(x) = 0^\ell] \leq \frac{|S|}{2^\ell}.$$

Proof. For the upper bound, by a union bound we have

$$\mathbb{P}_h[\text{exists } x \in S \text{ with } h(x) = 0^\ell] \leq \sum_{x \in S} \mathbb{P}_h[h(x) = 0^\ell] = \frac{|S|}{2^\ell}.$$

For the lower bound, by the principle of inclusion-exclusion and the pairwise independence of $\mathcal{H}_{r,\ell}$ we have

$$\begin{aligned} \mathbb{P}_h[\text{exists } x \in S \text{ with } h(x) = 0^\ell] &\geq \sum_{x \in S} \mathbb{P}_h[h(x) = 0^\ell] - \sum_{\{x,y\} \in \binom{S}{2}} \mathbb{P}_h[h(x) = h(y) = 0^\ell] \\ &\geq \frac{|S|}{2^\ell}\left(1 - \frac{|S|}{2^{\ell+1}}\right). \end{aligned}$$ □

We then define our oracle in the following way.

Definition 4.7 (Succinct-MCSP-with-Hash)

- Input: $\langle 1^s, 1^\ell, v, (z_1, b_1), \ldots, (z_t, b_t) \rangle$, where $z_1, \ldots, z_t \in \{0,1\}^m$ for some $m \leq s$ and $b_1, \ldots, b_t \in \{0,1\}$, and v is the seed to a hash function $h_v \colon \{0,1\}^{64s \log s} \to \{0,1\}^\ell$ (from $\mathcal{H}_{64s \log s, \ell}$).
- Decide: whether there exists a circuit C with $\mathsf{size}(C) \leq s$ such that $C(z_i) = b_i$ for all $i \in [t]$ and $h_v(\langle C \rangle) = 0^\ell$.

It's clear that $\mathsf{Succinct\text{-}MCSP\text{-}with\text{-}Hash} \in \mathsf{NP}$, as we can simply guess C and check that it works.

Proof of Lemma 4.5. We can assume without loss of generality that $\varepsilon \leq 1$ (by replacing it with 1 otherwise). Let $r = 64s\log s$ and choose $\ell \in \mathbb{N}$ such that $\frac{\varepsilon}{8} \leq \frac{k}{2^\ell} \leq \frac{\varepsilon}{4}$ (note that $\ell = \mathsf{poly}(s, \log \varepsilon^{-1})$).

Now imagine performing $4096\varepsilon^{-4}t(m+1) = \mathsf{poly}(s, t, \varepsilon^{-1})$ trials where in each trial, we choose $h_v \in \mathcal{H}_{r,\ell}$ uniformly at random and feed $\langle 1^s, 1^\ell, v, (z_1, b_1), \ldots, (z_t, b_t)\rangle$ to the Succinct-MCSP-with-Hash oracle. We then compute the fraction p of trials on which the oracle accepts; we *accept* if $p \leq \frac{k}{2^\ell}(1 + \frac{\varepsilon}{4})$ and *reject* otherwise.

We'll show (using a Chernoff bound) that this works (i.e., outputs the correct answer) with extremely high probability, so that we can union-bound over all $2^{t(m+1)}$ possible inputs to find a choice of randomness for which it works for all inputs (then we can hardcode these random strings v into our circuit). To do so, let p^* be the probability that the oracle accepts on a single trial. Let

$$S = \{\langle C\rangle \mid \mathsf{size}(S) \leq s \text{ and } C(z_i) = b_i \text{ for all } i \in [t]\} \subseteq \{0,1\}^{64s\log s}$$

be the set whose size we are trying to estimate, so that

$$p^* = \mathbb{P}_h[\text{exists } x \in S \text{ with } h(x) = 0^\ell]$$

(essentially by definition). By Claim 4.6, if $|S| \leq k$ then $p^* \leq \frac{k}{2^\ell}$, while if $|S| \geq k(1+\varepsilon)$ then

$$p^* \geq \frac{k(1+\varepsilon)}{2^\ell}\left(1 - \frac{k(1+\varepsilon)}{2^{\ell+1}}\right) \geq \frac{k(1+\varepsilon)}{2^\ell}\left(1 - \frac{k}{2^\ell}\right) \geq \frac{k(1+\varepsilon)}{2^\ell}\left(1 - \frac{\varepsilon}{4}\right) \geq \frac{k}{2^\ell}\left(1 + \frac{\varepsilon}{2}\right).$$

This means that for the algorithm to make an error in either direction, we must have

$$|p^* - p| \geq \frac{k}{2^\ell} \cdot \frac{\varepsilon}{4} \geq \frac{\varepsilon}{8} \cdot \frac{\varepsilon}{4} = \frac{\varepsilon^2}{32}.$$

By a Chernoff bound (since we're performing $4096\varepsilon^{-4}t(m+1)$ independent trials), the probability that this occurs is much less than $2^{t(m+1)}$, allowing us to union-bound over all possible inputs and find a choice of randomness for which this algorithm works for all inputs. We then hardcode this choice of randomness into our circuit; this gives us a circuit of size $\mathsf{poly}(s, t, \varepsilon^{-1})$. ☐

§4.3 An oracle circuit for Gap-MCSP

Finally, we'll use the approximate counting circuit of Lemma 4.5 to implement the greedy construction of Lemma 4.3 with an efficient Succinct-MCSP-with-Hash-oracle circuit making short queries; this will give the following lemma, which immediately implies Theorem 4.1.

Lemma 4.8

For any s and n with $s \geq \log n$, there is a Succinct-MCSP-with-Hash-oracle circuit of size $n \cdot \mathsf{poly}(s)$ making queries of fan-in $\mathsf{poly}(s)$ that decides Gap-MCSP$[s, s\log n]$ on length-n inputs.

Proof. Fix $\varepsilon > 0$ to be a constant with $(1 - \frac{1}{128})(1+\varepsilon) \leq 1 - \frac{1}{256}$. Let $t = 256 \cdot 64s\log s$ and $k_0 = 2^{64s\log s}$, and for each $1 \leq i \leq t$, let $k_i = (1 - \frac{1}{256})k_{i-1}$.

