Connecting Perebor Conjectures: Towards a Search to Decision Reduction for Minimizing Formulas

Rahul Ilango
Massachusetts Institute of Technology, Cambridge, MA, USA
rilango@mit.edu

Abstract

A longstanding open question is whether there is an equivalence between the computational task of determining the minimum size of any circuit computing a given function and the task of producing a minimum-sized circuit for a given function. While it is widely conjectured that both tasks require "perebor," or brute-force search, researchers have not yet ruled out the possibility that the search problem requires exponential time but the decision problem has a linear time algorithm.

In this paper, we make progress in connecting the search and decision complexity of minimizing *formulas*. Let MFSP denote the problem that takes as input the truth table of a Boolean function f and an integer size parameter s and decides whether there is a formula for f of size at most s. Let Search-MFSP denote the corresponding search problem where one has to output some optimal formula for computing f.

Our main result is that given an oracle to MFSP, one can solve Search-MFSP in time polynomial in the length N of the truth table of f and the number t of "near-optimal" formulas for f, in particular $O(N^6t^2)$-time. While the quantity t is not well understood, we use this result (and some extensions) to prove that given an oracle to MFSP:

- there is a deterministic $2^{O(\frac{N}{\log\log N})}$-time oracle algorithm for solving Search-MFSP on all but a $o(1)$-fraction of instances, and
- there is a randomized $O(2^{.67N})$-time oracle algorithm for solving Search-MFSP on *all* instances.

Intriguingly, the main idea behind our algorithms is in some sense a "reverse application" of the gate elimination technique.

2012 ACM Subject Classification Theory of computation → Circuit complexity; Theory of computation → Problems, reductions and completeness

Keywords and phrases minimum circuit size problem, minimum formula size problem, gate elimination, search to decision reduction, self-reducibility

Digital Object Identifier 10.4230/LIPIcs.CCC.2020.31

Funding During this work the author was supported by an Akamai Presdential fellowship.

Acknowledgements I want to thank Eric Allender, Mathew Katzman, Aditya Potukuchi, Michael Saks, Rahul Santhanam, and Ryan Williams for many helpful discussions regarding this work.

1 Introduction

In his fascinating historical account, Trakhtenbrot [20] describes the early developments of the Russian cybernetics program. Beginning in the 1950s, this program was largely driven by a desire to understand the necessity of "perebor," or brute-force, in solving various problems related to complexity minimization. What Trakhtenbrot calls "Task 1" in [20] is an analogue[1] of what is now commonly referred to as the Minimum Circuit Size Problem, MCSP. In his article, Trakhtenbrot delineates two versions of "Task 1": an "existential version," where

[1] We say analogue since Task 1 was defined in the slightly different model of switching circuits.

35th Computational Complexity Conference (CCC 2020).
Editor: Shubhangi Saraf; Article No. 31; pp. 31:1–31:35
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

COMPUTATIONAL COMPLEXITY CONFERENCE

given a Boolean function f one must compute the minimum number of gates needed in a circuit computing f, corresponding to MCSP and a "constructive version," where one must produce such an optimal circuit for f, corresponding to Search-MCSP.

Both versions were conjectured to require "perebor," or brute-force to solve. However, while it is clear that if perebor is required for MCSP then perebor must also be required for Search-MCSP, it is a longstanding open question (since at least 1999 [10]) to prove a reverse implication: that is, to show that if Search-MCSP requires brute-force to solve, then MCSP requires brute-force.

Indeed, this question is closely related to another major open question surrounding MCSP: is MCSP NP-complete? Despite being an open problem since the discovery of NP-completeness[2] in the 1970s and numerous fascinating papers studying MCSP, we still know little about the computational complexity of MCSP. The problem is known to lie in NP, but even formal evidence supporting or opposing the NP-completeness of MCSP is lacking. This is in contrast to other prominent problems that are believed to be intractable yet are not known to be NP-complete (such as integer factorization or the discrete logarithm[3]).

However, a remarkable line of research demonstrates that a proof that MCSP is NP-complete would have significant ramifications. For example, Murray and Williams [14] show that it would imply the breakthrough complexity separation $\mathsf{EXP} \neq \mathsf{ZPP}$, and Hirahara [8] shows that it implies a worst-case to average case reduction for NP (if the hardness holds for an approximate version of MCSP).

An NP-completeness proof for MCSP would also resolve the "search versus decision" question mentioned at the beginning of this paper. In particular, since SAT is known to have a polynomial-time search to decision reduction, MCSP being NP-complete would imply that MCSP would also have a polynomial-time search to decision reduction. Hence, the time complexity of computing MCSP and Search-MCSP would be equivalent up to a polynomial.

Because of this, Kabanets and Cai observed that finding a search to decision reduction for MCSP is, in fact, a *necessary* step to showing that MCSP is NP-complete, and left finding such a reduction as an open question. Indeed, it is a bit unnerving (at least to the author) that researchers have not yet ruled out the possibility that MCSP has a linear-time algorithm but solving Search-MCSP requires exponential-time! The present work was born out of a motivation to (at least partially) mediate this large gap.

Alas, while we fail to improve the status of this question for MCSP, we make considerable progress in connecting the search and decision complexity of the analogous *Formula* Minimization Problem, MFSP.

1.1 Prior Work

In light of the numerous research papers studying MCSP and its variants, we do not attempt to survey the full body of literature but rather concentrate on those works related to search to decision reductions and MFSP. We point a reader interested in a more detailed overview to Allender's excellent new survey [1] and the references therein.

Search to decision reductions for MCSP. There are two main prior works for search to decision reductions for MCSP-like problems. Both provide algorithms that find approximately optimal circuits that are efficient as long as MCSP has efficient algorithms. Interestingly,

[2] [4] cites a personal communication from Levin that he delayed publishing his initial NP-completeness results in hopes of showing MCSP is NP-complete.

[3] Intriguingly, it is known [18, 2] that both of these problems reduce to MCSP under randomized reductions!

both algorithms require that MCSP actually has efficient algorithms and seemingly fail if they are "only" provided oracle access to MCSP (the reason is that the approximately optimal circuit that these algorithms output actually include a small MCSP circuit within them).

The first prior work is a celebrated paper by Carmosino, Impagliazzo, Kabanets, and Kolokolova [6] that establishes connections between algorithms for MCSP-like problems and PAC-learning of circuits. In their paper, they show the following theorem.