Our circuit now works as follows. We attempt to construct an antichecker $\mathcal{S}$ for f one string at a time. To do so, on the ith step (for $1 \leq i \leq t$), suppose that we've already chosen $z_1, \ldots, z_{i-1}$. We then loop through all $z \in \{0,1\}^m$ one at a time. For each, we run the oracle circuit from Lemma 4.5 on $z_1, \ldots, z_{i-1}, z$ and $f(z_1), \ldots, f(z_{i-1}), f(z)$ with t replaced by i and k replaced by $(1 - \frac{1}{128})k_{i-1}$. If this oracle circuit accepts on any z, then we set z_i to be the lexicographically first such z; otherwise, if no such z exists (i.e., the oracle rejects on all z), then we *accept*. Finally, after t steps, if we haven't yet accepted (so we've successfully constructed $z_1, \ldots, z_t$), then we *reject*.

We'll first check that this procedure works. First, we claim that if the procedure rejects, then f cannot be computed by a circuit of size s. To see this, on the last step the circuit from Lemma 4.5 accepts $z_1, \dots, z_t$, $f(z_1), \dots, f(z_t)$, which means we must have

$$\#\{C \mid \mathsf{size}(C) \leq s \text{ and } C(z_i) = f(z_i) \text{ for all } i \in [t]\} < \left(1 - \frac{1}{128}\right) k_{t-1}(1+\varepsilon) \leq k_t.$$

But we chose t in such a way that $k_t < 1$, which means there cannot exist any such circuits. (In other words, we've successfully constructed an antichecker for f, which means f certainly cannot have a small circuit.)

On the other hand, we claim that if f cannot be computed by a circuit of size $s \log n = sm$, then this procedure necessarily rejects (intuitively, this corresponds to the procedure successfully running the greedy construction of Lemma 4.3 without getting stuck, eventually producing an antichecker $\mathcal{S} = \{z_1, \dots, z_t\}$). To see this, for each $0 \leq i \leq t$, let $\mathcal{C}_i$ be the collection of 'surviving' circuits once we've chosen $z_1, \dots, z_i$. Let $(\mathrm{I})_i$ be the statement that $|\mathcal{C}_i| \leq k_i$ (for each $0 \leq i \leq t$), and let $(\mathrm{II})_i$ be the statement that the procedure successfully chooses some string z_i (for each $1 \leq i \leq t$). Then:

- $(\mathrm{I})_0$ is true because $|\mathcal{C}_0| \leq 2^{64 s \log s} = k_0$.
- If $(\mathrm{I})_{i-1}$ is true, then $(\mathrm{II})_i$ is true by Lemma 4.4 — explicitly, on the ith step, for each z the circuit from Lemma 4.5 is trying to approximately count the quantity

$$\#\{C \in \mathcal{C}_{i-1} \mid C(z) = f(z)\}$$

  (with $k = (1 - \frac{1}{128})k_{i-1}$), and by Lemma 4.4 there exists some z for which this quantity is at most $(1 - \frac{1}{128})\,|\mathcal{C}_{i-1}|$, which by $(\mathrm{I})_{i-1}$ is at most $(1 - \frac{1}{128})k_{i-1}$; this means the circuit from Lemma 4.5 must accept on this choice of z (by the first guarantee of Lemma 4.5).
- If $(\mathrm{II})_i$ is true, then $(\mathrm{I})_i$ must be true by the second guarantee of Lemma 4.5 — the fact that the circuit from Lemma 4.5 accepted on $z = z_i$ means that

$$|\mathcal{C}_i| = \#\{C \mid \mathsf{size}(C) \leq s \text{ and } C(z_j) = f(z_j) \text{ for all } j \in [i]\} < \left(1 - \frac{1}{128}\right) k_{i-1}(1+\varepsilon) \leq k_i.$$

So by induction, both statements are true for all i, which in particular means the procedure successfully runs to completion.

Finally, this procedure can be implemented by a $n \cdot \mathsf{poly}(s)$-size circuit — we essentially have $t = \mathsf{poly}(s)$ 'layers' where the ith layer ends up producing $(z_i, f(z_i))$ (or a FAILURE message telling the circuit to accept). At each layer, for every $z \in \{0,1\}^m$ we have a $\mathsf{poly}(s)$-sized oracle circuit from Lemma 4.5 where $z_1, \dots, z_{i-1}$ and $f(z_1), \dots, f(z_{i-1})$ are wired in from the previous layers, z is hardcoded, and $f(z)$ (which is a specific bit of our input f) is taken straight from the appropriate input wire. This oracle circuit at each z eventually produces a single bit (either *accept* or *reject*), and we can find the lexicographically first z for which this bit is *accept* and take the corresponding $(z, f(z))$ to be the output of the layer. □

§5 Magnification via compression

In this section, we'll prove Theorems 1.5, 1.6, and 1.7, following the outline in Subsubsection 1.2.3.

§5.1 Magnification for Search-MCSP with circuits

In this subsection, we prove the following slightly more general version of Theorem 1.5.

Theorem 5.1 ([MMW19, Theorem 1.4])

Suppose there exists $s\colon \mathbb{N} \to \mathbb{N}$ such that $\mathsf{Search\text{-}MCSP}[s]$ cannot be solved (on inputs of length n) by circuits of size $n \cdot \mathsf{poly}(s)$ and depth $\mathsf{poly}(s)$. Then $\mathsf{NP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$.

For this, we'll define the following auxiliary problem to serve as our oracle. Intuitively (following the sketch in Subsubsection 1.2.3), we want an oracle that takes in a collection of small circuits corresponding to disjoint intervals, and finds a small circuit C that 'merges' them (i.e., that matches each circuit C_i on its specified interval) or tells us that no such circuit exists. But we need the oracle to output only a single bit, so we instead have it output the jth bit of the description of the *lexicographically first* such circuit C (where we give the oracle j as well), so that running the oracle over all indices j gives us a full description of such a circuit C.

Definition 5.2 (Circuit-Merge)

- Input: $\langle 1^s, C_1, \ldots, C_t, [z_1, z_1'], \ldots, [z_t, z_t'], j\rangle$, where:
  - $C_1, \ldots, C_t$ are m-input circuits of size at most s (for some $m \le s$).
  - $[z_1, z_1'], \ldots, [z_t, z_t']$ are disjoint intervals in $\{0,1\}^m$ under the lexicographical order (with each interval specified by its two endpoints $z_i, z_i' \in \{0,1\}^m$).
  - j is an integer with $1 \le j \le 64s\log s$.
- Output: $\langle C\rangle_j$, where C is the lexicographically first circuit with the properties that $\mathsf{size}(C) \le s$ and $C(x) = C_i(x)$ for all $i \in [t]$ and $x \in [z_i, z_i']$, or 0 if no such C exists.

(We allow some of $C_1, \ldots, C_t$ to be the all-0's string as well, in which case the oracle should output 0.)

Claim 5.3 — We have $\mathsf{Circuit\text{-}Merge} \in \Sigma_3\mathsf{P}$.

Proof. We can solve Circuit-Merge using the following Σ_3 algorithm:

- Existentially guess a circuit C of size at most s (as a $64s\log s$-bit string) with $\langle C\rangle_j = 1$.
- To check that C itself merges $C_1, \ldots, C_t$ successfully, we universally guess $x \in \{0,1\}^m$. Then checking that $C(x)$ is what it should be can be done in polynomial time — we first find the index i such that $x \in [z_i, z_i']$ (if such i exists), then compute $C(x)$ and $C_i(x)$ (which can be done in $\mathsf{poly}(s)$ time) and check that they match.
- To check that there is no lexicographically smaller circuit that merges $C_1, \ldots, C_t$, we universally guess a circuit C' of size at most s which is lexicographically smaller than C. We then existentially guess an input $x' \in \{0,1\}^m$. Then we can check that $C'(x')$ is *not* what it should be in polynomial time — we first find i such that $x' \in [z_i, z_i']$, then compute $C'(x')$ and $C_i(x')$ and check that they *don't* match.

(This can be quantified as $(\exists\, C)(\forall\, x, C')(\exists\, x')[\cdots]$, where $[\cdots]$ is a polynomial-time checkable predicate.) □

We'll then construct the following oracle circuit for Search-MCSP.

Lemma 5.4

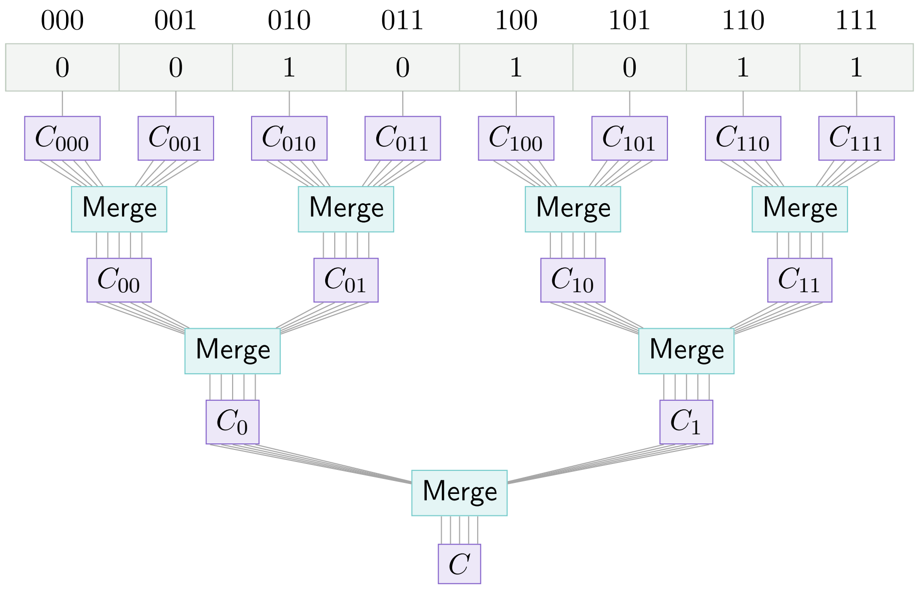
For any s and n with $s \ge \log n$, there exists a Circuit-Merge-oracle circuit of size $n \cdot \mathsf{poly}(s)$ and depth $O(\log n)$ where all oracle gates have fan-in $\mathsf{poly}(s)$ that solves $\mathsf{Search\text{-}MCSP}[s]$ on length-n inputs.

This implies Theorem 5.1 because if $\mathsf{NP} \subseteq \mathsf{Circuit}[\mathsf{poly}]$ then $\mathsf{PH} \subseteq \mathsf{Circuit}[\mathsf{poly}]$, which lets us implement each Circuit-Merge-oracle gate with a $\mathsf{poly}(s)$-sized circuit (this increases the size and depth of our circuit by a factor of $\mathsf{poly}(s)$).

Proof. We imagine building a binary tree with n leaves, each corresponding to a string $z \in \{0,1\}^m$ (with these leaves arranged in lexicographical order), and each node in the tree represents an interval in $\{0,1\}^m$ — specifically, the interval obtained by combining those of its children. We'll build our circuit so that for every node $[z, z']$ in the tree, the circuit contains $64s \log s$ nodes (in parallel) that together make up the description of a circuit C of size at most s that computes f on $[z, z']$ (or the all-0's string if no such circuit exists); then we can output the $64s \log s$ nodes corresponding to the root.

For the leaves of the tree, each leaf corresponds to an interval consisting of just one string z, so we can simply get $f(z)$ from the appropriate input wire and then produce (a description of) the constant circuit C that outputs $f(z)$ on all inputs.

Meanwhile, for each non-leaf node, suppose that its children correspond to the intervals $[z_1, z_1']$ and $[z_2, z_2']$, and that we've built the portions of our Search-MCSP circuit corresponding to these nodes (i.e., the $64s \log s$-node stretches storing circuits C_1 and C_2 that compute f on these intervals). Then to build the portion corresponding to the current node, we can simply call Circuit-Merge$(\langle 1^s, C_1, C_2, [z_1, z_1'], [z_2, z_2'], j\rangle)$ for all $1 \le j \le 64s \log s$ in parallel (where we hardcode everything except C_1 and C_2).