▶ **Theorem 1** (Carmosino, Impagliazzo, Kabanets, and Kolokolova [6]). *Suppose* MCSP $\in$ BPP. *Then there is a randomized polynomial-time algorithm that, given the truth table of a function f with n-bit inputs, outputs a circuit C of size at most* $\mathsf{poly}(s)$ *such that $C(x) = f(x)$ for all but a $\frac{1}{\mathsf{poly}(n)}$ fraction of inputs x, where s is the minimum size of any circuit computing f.*

Building on [6], Hirahara [8] proved a breakthrough worst-case to average-case reduction for an approximation version of MCSP. In said paper, Hirahara shows the following theorem.

▶ **Theorem 2** (Hirahara [8]). *Suppose for some $\epsilon > 0$ that one can approximate* MCSP *to within a factor of $N^{1-\epsilon}$ in randomized polynomial-time (where N is the length of the truth table). Then there is some $\epsilon' > 0$ such that, given a length-N truth table for computing f, one can, in randomized polynomial-time, output a circuit for computing f (exactly) whose size is within a $N^{1-\epsilon'}$ factor of being optimal.*

Using similar ideas, Santhanam [19] independently obtained a comparable search-to-decision reduction (with somewhat better parameters than Theorem 2) for AveMCSP, a natural variant of MCSP where one asks for the smallest circuit computing a function on a 0.9-fraction of the inputs.

We find it interesting that "approximate" search to decision reductions for MCSP have been a building block in these celebrated results. It seems to suggest that further exploring the interplay between the search and decision versions of MCSP could be a fruitful direction.

Hardness of MFSP. As with MCSP, we have good reason to believe that MFSP is intractable, since it is in some sense "hard" for cryptography computable in NC^1.

▶ **Theorem 3** (Razborov and Rudich [17], Kabanets and Cai [10]). *If* MFSP $\in$ P*, then there are no pseudorandom function generators computable in* NC^1*.*

Allender, Koucký, Ronneburger, and Roy [4] build on this connection to show that MFSP is hard to approximate if factoring Blum integers is intractable.

Despite the strength of this cryptographic hardness connection, we know very little about the complexity of MFSP unconditionally. Indeed, part of the difficulty is that it seems difficult to design reductions that make use of an MFSP (or MCSP) oracle, since we do not understand the model of formulas (or circuits) very well. Until very recently [7], it was even open whether MFSP was in $\mathsf{AC}^0[2]$!

One reason for focusing on MFSP is that one might expect it to be an easier problem to analyze than MCSP since formulas are somewhat better understood than circuits. In support of this intuition, we know that the formula minimization problem for DNFs and DNF $\circ$ XOR formulas are NP-complete [13, 9] and that the natural Σ_2 variant of MFSP is complete for Σ_2 [5].

However, counter to this intuition, there are some cases in which it has been more difficult to prove hardness for MFSP than for MCSP. While it is known that MCSP is hard for SZK under randomized reductions [3], it remains open to prove such a result for MFSP. We take this as further evidence of the subtleties involved in designing reductions for MFSP.

1.2 Our Results

In contrast to prior results, we examine the case of having to *exactly* solve Search-MFSP, that is, producing an exactly optimal (instead of approximately optimal) formula.

We define MFSP over the model of DeMorgan formulas (formulas with AND and OR gates) where the size of a formula is the number of leaf nodes in its binary tree. Our main results are robust to changes in the model however. In particular, unless otherwise stated, all our results also extend to the case when gates are from the full binary basis $\mathbb{B}_2$ and to the case when the notion of size is the number of wires or the number of gates.

Our main result is to show that one can efficiently find an optimal formula for a given function f using an oracle to MFSP when f has a small number of "near-optimal formulas" (we say what this means after our theorem statement).

▶ **Theorem 4** (also Theorem 33). *There is an algorithm solving* Search-MFSP *using an oracle to* MFSP *that given a length-N truth table of a function f runs in time $O(N^6t^2)$ where t is the number of "near-optimal" formulas computing f.*

Defining "near-optimal" formulas. We now define what we mean by "near-optimal" formulas. Let $\mathsf{L}(f)$ denote the minimum size of any formula computing f. We say a formula φ is a *near-optimal formula* for $f : \{0,1\}^n \to \{0,1\}$ if φ has size at most $\mathsf{L}(f) + n + 1$.

Furthermore, in counting the number of near-optimal formulas, we consider formulas that are isomorphic as labelled binary trees to be the same formula. This avoids counting many trivially equivalent formulas as distinct near-optimal formulas. See Section 2.2 for a precise definition.

Bounding the number of near-optimal formulas. Unfortunately, we do not understand the quantity t in Theorem 4 very well. However, using the nearly tight upper bounds by Lozhkin [11] on the maximum formula size required to compute an n-input function, we get that with high probability a random function on n-inputs has at most

$$2^{O(\frac{N}{\log\log N})}$$

many near-optimal formulas where $N = 2^n$.

Thus, we have the following corollary.

▶ **Corollary 5** (also Corollary 34). *There is an algorithm A for solving* Search-MFSP *on all but a $o(1)$ fraction of instances that runs in time $2^{O(\frac{N}{\log\log N})}$ using an oracle to* MFSP.

Corollary 5 has a nice interpretation with respect to the perebor conjecture. The queries algorithm A (run on a truth table input of length N) makes to its MFSP-oracle can be answered using a deterministic brute-force algorithm in time $2^{(1+o(1))N}$. In particular, the queries A makes are of length at most $2N$ and have complexity at most $(1 + o(1))\frac{N}{\log\log N}$. On the other hand, the naive brute-force algorithm for Search-MFSP on an input of length N runs in time $2^{(1+o(1))N}$. Thus, we have the following further corollary.

▶ **Corollary 6** (Informal). *If the brute-force algorithm for* Search-MFSP *is essentially optimal on average, then the brute-force algorithm for* MFSP *is essentially optimal in the worst-case on a large subset of instances (in particular queries of length $2N$ with complexity at most $(1 + o(1))\frac{N}{\log\log N}$).*

It would be nice to improve the running-time of the algorithm in Corollary 5. The bound that $t \leq 2^{O(\frac{N}{\log\log N})}$ for a random function hardly seems tight. In fact, in the setting of Kolmogorov complexity, one can prove that a random string of length N has only $\mathsf{poly}(N)$

many near-optimal descriptions with high probability (this is because the worst-case upper bound for Kolmogorov complexity is much tighter than the one for formulas). If we could prove an analogous result for formulas, then Corollary 5 would give a polynomial-time search to decision reduction for a random function!