(This works because if f can be computed by a circuit of size at most s, then it can certainly be computed by a circuit of size at most s on every interval $[z, z']$.) □

§5.2 Magnification for Search-MCSP with streaming algorithms

We can prove Theorem 1.7 (which we restate in the more general form) in almost the same way.

Theorem 5.5 ([MMW19, Theorem 1.3])

Suppose that there is a poly(s)-time constructible $s\colon \mathbb{N} \to \mathbb{N}$ such that $s(n) \ge \log n$ for all $n \in \mathbb{N}$ and such that Search-MCSP$[s]$ cannot be solved by a poly(s)-space streaming algorithm with poly(s) update time. Then $\mathsf{P} \ne \mathsf{NP}$.

A streaming algorithm is only allowed to read its input once from left to right, and its update time is the maximum amount of time it spends between reading two consecutive bits of the input. We include the time spent before reading the first bit or reading the last bit of the input, so that an algorithm with poly(s) update time automatically runs in time $n \cdot \text{poly}(s)$. We also assume that the algorithm is given m (or n) at the start of its input.

We'll use the same auxiliary problem Circuit-Merge as our oracle, and we'll construct an oracle streaming algorithm with the following properties.

Lemma 5.6

Let $s\colon \mathbb{N} \to \mathbb{N}$ be any $\mathsf{poly}(s)$-time constructible function such that $s(n) \geq \log n$ for all $n \in \mathbb{N}$. Then Search-MCSP$[s]$ can be solved by a $\mathsf{poly}(s)$-space streaming algorithm with $\mathsf{poly}(s)$ update time making $\mathsf{poly}(s)$-length queries to a Circuit-Merge oracle.

This implies Theorem 5.5 because if $\mathsf{P} = \mathsf{NP}$ then $\mathsf{P} = \mathsf{PH}$, allowing us to implement the Circuit-Merge oracle with a polynomial-time algorithm (which contributes only $\mathsf{poly}(s)$ to our update time and space usage).

Proof. As we read through the input, we'll store the string z that we've read up to so far (meaning that we've read the truth table of f on the interval $[0^m, z]$) and a circuit C of size at most s computing f on $[0^m, z]$ (or that no such circuit exists) — this information takes $\mathsf{poly}(s)$ space. To maintain this property, suppose we currently have z and C in our storage, and the next input bit we read is b — so we now know that $f(x) = C(x)$ for all $x \in [0^m, z]$ and $f(z') = b$, where z' is the string immediately lexicographically after z. We then construct the circuit C' that always outputs b, and for each $1 \leq j \leq 64 s \log s$ we run the Circuit-Merge oracle on $\langle 1^s, C, C', [0^m, z], [z', z'], j\rangle$. Finally, we update z to z' and C to the output of the oracle (or more precisely, the concatenation of its outputs over all j), which is now a circuit of size at most s computing f on $[0^m, z']$ (or the all-0's string if no such circuit exists). Finally, once we've read through the entire input, we output the circuit we're storing (which computes f on all inputs by construction). □

§5.3 Magnification for Search-MKtP with circuits

Finally, we'll sketch the proof of the following general version of Theorem 1.6.

Theorem 5.7 ([MMW19, Theorem 1.7])

Suppose there exists $p\colon \mathbb{N} \to \mathbb{N}$ such that Search-MKtP$[p]$ cannot be solved (on inputs of length n) by circuits of size $n \cdot \mathsf{poly}(s)$ and depth $\mathsf{poly}(s)$. Then $\mathsf{EXP} \not\subseteq \mathsf{Circuit}[\mathsf{poly}]$.

To prove this, instead of Circuit-Merge, we define the obvious analog for MKtP.

Definition 5.8 (TM-Merge)

- INPUT: $\langle 1^p, n, M_1, \ldots, M_t, [d_1, d_1'], \ldots, [d_t, d_t'], j\rangle$, where:
  - n is written in binary, $[d_1, d_1']$, …, $[d_t, d_t']$ are disjoint intervals in $[n]$ (given by their endpoints $d_i, d_i' \in [n]$, which are also written in binary), and $p \geq \log n$.
  - M_1, …, M_t are Turing machines with $|\langle M_i\rangle| + \log \mathsf{time}(M_i) \leq p$.
  - j is an integer with $1 \leq j \leq p$.
- OUTPUT: $\langle M\rangle_j$, where M is the lexicographically first Turing machine with the properties that $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$ and that the output of M has length n and agrees with the output of M_i on all indices in $[d_i, d_i']$ for all $i \in [t]$, or 0 if no such M exists (or if the lexicographically first M has $|\langle M\rangle| < j$).

Note that the conditions on the input can all be checked in exponential time.

Claim 5.9 — We have TM-Merge $\in \mathsf{EXP}$.

Proof. We first run each of $M_1, \ldots, M_t$ for 2^p steps, to make sure they satisfy $|\langle M_i\rangle| + \log \mathsf{time}(M_i) \leq p$ and to get their outputs. Then we enumerate over all Turing machines M with description length at most p (of which there are at most 2^p); for each, we check whether $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$ and its output meets the given specifications. If we find some M which works, then we output $\langle M\rangle_j$ for the lexicographically first such M; otherwise we output 0. $\square$

Given this, the proof of Theorem 1.6 is essentially identical to that of Theorem 1.5 (we prove an analog of Lemma 5.4 with a TM-Merge oracle instead of a Circuit-Merge one), so we omit the rest of the details.

§6 Magnification via hashing

In this section, we'll prove Theorems 1.9, 1.10, 1.11, and 1.12, following the outline in Subsubsection 1.2.4. All of these proofs rely on the existence of 'nice' hash functions that hash all **YES** instances of our sparse language to different values. Chen, Jin, and Williams construct such hash functions in [CJW19, Section 3] using various tools from pseudorandomness. We won't discuss this construction; instead, we'll just state its relevant properties in the following lemma.

Lemma 6.1 ([CJW19, Lemma 3.3])
For all $p, n \in \mathbb{N}$ with $\log n \leq p \leq n^{1/2}$, there exists a hash family $\mathcal{H}_{n,p} = \{h_v \colon \{0,1\}^n \to \{0,1\}^q\}$ with the following properties:

(i) The output length and seed length of each h_v are both $q = ap$ (for a constant $a \in \mathbb{N}$).

(ii) For every $S \subseteq \{0,1\}^n$ with $|S| \leq 2^p$, there exists a seed v such that the values of $h_v(x)$ over all $x \in S$ are all distinct. (We say such a v is a *good seed* for S.)

(iii) Each hash function h_v is linear — for each seed v and index $i \in [q]$, we can write $h_v(x)_i$ as $\langle x, w_{v,i}\rangle$ for some $w_{v,i} \in \{0,1\}^n$.

(iv) There is an algorithm that, given n, p, v, i, and j (for indices $i \in [q]$ and $j \in [n]$), computes the jth bit of $w_{v,i}$ in $p \cdot \mathsf{polylog}(n)$ time and space.

Note that (iv) means that given n, p, and v, we (i.e., an algorithm) can compute the entire length-q string $h_v(x)$ in $np^2 \cdot \mathsf{polylog}(n)$ time and $p \cdot \mathsf{polylog}(n)$ space.

Remark 6.2. The construction of [CJW19] actually works for p all the way up to $n/\mathsf{polylog}(n)$. But for our purposes, p will always be a parameter representing the sparsity of our language (e.g., it'll be n^β for Theorems 1.9 and 1.10), so we only really need the case where p is a small power of n (or potentially even smaller, for the theorems regarding MCSP and MKtP).

§6.1 Magnification for sparse NP languages with formulas

In this subsection, we prove Theorem 1.9.

Theorem 1.9 ([CJW19, Theorem 1.1])
Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that there is a 2^{n^β}-sparse language $L \in \mathsf{NP}$ with $L \notin \mathsf{Formula\text{-}XOR}[n^{1+\varepsilon}]$. Then $\mathsf{NP} \not\subseteq \mathsf{Formula}[n^k]$ for all $k \in \mathbb{N}$.

For this proof, given any language $L \in \mathsf{NP}$ and any $\beta \in (0, \frac{1}{2})$, we define the following auxiliary problem as our oracle (note that here we're defining a whole *family* of oracles, rather than just one).

Definition 6.3 ((L, β)-Hash-Match)

- Input: $\langle n, 1^p, v, x^*, i, b\rangle$ where $p = n^\beta$, v is a seed to a hash function $h_v \in \mathcal{H}_{n,p}$ as in Lemma 6.1 (we'll use q to denote ap, so $v \in \{0,1\}^q$), $x^* \in \{0,1\}^q$, $i \in [n]$, and $b \in \{0,1\}$ (n and i are given in binary, with i padded with leading 0's to have the same length as n).
- Decide: whether there exists $y \in \{0,1\}^n$ such that $y \in L$, $h_v(y) = x^*$, and $y_i = b$.

Claim 6.4 — For any $L \in \mathsf{NP}$ and $\beta \in (0, \frac{1}{2})$, we have (L, β)-Hash-Match $\in \mathsf{NP}$.

Proof. We can decide (L, β)-Hash-Match by nondeterministically guessing $y \in \{0,1\}^n$ and then nondeterministically checking that $y \in L$ (which we can do because $L \in \mathsf{NP}$), computing $h_v(y)$ using the polynomial-time algorithm given by Lemma 6.1(iv) to check that $h_v(y) = x^*$, and checking that $y_i = b$.

This algorithm clearly takes time $\mathsf{poly}(n)$. And the length of our input is at least n^β (and $\beta > 0$ is fixed, so n is a fixed polynomial in n^β), so this algorithm takes polynomial time in its input length. □

We'll then construct oracle formulas with the following parameters.

Lemma 6.5

Let $\beta \in (0, \frac{1}{2})$, and suppose that $L \in \mathsf{NP}$ is 2^{n^β}-sparse. Then for each n, there is a (L, β)-Hash-Match-oracle **XOR**-formula of size $O(n^{1+\beta})$ making nonadaptive queries of fan-in $O(n^\beta)$ that decides L on inputs of length n.

Remark 6.6. The exact bounds in the following lemma are not very important — for Theorem 1.9 it's enough to bound the size of the formula by $n \cdot \mathsf{poly}(n^\beta)$ and the fan-in by $\mathsf{poly}(n^\beta)$ (in the style of the bounds we've stated for the previous oracle algorithms) — but we state specific bounds in this case because this makes the proof of Theorem 1.9 notationally simpler.

Proof. First, we let $p = n^\beta$ and fix a good seed v (for a hash function $h_v \in \mathcal{H}_{n,p}$) for the set of **YES** instances to L of length n (as in Lemma 6.1(ii)). We then take the formula

$$\bigwedge_{i \in [n]} (L, \beta)\text{-Hash-Match}(\langle n, 1^p, v, h_v(x), i, x_i\rangle).$$

Here n, 1^p, v, and i are all hardcoded; h_v is a linear function $\{0,1\}^n \to \{0,1\}^q$ (where $q = ap$, as in Lemma 6.1), so each of the q bits of $h_v(x)$ is a parity function of x and therefore is a valid leaf of a **XOR**-formula; and x_i is a single bit of x and therefore also a valid leaf. So the input to each oracle query consists of $O(\log n + p + q + q + \log n + 1) = O(n^\beta)$ leaves, and we have n such oracle queries.

We'll now explain why this formula works. In words, what we're doing is first hashing x and then asking, for each index $i \in [n]$, whether there is some $y \in \{0,1\}^n$ such that $y \in L$, $h_v(y) = h_v(x)$, and $y_i = x_i$. If x is in L, then the answer is certainly yes for all indices i, as we can take $y = x$. Meanwhile if x is not in L, then there's at most one $y \in L$ with $h_v(y) = h_v(x)$. If there is no such y, then the answer will be no for *every* $i \in [n]$. Meanwhile, if there is such a y, then it has to disagree with x on at least one index (i.e., there must exist some $i \in [n]$ with $y_i \neq x_i$), and the answer will be no for this index i. □

Finally, we'll deduce Theorem 1.9 from this construction.