Solving Search-MFSP in the worst-case. We also give a reduction that shows that even in the worst-case, one can get exponential savings over the brute-force algorithm for Search-MFSP by using a MFSP-oracle. In light of Theorem 4, a natural approach is split into two cases:

- If there are a lot of near-optimal formulas for f, then just guess random formulas and see if they compute f.
- If there are not a lot of near-optimal formulas for f, then run the algorithm in Theorem 4.

However, this approach will only be able to output a near-optimal formula for computing f, and we desire to solve Search-MFSP exactly.

We manage to overcome this issue and prove the following theorem.

▶ **Theorem 7** (also Theorem 41). *There is a randomized algorithm for solving* Search-MFSP *using an oracle to* MFSP *that runs in* $O(2^{.67N})$ *time on instances of length* N*.*

By examining the queries that this algorithm makes to MFSP, we get the following consequence regarding the perebor conjecture.

▶ **Corollary 8** (Informal). *If brute-force is essentially optimal for solving* Search-MFSP*, then any algorithm solving* MFSP *can give at most an* ϵ *power speed up over the brute-force algorithm where* $\epsilon = \frac{1}{7}$*.*

A bottom-up approach for DeMorgan formulas. All of the results mentioned so far are proved by building an optimal formula for a function in a "top-down" way (i.e. starting from the output gate and working its way down to the tree leafs). It is natural to wonder if a "bottom-up" approach could also work.[4]

Indeed, we give such a bottom-up reduction for solving Search-MFSP using an oracle to MFSP that is efficient on average. Unfortunately, the guarantees we prove on the running time on this bottom-up algorithm are weaker than the guarantees provided in Theorem 4. Moreover, the proof of correctness for the algorithm requires our formulas to be DeMorgan formulas and not, say, $\mathbb{B}_2$ formulas. Still, we include this result because we think the algorithm is interesting and because it makes use of the following lemma (which is the part where DeMorgan formulas are crucial) that may be of independent interest. Roughly speaking, the lemma shows that optimal DeMorgan formulas must not have too large depth.

▶ **Lemma 9** (also Lemma 53). *Suppose* φ *is an optimal DeMorgan formula for a function on* n*-inputs. Then the depth of* φ *is at most* $O(\frac{2^n}{n \log n})$*.*

1.3 Techniques and Proof Overviews

The top-down approach. As mentioned earlier, our reduction works in a top-down manner. We formalize this as follows. For any Boolean function f on n-inputs, we define the set OptSubcomps(f) to consist of elements of the form $\{g, \triangledown, h\}$ – where $g, h : \{0,1\}^n \to \{0,1\}$ and $\triangledown \in \{\wedge, \vee\}$ – satisfying the property that there exists an optimal formula φ for computing f such that $\varphi = \varphi_g \triangledown \varphi_h$ where φ_g and φ_h are subformulas computing g and h respectively.

[4] The idea that a bottom-up approach could also be an efficient way to solve Search-MFSP was given to me by Ryan Williams.

We can naturally define the Decomposition Problem, denoted $\mathsf{DecompProblem}$ as follows:

- **Given:** a non-trivial[5] function f,
- **Output:** some element of $\mathsf{OptSubcomps}(f)$.

Our two main reductions work by solving the $\mathsf{DecompProblem}$. It is easy to show that one can solve $\mathsf{Search\text{-}MFSP}$ efficiently by recursively calling an $\mathsf{DecompProblem}$ oracle to build an optimal formula gate-by-gate from top to bottom. (See Theorem 20 for details.)

Thus, we now focus on trying to solve $\mathsf{DecompProblem}$.

A high level approach to solving DecompProblem. Our two top-down reductions will use a similar approach to solving $\mathsf{DecompProblem}$. (Actually, our worst-case reduction will use three different approaches, but this will be one of them.)

1. Find an efficient "test" that functions in[6] an optimal subcomputation of f pass, but not too many other functions pass.
2. Efficiently build the (not too long) list *Candidates* of functions that pass the "test."
3. Iterate through all pairs of functions in *Candidates* and each possible gate, and check if this constitutes an element of $\mathsf{OptSubcomps}(f)$.

We first describe how we do Item 3 since it is simpler and then describe our "test" for Item 1. Our method for Item 2 will be different in both reductions.

Item 3: checking membership in OptSubcomps(f). Given access to a MFSP oracle it is actually very easy to check whether some $\{g, \triangledown, h\}$ is an element of $\mathsf{OptSubcomps}(f)$ or not. In Lemma 21 we observe that $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$ if and only if $f(x) = g(x) \triangledown h(x)$ for all x and $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$.

Item 1: the Select[f, g] test. The idea for our "test" is based on the gate elimination technique and the implications gate elimination has on the $\mathsf{Select}[\cdot,\cdot]$ function defined as follows. Given functions $f, g : \{0,1\}^n \to \{0,1\}$, we define $\mathsf{Select}[f,g] : \{0,1\}^n \times \{0,1\} \to \{0,1\}$ by

$$\mathsf{Select}[f,g](x,z) = \begin{cases} f(x) & \text{, if } z = 0 \\ g(x) & \text{, if } z = 1. \end{cases}$$

Our test for whether g might be part of an optimal subcomputation for f will be whether the quantity

$$\mathsf{L}(\mathsf{Select}[f,g]) - \mathsf{L}(f)$$

is small – in particular, no more than a parameter C. The exact value of C will depend on the reduction (we use this test in all three of our reductions with a different value for C), but to give a reader some idea, C will be an element of $\{1, n+2, 10 \cdot \frac{2^n}{n}\}$ where n is the number of input bits f takes.

[5] here by non-trivial we mean a function that cannot be computed by a formula of size one

[6] In case it is not clear, we say a function g is *in an optimal subcomputation* for f if there exists a gate $\triangledown$ and function h such that $\{g, \triangledown, h\}$ is an element of $\mathsf{OptSubcomps}(f)$.

Now, we needed our test to have two properties:

- Property 1: any function that is in an optimal subcomputation for f must pass this test, and
- Property 2: this test does not accept too many other functions.

With regards to Property 1, we show in Lemma 23 that if $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$, then $\mathsf{L}(\mathsf{Select}[f, g]) \leq \mathsf{L}(f) + 1$ and $\mathsf{L}(\mathsf{Select}[f, h]) \leq \mathsf{L}(f) + 1$.

We can give the relatively straightforward proof that $\mathsf{L}(\mathsf{Select}[f, g]) \leq \mathsf{L}(f) + 1$ here. Suppose that $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$. To avoid some case analysis, assume that $\triangledown = \wedge$. Then there exists an optimal formula $\varphi = \varphi_g \wedge \varphi_h$ such that φ_g computes g and φ_h computes h. Then the formula $\varphi_g(x) \wedge (\varphi_h(x) \vee z)$ computes $\mathsf{Select}[f, g](x, z)$ and has size $\mathsf{L}(f) + 1$.

For Property 2, our test must be such that the set of all functions q satisfying

$$\mathsf{L}(\mathsf{Select}[f, q]) - \mathsf{L}(f) \leq C$$

is not too large. In Lemma 24, we show that the number of such q is bounded by

$$O(t \cdot 2^{C-1} N \log N)$$

where N is the length the truth table of f and t is the number of distinct formulas (modulo an isomorphism between formulas defined in Section 2.2) computing f of size $\mathsf{L}(f) + C - 1$. (In the case that $C = n + 2$, t is the number of "near-optimal" formulas discussed earlier in Section 1.2.)

The intuition behind this proof is to use gate elimination. In more detail, if φ is a formula of size $\mathsf{L}(f) + C$ computing $\mathsf{Select}[f, g]$, then we can set $z = 0$ in φ and eliminate between one and C gates from φ to obtain a new formula φ' of size at most $\mathsf{L}(f) + C - 1$ computing f. Hence, we can describe φ (and hence g) by first describing φ' (a small-ish formula for f) and the gates that need to be added back to φ' in order to obtain φ.

While this intuition is relatively straightforward, the proof itself is surprisingly tedious. In particular, the intuition, as stated, only gives a bound with a N^C factor dependence on C. To achieve the stated bound with a 2^C factor dependence on C requires some details. Moreover, this dependence on C is important since a N^C dependence would make Theorem 4 have a quasipolynomial dependence on t instead of a polynomial dependence.

Our top-down deterministic reduction. We now outline how the deterministic algorithm in Theorem 4 works to solve $\mathsf{DecompProblem}$ on an input f.

We have already introduced the some of the ideas for the algorithm in Theorem 4. In detail, let $BestFunctions$ be the set of functions that are in an optimal subcomputation of f. Let $GoodFunctions$ denote the set of functions g that pass the test $\mathsf{L}(\mathsf{Select}[f, g]) - \mathsf{L}(f) \leq n + 2$ (for this algorithm we set $C = n + 2$). From our previous discussions, we know that the size of $GoodFunctions$ can be bounded by a quantity related to the number of near-optimal formulas for f, and we know that $GoodFunctions$ contains all the functions in $BestFunctions$.

Later we explain how to construct the list $GoodFunctions$. Note though that once the list $GoodFunctions$ is constructed, we can then iterate through all pairs of functions in $GoodFunctions$ and efficiently check if they yield an optimal subcomputation, as we discussed previously.

Hence, the missing piece is to efficiently enumerate the elements of $GoodFunctions$. In fact, we do not quite need to enumerate *all* the elements of $GoodFunctions$. It suffices to enumerate a subset, that we call $Candidates$, of $GoodFunctions$ that contains all the elements of $BestFunctions$. Informally, one can think of the $Candidates$ subset as a set of "good enough functions."

The key observation is as follows. If q is a function on n-inputs and one defines the truth table $T_{q,i}$ of length 2^n that is equal to q on its first i bits and equals one on the remaining bits, then

$$\mathsf{L}(T_{q,i}) \leq \mathsf{L}(q) + n + 1$$

since one can compute $T_{q,i}$ by computing q, computing whether the input is greater than i, and ORing these two values. The $\mathsf{Select}[\cdot,\cdot]$ function actually respects this observation in a nice way. In particular, since functions g in *BestFunctions* satisfy the stronger property that $\mathsf{L}(\mathsf{Select}[f,g]) - \mathsf{L}(f) \leq 1$, one can show that if $g \in \mathit{BestFunctions}$, then

$$\mathsf{L}(\mathsf{Select}[f, T_{g,i}]) \leq \mathsf{L}(f) + n + 2$$

for all i. In other words, if $g \in \mathit{BestFunctions}$, then $T_{g,i}$ is in *GoodFunctions* for all i.

Using this fact, we can construct a subset *Candidates* of *GoodFunctions* that contains all the elements of *BestFunctions* by bit-by-bit extending a set of prefixes *PartialCandidates* that pass our test (and prefixes of functions in *BestFunctions* do pass our test) until these prefixes become full functions.

In more detail, we start with a set *PartialCandidates* that initially only contains the empty prefix. While *PartialCandidates* is non-empty, we remove a prefix γ from it and try to extend it by one bit. That is, for each bit $b \in \{0,1\}$, we consider γ_b obtained by appending b to γ. We then see if the prefix γ_b "passes our test" by seeing if the truth table T_{γ_b}, obtained by padding γ_b with ones until it has length 2^n, has the property

$$\mathsf{L}(\mathsf{Select}[f, T_{\gamma_b}]) \leq \mathsf{L}(f) + n + 2.$$

If so, we either add γ_b to *Candidates* or back to *PartialCandidates* depending on whether the string γ_b is of length 2^n or not. We continue until *PartialCandidates* is empty. The full details can be found in Algorithm 2.

Our top-down randomized worst-case reduction. The algorithm in Theorem 7 uses three different strategies for finding an optimal subcomputation in the worst-case using an oracle to MFSP. We give a a rough overview of each of these three parts.

Suppose the input to the algorithm is a function f on n-inputs. First, the algorithm picks $2^{2N/3}$ random formulas of size $\mathsf{L}(f)$ and checks if any of these formulas compute f. If so, we are done. Otherwise, we know that the number of optimal formulas for f cannot be too large (in particular, is upper bounded by roughly $2^{N/3}$ with high probability).

In the second part, we construct a set of candidate functions that pass a test. The guarantee on the number of optimal formulas from the previous step ensures that the size of the set

$$\{g : \mathsf{L}(\mathsf{Select}[f,g]) \leq \mathsf{L}(f) + 1\}$$

is bounded by $O(2^{N/3})$, and we know that all functions that are in an optimal subcomputation for f are in this set. Hence, what we would like to do is enumerate the functions in this set, however, the author does not know how to do this efficiently. Instead, we examine the subset of functions in this set that have not too large complexity. That is, we iterate through all functions with complexity at most $\frac{2}{3} \cdot \frac{2^n}{\log n}$ and build

$$\mathit{Candidates} = \{g : \mathsf{L}(\mathsf{Select}[f,g]) - \mathsf{L}(f) \leq 1 \text{ and } \mathsf{L}(g) \leq \frac{2}{3} \cdot \frac{2^n}{\log n}\}.$$

This takes time $O(2^{2N/3})$. We then try to find a pair of functions in *Candidates* that form an optimal subcomputation.