Proof of Theorem 1.9. Assume that $\mathsf{NP} \subseteq \mathsf{Formula}[n^k]$ for some $k \in \mathbb{N}$. We'll show that for every $\varepsilon > 0$, for every $\beta \in (0, \frac{\varepsilon}{2k})$ and every 2^{n^β}-sparse $L \in \mathsf{NP}$, we have $L \in \mathsf{Formula\text{-}XOR}[n^{1+\varepsilon}]$.

Fix β and L, and for each n, consider the (L, β)-$\mathsf{Hash\text{-}Match}$-oracle **XOR**-formula for L (on inputs of length n) given by Lemma 6.5. Since the problem (L, β)-$\mathsf{Hash\text{-}Match}$ is in NP, the assumption $\mathsf{NP} \subseteq \mathsf{Formula}[n^k]$ allows us to replace each oracle query with a formula of size $O(n^{\beta k})$ in its $O(n^\beta)$ inputs. This multiplies the size of our formula by a factor of at most $O(n^{\beta k})$, so the resulting formula is a **XOR**-formula of size $O(n^{1+\beta+\beta k}) \leq O(n^\varepsilon)$ for L, showing that $L \in \mathsf{Formula\text{-}XOR}[n^{1+\varepsilon}]$. □

§6.2 Magnification for sparse NP languages with small-advice algorithms

In this subsection, we prove Theorem 1.10.

Theorem 1.10 ([CJW19, Theorem 1.2])

Suppose that there is $\varepsilon > 0$ for which there are arbitrarily small $\beta > 0$ such that there is a 2^{n^β}-sparse language $L \in \mathsf{NP}$ that cannot be computed by a $n^{1+\varepsilon}$-time n^ε-space deterministic algorithm with n^ε bits of advice. Then $\mathsf{NP} \not\subseteq \mathsf{Circuit}[n^k]$ for all $k \in \mathbb{N}$.

Proof. As usual, we'll prove this by contrapositive — assume that $\mathsf{NP} \subseteq \mathsf{Circuit}[n^k]$. Our goal is to show that for every $\varepsilon > 0$, for all sufficiently small β and all 2^{n^β}-sparse languages $L \in \mathsf{NP}$, we *can* decide L with a $n^{1+\varepsilon}$-time n^ε-space deterministic algorithm with n^ε bits of advice.

The main idea is that to decide L, we'd like to compute the same quantity

$$\bigwedge_{i \in [n]} (L, \beta)\text{-}\mathsf{Hash\text{-}Match}(\langle n, 1^p, v, h_v(x), i, x_i \rangle) \qquad (\star)$$

from the proof of Lemma 6.5 (where x is our input, n is its length, and $p = n^\beta$), but this time with a low-time low-space algorithm with a small amount of advice instead of an oracle formula. We can include v in our advice, and if we do so then we can compute all the quantities in $\langle n, 1^p, v, h_v(x), i, x_i \rangle$ in low time and space (using Lemma 6.1(iv) to compute $h_v(x)$ efficiently), but then we need a way to implement the oracle calls. We'll do so by including a *circuit* for (L, β)-$\mathsf{Hash\text{-}Match}$ in our advice — all our oracle calls involve calling (L, β)-$\mathsf{Hash\text{-}Match}$ on inputs of a very small, fixed length, so the assumption $\mathsf{NP} \subseteq \mathsf{Circuit}[n^k]$ means this circuit is small enough to fit into our n^ε bits of advice (if β is sufficiently small with respect to ε), and small enough that we can evaluate it (for each of these calls) in low time and space.

To formalize this, let $c \in \mathbb{N}$ be a constant such that given a circuit C with $\mathsf{size}(C) \leq s$ and with at most s inputs, we can evaluate C on any given input string in time $O(s^c)$. (Such c exists because $\mathsf{Circ\text{-}Eval} \in \mathsf{P}$.) We'll take 'sufficiently small' to mean that $\beta \in (0, \frac{\varepsilon}{4kc})$ — we'll show that for any such β and any 2^{n^β}-sparse $L \in \mathsf{NP}$, we can decide L with a $n^{1+\varepsilon}$-time n^ε-space deterministic algorithm with n^ε bits of advice.

Fix L and β, let $p = n^\beta$ and $q = ap$, and let $v \in \{0,1\}^q$ be a good seed (for a hash function $h_v \in \mathcal{H}_{n,p}$) for the set of **YES** instances to L of length n (as in Lemma 6.1). Let ℓ be the common length of $\langle n, 1^p, v, h_v(x), i, x_i \rangle$ over all oracle queries in $(\star)$ (we defined (L, β)-$\mathsf{Hatch\text{-}Match}$ so that this length doesn't depend on i — it's always $\log n + p + q + q + \log n + 1$, which only depends on n), so that $\ell = O(n^\beta)$.

We'll then take our advice (for inputs of length n) to consist of v and the description of a circuit of size at most ℓ^k computing (L, β)-$\mathsf{Hash\text{-}Match}$ on inputs of length ℓ. The size of this circuit is at most $n^{\beta k}$, so its description length is at most $n^{\beta k} \log n < n^\varepsilon$ (ignoring constant factors).

Now our algorithm works as follows: we first compute $h_v(x)$ and write it down (this takes $n^{1+2\beta} \cdot \mathsf{polylog}(n)$ time and $n^\beta \cdot \mathsf{polylog}(n)$ space by Lemma 6.1(iv); both of these are well within bounds). Then we go through the indices $i \in [n]$ one at a time, and for each, we evaluate the (L, β)-$\mathsf{Hash\text{-}Match}$ circuit (given in our advice) on the appropriate input $\langle n, 1^p, v, h_v(x), i, x_i\rangle$ in $(\star)$. At each index we're evaluating a circuit of size at most $n^{\beta k}$, which we can do in time and space $n^{\beta kc} < n^\varepsilon$ (and of course we can reuse the space across different oracle calls); so this lets us compute $(\star)$ in $n^{1+\varepsilon}$ time and n^ε space. □

§6.3 Magnification for Search-MKtP with formulas

We'll now use the ideas in Subsection 6.1 to prove Theorems 1.11 and 1.12 — magnification results for MCSP and MKtP for more restrictive models of computation (specifically, Formula-XOR). Note that $\mathsf{MCSP}[s]$ is $2^{64s\log s}$-sparse and $\mathsf{MKtP}[p]$ is 2^p-sparse, so Theorem 1.9 automatically implies magnification to *fixed*-polynomial lower bounds for both languages (in the regime where s and p are of the form n^β). But we'd ideally like to prove magnification to *arbitrary*-polynomial lower bounds (as in the previous magnification theorems we've seen for MCSP and MKtP). (We'd also like to consider the search versions of these problems, and to allow for general parameters s and p.) It turns out that we can do all of these things, by modifying how we define and implement the auxiliary problem (that we use as an oracle).