If we succeed, we are done. Otherwise, we know that there exists an optimal subcomputation $\{g, \triangledown, h\}$ of f where h has complexity greater than $\frac{2}{3} \cdot \frac{2^n}{\log n}$. This also implies that g has complexity at most $(1 + o(1))\frac{2^n}{3 \log n}$ since $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ and $\mathsf{L}(f) \leq (1 + o(1))\frac{2^n}{\log n}$.

In the third part, we look for such an "unbalanced" subcomputation as follows. We iterate through each g in *Candidates* with complexity at most $(1 + o(1))\frac{2^n}{3 \log n}$ and each $\triangledown \in \{\wedge, \vee\}$ and try to find a matching h by considering each h satisfying $f = g \triangledown h$. We argue that this is efficient because the the set of h satisfying the constraint that $f = g \triangledown h$ is not too large (in particular, of size at most $2^{N/3}$). The reason for why this set must be small is that the constraint that $f = g \triangledown h$ actually forces many of the values of h to a fixed zero/one value. Indeed, we argue that a large number of values must be "forced," since if only a small number of values of h were "forced," then a theorem of Pippenger [16] ensures that there would be a function h of too small complexity (smaller than $\frac{2}{3} \cdot \frac{2^n}{\log n}$) that satisfied $f = g \triangledown h$, which would contradict that fact that the second part of the algorithm failed.

A bottom-up approach. Our final algorithm takes a different approach than our previous reductions, working bottom-up instead of top-down. The basic idea of the bottom-up approach is as follows. Begin with the set *Candidates* of all functions computed by formulas of size one. For each pair of functions g, h in *Candidates* and each $\triangledown \in \{\wedge, \vee\}$, compute the function $q = g \triangledown h$. Next, see if $g \triangledown h$ is an optimal formula for q using the MFSP oracle. If so, use some one-sided heuristic (that never gives an incorrect NO answer) to test if q is computed by some gate in an optimal formula for f, and add q to *Candidates* if it passes this heuristic. Repeat this process until f is added to *Candidates*, in which case one can construct an optimal formula for f by tracing back through the functions that led to it.

The difficulty in this approach is in finding an appropriate heuristic that significantly prunes the search space of possible candidates. A natural contender for such a heuristic, in light of our previous algorithms, is testing if $\mathsf{L}(\mathsf{Select}[f, q]) - \mathsf{L}(f)$ is small. However, if our only guarantee is that q is computed by some gate in some optimal formula φ for f, the best upper bound we manage to prove for the quantity $\mathsf{L}(\mathsf{Select}[f, q]) - \mathsf{L}(f)$ is linear in the depth of φ.

Luckily, Lemma 53 shows that the depth of φ cannot be too large. In particular, if φ is an optimal DeMorgan formula, then the depth of φ is bounded by $O(\frac{2^n}{n})$ where n is the number of inputs f takes. At a high-level, the proof of this lemma works by saying that if a formula has very large depth, then there are many small subformulas that lie along a path in the binary tree of φ. Because there are so many of these small subformulas, there must be a pair that compute the same function, and this can be used to produce a slightly smaller formula.

Using this lemma, we show that the above bottom-up approach runs in time quadratic in the number of formulas for computing f that are within $O(\frac{2^n}{n})$ of being optimal, additively.

1.4 Open Questions

There are several intriguing questions raised by this work. Looking at our main theorem, the most obvious question is whether one can improve the bound we give on the number of near-optimal formulas for a random function. Our bound hardly seems correct, although its hard to imagine how one could do better with current techniques.

Perhaps an indirect approach could work. Is there any operation one can apply to a function in order to reduce the number of optimal formulas it has? It seems plausible that multiple applications of the $\mathsf{Select}[\cdot, \cdot]$ function might cut down the number of optimal formulas.

Another idea would be to try to modify the heuristic "tests" in our reduction. At their heart, all our "tests" are powered by the gate elimination technique. It seems reasonable that more powerful lower bound techniques (which we indeed do have for formulas) might lead to better heuristics and thus more efficient search-to-decision reductions.

There is also the question this paper began with: can one prove a non-trivial exact search to decision reduction for MCSP? The difficulty in adopting our approach to MCSP is that there are just too many ways to add a single gate to a circuit, which ruins the bounds we get on the number of functions passing our $\mathsf{Select}[f,g]$ test. Is there any way to get around this?

Taking a step back, one can also ask what role relativization plays in the search versus decision question. Can one show that there is an oracle relative to which MCSP or MFSP can be solved in linear time, but the corresponding search problem requires exponential time?

Finally, can one extend Lemma 9 to the case of formulas over $\mathbb{B}_2$ or even just prove a better bound for DeMorgan formulas?

1.5 Organization

In Section 2, we fix our notation and definitions, including our notion of formula isomorphism. In Section 3, we introduce the top-down approach and outline our basic strategy for solving Search-MFSP. Section 4 introduces the $\mathsf{Select}[\cdot,\cdot]$ function and proves bounds on number of functions that pass "tests" related to the $\mathsf{Select}[\cdot,\cdot]$ function. Section 5 gives a deterministic search to decision reduction for MFSP and shows it is efficient on average. Section 6 then gives a reduction that works in the worst case. Finally, Section 7 demonstrates a bottom-up approach for trying to solve Search-MFSP.

2 Preliminaries

For a positive integer n, we let $[n]$ denote the subset of integers $\{1,\dots,n\}$.

2.1 DeMorgan Formulas and Formula Size

Our notion of formulas will be DeMorgan formulas. A DeMorgan formula φ on n-inputs of size s is given by:

- a directed rooted binary tree on the vertex set $[2s-1]$, specified by a subset $E_\varphi \subseteq [2s-1]\times[2s-1]$ of edges, and
- a gate labeling function $\tau_\varphi : [2s-1] \to \{\wedge,\vee\} \cup \{0,1,x_1,\dots,x_n,\neg x_1,\dots,\neg x_n\}$

where τ takes values in $\{\wedge,\vee\}$ on the internal nodes in φ and τ takes values in

$$\{0,1,x_1,\dots,x_n,\neg x_1,\dots,\neg x_n\}$$

on the leaf nodes in φ. The edges in E_φ point from inputs towards outputs. We note that our definition implicitly uses the fact that a binary tree with s leaf nodes has $s-1$ internal nodes. We also note that in our definition we do not need to specify the "left" and "right" child of an internal node since our gate set $\{\wedge,\vee\}$ is made up of symmetric functions. We will define a notion of formula isomorphism in Section 2.2.

We will use the notation $|\varphi|$ to denote the *size* of a formula φ (i.e. the number of leaves in the binary tree underlying φ). Given a Boolean function f, we denote the *minimum formula size* of f by

$$\mathsf{L}(f) = \min\{|\varphi| : \ \varphi \text{ is a formula computing } f\}.$$

We say a formula φ is an *optimal* formula for a Boolean function f, if φ computes f and $|\varphi| = \mathsf{L}(f)$.

We note, however, that all of our results except the ones presented in Section 7 apply equally well to formulas with arbitrary fan-in-two gates (i.e. the formulas over the $\mathbb{B}_2$ basis). Moreover, all our results are hold for other size notions such as gates and wires.

2.2 Optimal Formulas and Formula Isomorphism

Since our results will depend on the number of formulas satisfying certain properties, we will be clear about when exactly we are saying formulas are distinct in our count.

In particular, as we have defined formulas, one can obtain many optimal formulas from a single optimal formula by relabeling the nodes in underlying binary tree.

Thus, it will be useful to define an isomorphism on formulas and only count formulas modulo this isomorphism. In particular, we will define two formulas to be *isomorphic* if they are isomorphic as labelled binary trees.

In order to properly define this, we introduce some notation. If φ is a formula of size s with an underlying edge set E_φ and a labelling function τ_φ and $\sigma : [2s-1] \to [2s-1]$ is a permutation, then we let $\psi = \sigma(\varphi)$ be the formula of size s whose edge set E_ψ is given by

$$E_\psi = \{(\sigma(i), \sigma(j)) : (i, j) \in E_\varphi\}$$

and whose labelling function τ_ψ is given by

$$\tau_\psi(\sigma(i)) = \tau_\varphi(i).$$

We say two formulas φ and φ' are *isomorphic* if $|\varphi| = |\varphi'|$ and there is a permutation σ such that $\varphi' = \sigma(\varphi)$.

From each equivalence class of isomorphic formulas, we pick a single representative that we call the *canonical formula* for that equivalence class. Note that for our purposes we do not need that this canonical formula to be computable, as we will just be using them in our analysis. Then we define $\mathsf{CanonOpt}_k\mathsf{Formulas}(f)$ to be the set of canonical formulas that are optimal for computing f up to an additive k-term. In other words

$$\mathsf{CanonOpt}_k\mathsf{Formulas}(f) = \{\varphi : \varphi \text{ is a canonical formula and } |\varphi| \le \mathsf{L}(f) + k\}.$$

2.3 MFSP, Search-MFSP and Conventions on n and N

We now define the *Minimum Formula Size Problem* denoted MFSP.

▶ **Definition 10** (MFSP). *We define the problem* MFSP *as follows*

- ***Given:*** *a truth table of a Boolean function f and an integer size parameter $s \ge 1$*
- ***Determine:*** *if $\mathsf{L}(f) \le s$.*

We define the search version of MFSP analogously.

▶ **Definition 11** (Search-MFSP). Search-MFSP *is the problem defined as follows:*

- ***Given:*** *a truth table of a Boolean function f*
- ***Output:*** *a formula φ of size $\mathsf{L}(f)$ computing f.*

We note that $\mathsf{MFSP} \in \mathsf{NP}$ since given a minimum-sized formula as a witness, one can check that this indeed computes f efficiently since the truth table of f is provided and every function has a formula of size at most the length of its truth table (see Theorem 13).

When describing a function f that is an input to MFSP, one naturally wants to denote by n two different quantities: the number of variable inputs to a function f and the length of the truth table of f (which is the true input length for MFSP). We maintain the convention throughout this paper that n denotes the input arity of f and $N = 2^n$ denotes the length of the truth table of f.

2.4 Useful Facts About Formulas

We will make use of some basic facts about formulas in our work. First, one can easily bound the number of formulas of size at most s.

▶ **Proposition 12.** *The number of formulas on n-inputs of size at most s is at most* $2^{s\log n(1+o(1))}$

We also know tight upper bounds on the maximum formula complexity of a n-input function.

▶ **Theorem 13** (Lozhkin [11] improving on Lupanov [12])**.** *Let $f : \{0,1\}^n \to \{0,1\}$. Then*

$$\mathsf{L}(f) \leq \frac{2^n}{\log n}(1 + O(\frac{1}{\log n}))$$

Combining the size upper bound in Theorem 13 with the bound on the number of formulas of size s, we get the following proposition.

▶ **Proposition 14** (Random functions have not too many near optimal formulas)**.** *Let n and k be positive integers. Let $N = 2^n$. Assume $k = O(\frac{2^n}{\log^2 n})$. Then all but a $o(1)$-fraction of n-input Boolean functions f satisfy*

$$|\mathsf{CanonOpt}_k\mathsf{Formulas}(f)| = 2^{O(\frac{N}{\log\log N})}.$$

Proof. Theorem 13 say that every n-input function has a formula of size at most

$$\frac{2^n}{\log n}(1 + O(\frac{1}{\log n})).$$

Thus, any formula for computing n-input function that is within an additive k of being optimal has size at most s where

$$s \leq k + \frac{2^n}{\log n}(1 + O(\frac{1}{\log n})) = \frac{2^n}{\log n}(1 + O(\frac{1}{\log n})).$$

Proposition 12 implies that the number of formulas of size at most s is upper bounded by

$$2^{s\log n(1+o(1))} = 2^{N(1+O(\frac{1}{\log\log N}))}.$$

Hence, since there are 2^N Boolean functions on n-inputs, it follows that in expectation a random function has at most

$$2^{O(\frac{N}{\log\log N})}$$

formulas within k of being optimal. The desired claim then follows by an application of Markov's inequality. ◀

We note that the bound given by Proposition 14 is actually counting formulas that are isomorphic to each other as distinct. Unfortunately removing this redundancy does not improve on the bound in Proposition 14. However, the fact that our results rely on the number of distinct formulas up to isomorphism means that there is no obvious obstruction to better bounds being proved and hence to our algorithms being more efficient.

We will also make use of the fact that integer comparison can be implemented by linear-sized formulas.