We'll begin with MKtP (because the modifications we need to make to the argument in Subsection 6.1 are simpler) — in this subsection, we'll prove the following general version of Theorem 1.12.

Theorem 6.7 ([CJW19, Theorem 1.6])
Suppose there exists $p\colon \mathbb{N}\to\mathbb{N}$ with $\log n \le p \le n^{1/2}$ for all $n\in\mathbb{N}$ and such that $\mathsf{Search\text{-}MKtP}[p]$ cannot be solved by **XOR**-formulas of size $n\cdot\mathsf{poly}(p)$. Then $\mathsf{EXP} \not\subseteq \mathsf{Formula}[\mathsf{poly}]$.

In order to prove this, we'll replace (L,β)-Hash-Match from Subsection 6.1 with the following auxiliary problem (where we essentially specialize the definition of (L,β)-Hash-Match to the case of MKtP, and adapt it for a search problem in the same way as in Section 5).

Definition 6.8 (MKtP-Hash-Match)

- Input: $\langle n, 1^p, v, x^*, i, b, j\rangle$ where:
  - n is an integer given in binary, and $\log n \le p \le n^{1/2}$.
  - v is a seed to a hash function $h_v \in \mathcal{H}_{n,p}$ as in Lemma 6.1 (we'll use q to denote ap, so $v\in\{0,1\}^q$), and $x^*\in\{0,1\}^q$.
  - $i\in[n]$ is also an integer given in binary, and $b\in\{0,1\}$.
  - j is an integer with $1\le j\le p$.
- Output: $\langle M\rangle_j$, where M is the lexicographically first Turing machine such that:
  - $|\langle M\rangle| + \log \mathsf{time}(M) \le p$, and
  - the output y of M has length n and satisfies $h_v(y) = x^*$ and $y_i = b$,

  or 0 if no such M exists (or if the lexicographically first M has $|\langle M\rangle| < j$).

Claim 6.9 — We have $\mathsf{MKtP\text{-}Hash\text{-}Match} \in \mathsf{EXP}$.

Proof. As usual, we can simply enumerate over all 2^p Turing machines M of description length at most p. For each M, we run M for 2^p steps and check that $|\langle M\rangle| + \log\mathsf{time}(M) \le p$, and that its output y meets all the given specifications. Note that for y of length $n\le 2^p$, computing $h_v(y)$ takes time $\mathsf{poly}(n) = 2^{O(p)}$; so this gives an exponential-time algorithm for MKtP-Hash-Match. □

Then we can construct a MKtP-Hash-Match-oracle **XOR**-formula, using essentially the same construction as in Lemma 6.5. (This immediately implies Theorem 6.7.)

Lemma 6.10

For any $\log n \leq p \leq n^{1/2}$, there is a MKtP-Hash-Match-oracle **XOR**-formula of size $n \cdot \mathsf{poly}(p)$ making nonadaptive queries of fan-in $\mathsf{poly}(p)$ that solves Search-MKtP$[p]$ on inputs of length n.

Proof. We define our formula so that for each $j \in [p]$, the jth bit of its output is

$$\bigwedge_{i \in [n]} \mathsf{MKtP\text{-}Hash\text{-}Match}(\langle n, 1^p, v, h_v(x), i, x_i, j\rangle)$$

where v is a good seed for the **YES** instances of MKtP$[p]$ of length n, the quantities n, p, v, and i are all hardcoded, and x_i and each bit of $h_v(x)$ is a leaf of the formula (as in the proof of Lemma 6.5). (This is essentially the same construction as in Lemma 6.5; the fact that we now have p outputs instead of 1 only contributes a factor of p to the size of the formula.)

We'll now check that this formula works. First suppose that x is a **YES** instance of MKtP$[p]$. Then there are no other **YES** instances y of MKtP$[p]$ with $h_v(y) = h_v(x)$ (because we chose v to be a good seed for the **YES** instances of MKtP$[p]$, so they're all hashed to different values). This means that for all i and j, the only way a Turing machine M can meet the two conditions in the definition of MKtP-Hash-Match is if its output is x (because if M satisfies the first condition, i.e., $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$, then its output y has Kt-complexity at most p by definition). Conversely, any M with $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$ which outputs x does satisfy both conditions (and we assumed x is a **YES** instance of MKtP$[p]$, so such M exists). So for all i and j, the oracle call MKtP-Hash-Match$(\langle n, 1^p, v, h_v(x), i, x_i, j\rangle)$ will return $\langle M\rangle_j$ where M is the *lexicographically first* Turing machine with $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$ that outputs x; and so overall, our formula produces the full description of M.

On the other hand, suppose that x is a **NO** instance of MKtP$[p]$. Then there is at most one possible string y of length n that a Turing machine M with $|\langle M\rangle| + \log \mathsf{time}(M) \leq p$ could possibly output that satisfies $h_v(y) = h_v(x)$ (again because any such y is a **YES** instance of MKtP$[p]$, and all **YES** instances hash to different values), and there must exist some index i with $x_i \neq y_i$; for this index i, there can't exist any M meeting the given specifications, so MKtP-Hash-Match$(\langle n, 1^p, v, h_v(x), i, x_i, j\rangle)$ will return 0 for all j. Since for each j we're taking an **AND** over i, this means we'll (correctly) output the all-0's string. □

§6.4 Magnification for Search-MCSP with formulas

Finally, we'll adapt the arguments in Subsection 6.4 to work for MCSP as well, and prove the following general version of Theorem 1.11.