▶ **Proposition 15** (Small formulas for integer comparison). *Let $y \in \{0,1\}^n$. Let $GrtrThan_y : \{0,1\}^n \to \{0,1\}$ be the function given by $GrtrThan_y(x) = 1$ if and only if $x > y$ in the usual lexicographic order on $\{0,1\}^n$. Then $\mathsf{L}(GrtrThan_y(x)) \leq n$.*

Proof. We work by induction on n. If $n = 1$, then clearly $\mathsf{L}(GrtrThan_y) = 1$ (either it is 0 if $y = 1$ or it equals x if $y = 0$).

Now suppose $n > 1$. Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ denote the bits of x and y respectively where x_1 and y_1 denotes the highest order bit. Let $x', y' \in \{0,1\}^{n-1}$ be given by $x' = x_2 \ldots x_n$ and $y' = y_2 \ldots y_n$ respectively.

Now, $x > y$ if and only if one of the following two statements is true:

- $x_1 > y_1$, or
- $x_1 = y_1$ and $x' > y'$.

Since $x_1, y_1 \in \{0,1\}$, this is equivalent to

$$x > y \iff (x_1 > y_1) \vee (x' > y').$$

By induction this means $\mathsf{L}(GrtrThan_y) \leq 1 + n - 1 = n$. ◀

2.5 Partial Functions and their Formula Size

Partial functions will be a crucial building block in our reductions. A *partial Boolean function* is a function $\gamma : \{0,1\}^n \to \{0,1,?\}$ for some integer $n \geq 1$. We denote partial functions using Greek letters such as γ and μ, although sometimes we resort to the Roman alphabet with a ? subscript such as $h_?$.

In contrast, we say a Boolean function $f : \{0,1\} \to \{0,1\}$ is *total* Boolean function (though we allow for a partial Boolean function to indeed be total).

We say a total Boolean function g *agrees* with a partial Boolean function γ if

$$\gamma(x) \in \{0,1\} \implies \gamma(x) = g(x).$$

One can naturally define the minimum formula size of a partial Boolean function γ as follows

$$\mathsf{L}(\gamma) = \min\{\mathsf{L}(g) : g \text{ is a total function that agrees with } \gamma\}.$$

The following theorem regarding the formula complexity of partial functions will be useful in our randomized worst-case reduction.

▶ **Theorem 16** (Pippenger [16]). *Let $\gamma : \{0,1\}^n \to \{0,1,?\}$ be a partial function. Let $p_? = \frac{|\gamma^{-1}(?)|}{2^n}$. Then,*

$$\mathsf{L}(\gamma) \leq (1 + o(1)) \cdot (1 - p_?) \frac{2^n}{\log n}.$$

3 The Top-Down Approach

Our two main reductions both take a "top-down" approach to finding an optimal formula. That is, given a function f, they try to find functions g and h such that g and h are the two functions fed into the final output gate in an optimal formula for f and then recursing.

This is formalized as follows.

▶ **Definition 17** (Optimal Subcomputations Set). *Let $f : \{0,1\}^n \to \{0,1\}$. We define the set of* optimal subcomputations *for f, denoted* OptSubcomps(f)*, as follows.*

Let $g, h : \{0,1\}^n \to \{0,1\}$ be Boolean functions of the same arity as f and $\triangledown \in \{\wedge, \vee\}$. Then $\{g, \triangledown, h\} \in$ OptSubcomps(f) *if and only if there exists an optimal formula $\varphi = \varphi_g \triangledown \varphi_h$ for computing f such that φ_g computes g and φ_h computes h.*

We note that in this definition we are implicitly using that the gate set $\{\wedge, \vee\}$ is symmetric with respect to its inputs.

We say a function g is *in an optimal subcomputation* for f if g is contained in some element of OptSubcomps(f). In other words, g is in an optimal subcomputation for f if there exists an h and $\triangledown$ such that $\{g, \triangledown, h\} \in$ OptSubcomps(f).

It is easy to see that OptSubcomps(f) is almost always non-empty.

▶ **Proposition 18.** *Let $f : \{0,1\}^n \to \{0,1\}$ such that* L$(f) \geq 2$*. Then* OptSubcomps(f) *is non-empty.*

Next, we can define the problem of finding an optimal subcomputation.

▶ **Definition 19** (Decomposition Problem). *The Decomposition Problem,* DecompProblem *is as follows:*

- ***Given:*** *the truth table of a Boolean function f satisfying* L$(f) \geq 2$
- ***Output:*** *some element of* OptSubcomps(f)*.*

It is easy to see that DecompProblem is equivalent to Search-MFSP. DecompProblem can be easily solved with an oracle to Search-MFSP. The following recursive procedure shows the reverse direction.

▶ **Theorem 20** (Search-MFSP reduces to DecompProblem). *There is a deterministic $O(N^2)$-time algorithm for solving* Search-MFSP *on inputs of length N given access to an oracle that solve* DecompProblem *on instances of length N.*

Proof. The pseudocode for this reduction is written in Algorithm 1.

Algorithm 1 Reduction from Search-MFSP to DecompProblem.

```
procedure FindOptFormula($f$)
▷ Given the length-$N$ truth table of a function $f$ that takes $n$-inputs and oracle access to DecompProblem return an optimal formula for $f$.
    if there exists a size one formula $\varphi$ computing $f$ then
        return $\varphi$.
    end if
    Let $\{g, \triangledown, h\}$ be the output returned by the oracle DecompProblem($f$).
    Recursively compute the formula $\varphi_g \leftarrow FindOptFormula(g)$.
    Recursively compute the formula $\varphi_h \leftarrow FindOptFormula(h)$.
    return the formula given by $\varphi_g \triangledown \varphi_h$.
end procedure
```

The correctness of this algorithm is easy to see as long as one is able to bound the number of recursive calls the algorithm makes. To see that the number of recursive calls is bounded by $O(N)$, notice that each iteration of the algorithm reveals one more gate in the optimal formula for f. Thus, since L$(f) = O(N)$, we have that there are at most $O(N)$ recursive calls. ◀

Our goal is now to try to solve DecompProblem (i.e. find an element of $\mathsf{OptSubcomps}(f)$) given an oracle to MFSP. Recall from the introduction that our high-level approach is as follows

1. Find an efficient "test" that functions that in an optimal subcomputation of f pass but not too many other functions pass.
2. Efficiently build the (not too long) list *Candidates* of things that pass the test.
3. Iterate through all pairs of elements in *Candidates* and all possible gates, and efficiently check if this yields an element of $\mathsf{OptSubcomps}(f)$.