Theorem 6.11 ([CJW19, Theorem 1.6])

Suppose there exists $s\colon \mathbb{N} \to \mathbb{N}$ with $\log n \leq s \leq n^{1/3}$ for all $n \in \mathbb{N}$ and such that Search-MCSP$[s]$ cannot be solved by **XOR**-formulas of size $n \cdot \mathsf{poly}(s)$. Then PSPACE $\not\subseteq$ Formula[poly].

We'll first define an auxiliary problem very similar to MKtP-Hash-Match.

Definition 6.12 (MCSP-Hash-Match)

- Input: $\langle 1^s, v, x^*, z, b, j\rangle$, where:
  - $z \in \{0,1\}^m$ for some $m \leq s$ (we'll use n to denote 2^m), and $b \in \{0,1\}$.
  - v is a length-t seed to a hash function $h_v \in \mathcal{H}_{n,p}$ with $p = 64s\log s$, as in Lemma 6.1 (we'll use q to denote ap), and $x^* \in \{0,1\}^q$.
  - j is an integer with $1 \leq j \leq 64s\log s$.
- Output: $\langle C\rangle_j$, where C is the lexicographically first circuit (with m inputs) with $\mathsf{size}(C) \leq s$ and such that $h_v(\mathtt{tt}(C)) = x^*$ and $C(z) = b$, or 0 if no such circuit exists.

(We use $\mathtt{tt}(C)$ to denote the n-bit truth table of the function computed by C.)

Claim 6.13 — We have $\mathsf{MCSP\text{-}Hash\text{-}Match} \in \mathsf{PSPACE}$.

Proof. We can enumerate over all circuit descriptions $\langle C\rangle$ of length $64s\log s$ in lexicographical order. For each, we can easily check in polynomial space that $C(z) = b$, but we can't write out the entire truth table $\mathtt{tt}(C)$ and hash it — it's simply too big. Instead, we'll use Lemma 6.1(iv) in a more careful way.

We'll compute the bits $h_v(\mathtt{tt}(C))_i$ over all $i \in [q]$ one at a time (and compare them to the corresponding bits of x^*). For each $i \in [q]$, we know we can write

$$h_v(\mathtt{tt}(C))_i = \langle \mathtt{tt}(C), w_{v,i}\rangle$$

for some $w_{v,i} \in \{0,1\}^n$. Since $\mathtt{tt}(C)$ is indexed by strings $x \in \{0,1\}^m$, we'll also think of $w_{v,i}$ as being indexed by such x (and we'll write $w_{v,i}(x)$ to denote the xth coordinate of $w_{v,i}$); then this quantity is

$$h_v(\mathtt{tt}(C))_i = \langle \mathtt{tt}(C), w_{v,i}\rangle = \#\{x \in \{0,1\}^m \mid C(x) = w_{v,i}(x) = 1\} \pmod 2. \qquad (\star\star)$$

But by Lemma 6.1(iv), we can compute $w_{v,i}(x)$ (for any given v, i, and x) in $\mathsf{poly}(p) = \mathsf{poly}(s)$ space, and we can also compute $C(x)$ in $\mathsf{poly}(s)$ space; so by going through the values of $x \in \{0,1\}^m$ one at a time, we can compute $h_v(\mathtt{tt}(C))_i$ in $\mathsf{poly}(s)$ space as well.

This allows us to check all the relevant conditions on C in polynomial space, and therefore to find the lexicographically first C that satisfies them (or that no such C exists). □

Remark 6.14. By a slightly more careful argument, we can in fact see that $\mathsf{MCSP\text{-}Hash\text{-}Match} \in (\Sigma_2\mathsf{P})^{\oplus\mathsf{P}}$ — we existentially guess C and universally guess a lexicographically smaller circuit C', and from $(\star\star)$ we can see that computing each bit of $h_v(\mathtt{tt}(C))$ is essentially a $\oplus\mathsf{P}$ problem. Chen, Jin, and Williams [CJW19] use this observation and several facts about relationships between complexity classes to prove Theorem 1.11 with PSPACE replaced by $\oplus\mathsf{P}$ and PP.

Meanwhile, we can construct a MCSP-Hash-Match-oracle **XOR**-formula using the exact same construction as in Lemma 6.10; this implies Theorem 6.11.

Lemma 6.15

Let $\log n \leq s \leq n^{1/3}$. Then there exists a MCSP-Hash-Match-oracle **XOR**-formula of size $n \cdot \mathsf{poly}(s)$ making nonadaptive queries of fan-in $\mathsf{poly}(s)$ that solves $\mathsf{Search\text{-}MCSP}[s]$ on inputs of length n.

The proof is essentially identical to the one in Lemma 6.10, so we omit it — here we use the fact that if v is a good seed for the **YES** instances of $\mathsf{MCSP}[s]$, then there is at most one possibility for $\mathtt{tt}(C)$ hashing to any given value (under the condition that $\mathsf{size}(C) \leq s$), since any such $\mathtt{tt}(C)$ is a **YES** instance to $\mathsf{MCSP}[s]$.

References

[CJW19] Lijie Chen, Ce Jin, and R. Ryan Williams. Hardness magnification for all sparse NP languages. In *FOCS*, pages 1240–1255, 2019.

[DKKY10] E. Demenkov, A. Kojevnikov, A. Kulikov, and G. Yaroslavtsev. New upper bounds on the Boolean circuit complexity of symmetric functions. *Information Processing Letters*, 110(7):264–267, 2010.

[MMW19] Dylan M. McKay, Cody D. Murray, and R. Ryan Williams. Weak lower bounds on resource-bounded compression imply strong separations of complexity classes. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 1215–1225, Phoenix, AZ, USA, 2019.

[OPS19] Igor C. Oliveira, Ján Pich, and Rahul Santhanam. Hardness magnification near state-of-the-art lower bounds. In *34th Computational Complexity Conference*, CCC 2019, pages 27:1–27:29, New Brunswick, NJ, USA, 2019.

[OS18] Igor C. Oliveira and Rahul Santhanam. Hardness magnification for natural problems. In *59th IEEE Annual Symposium on Foundations of Computer Science*, FOCS 2018, pages 65–76, Paris, France, 2018.

[Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1731, 1996.

[Tal16] Avishay Tal. The bipartite formula complexity of inner-product is quadratic. *Electronic Colloquium on Computational Complexity*, 23(181), 2016.