Item 1 will be the subject of Section 4, Item 2 will be different in our two main reductions, and Item 3 is provided by the next lemma.

▶ **Lemma 21** (Test membership in $\mathsf{OptSubcomps}(f)$ efficiently with MFSP). *Let $f, g, h : \{0,1\}^n \to \{0,1\}$. Then*

$$\{g, \nabla, h\} \in \mathsf{OptSubcomps}(f) \iff f = g \nabla h \text{ and } \mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h).$$

Proof. We prove the forward direction first. Suppose that $\{g, \nabla, h\} \in \mathsf{OptSubcomps}(f)$. Then there exists an optimal formula $\varphi = \varphi_g \nabla \varphi_h$ for computing f such that φ_g computes g and φ_h computes h. Clearly this implies that $f = g \nabla h$.

Moreover, $|\varphi| = |\varphi_g| + |\varphi_h|$. On the other hand, since φ is optimal, we have that $|\varphi| = \mathsf{L}(f)$, $|\varphi_g| = \mathsf{L}(g)$, and $|\varphi_h| = \mathsf{L}(h)$. (Otherwise, one could build a smaller formula for f by replacing φ_g or φ_h with a smaller formula computing the same function.) Hence $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$.

For the reverse direction, suppose that $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ and $f = g \nabla h$. Let φ_g and φ_h be optimal formulas for g and h. Then $\varphi = \varphi_g \nabla \varphi_h$ clearly computes f and has size $\mathsf{L}(f)$. Hence $\{g, \nabla, h\} \in \mathsf{OptSubcomps}(f)$. ◀

4 Using gate elimination to find functions in an optimal subcomputation

Our approach to solving DecompProblem involves finding a "test" that functions in an optimal subcomputation pass but not too many other functions pass. The test will be based off the following function.

▶ **Definition 22** ($\mathsf{Select}[\cdot,\cdot]$). *Let $f, g : \{0,1\}^n \to \{0,1\}$. We define the function $\mathsf{Select}[f, g] : \{0,1\}^n \times \{0,1\} \to \{0,1\}$ by*

$$\mathsf{Select}[f,g](x,z) = \begin{cases} f(x) & , \text{ if } z = 0 \\ g(x) & , \text{ if } z = 1 \end{cases}$$

We emphasize that $\mathsf{Select}[f, g]$ function is only defined when f and g have the same arity.

Now, our "test" will be to see if the quantity

$$\mathsf{L}(\mathsf{Select}[f,g]) - \mathsf{L}(f)$$

is small (how small will depend on our reduction).

Indeed, for functions in an optimal subcomputation, this quantity is exactly one![7]

[7] We will only prove that it is as most one, but the reader can check that if $g \neq f$ that a gate elimination argument actually implies equality.

▶ **Lemma 23.** *Suppose g is in an optimal subcomputation for f. Then*

$$\mathsf{L}(\mathsf{Select}[f,g]) \leq \mathsf{L}(f) + 1.$$

Proof. Since g is in an optimal subcomputation for f, there exists an optimal formula $\varphi = \varphi_g \ \triangledown \varphi_h$ such that φ_g computes g. If $\triangledown = \wedge$, then

$$\varphi_g \wedge (\varphi_h \vee z)$$

is a formula for $\mathsf{Select}[f,g]$ of size $\mathsf{L}(f) + 1$. Otherwise $\triangledown = \vee$. Then

$$\varphi_g \vee (\varphi_h \wedge \neg z)$$

is a formula for $\mathsf{Select}[f,g]$ of size $\mathsf{L}(f) + 1$. ◀

On the other hand, the number of functions that "pass this test" can be upper bounded in terms of $|\mathsf{CanonOpt}_k\mathsf{Formulas}(f)|$.

▶ **Lemma 24.** *Let k be a positive integer. Let $f : \{0,1\}^n \to \{0,1\}$. Assume $\mathsf{L}(f) \geq 2$. Let $TestPassers = \{g : \mathsf{L}(\mathsf{Select}[f,g]) - \mathsf{L}(f) \leq k+1\}$. Then*

$$|TestPassers| \leq O(|\mathsf{CanonOpt}_k\mathsf{Formulas}(f)| \cdot 2^k N \log N)$$

where $N = 2^n$.

Proof. At a high-level the idea is that, given a formula φ of size $\mathsf{L}(f) + k + 1$ for computing $\mathsf{Select}[f,g]$, one can replace the z-leaves in φ with 0-leaves to obtain a formula φ' of size $\mathsf{L}(f) + k + 1$ for computing f with at least one constant leaf. One can then use a careful gate elimination argument to remove precisely one constant leaf from φ' to obtain a formula φ'' that still computes f but has size $\mathsf{L}(f) + k$. On the other hand, one can reverse this process by adding some constant leaf and gate to φ'' and then replacing some subset of the constant leaves by z-leaves.

This gives us a way to describe any g that passes the test, and thus allows us to bound the number of such g. In our bound, the $O(|\mathsf{CanonOpt}_k\mathsf{Formulas}(f)|)$ factor corresponds to the choices for φ'', the $O(N \log N)$ corresponds to the number of ways to add a new constant leaf and gate to φ'' in order to obtain φ', and the $O(2^k)$ factor comes from the number of ways to chose a subset of the (at most k) 0-leaves in φ' into z-leaves.

In detail, we prove this statement by giving a series of injections. At a high-level First, let P denote the set of canonical formulas computing f with size exactly $\mathsf{L}(f) + k + 1$ and with at least one constant-labelled leaf node in the formula.

We will give an injection from $TestPassers$ to $P \times [2^{k+1}]$. Before defining our injection, we will need the following claims and definitions.

▷ Claim 25. Suppose φ computes $\mathsf{Select}[f,g]$ and $f \neq g$, then φ has at least one leaf node labelled by z or $\neg z$.

Proof. If φ does not have any $\{z, \neg z\}$ labelled leaves, then the output of φ does not depend on z. But this contradicts that φ computes $\mathsf{Select}[f,g]$ since $\mathsf{Select}[f,g]$ does depend on the z input because $f \neq g$. ◁

▷ Claim 26. Suppose $\varphi \in P$. Then φ has at most $(k+1)$-many leaf nodes labelled by constants $\{0,1\}$.

Proof. Since $\varphi \in P$, we know that $|\varphi| = \mathsf{L}(f) + k + 1$ and φ computes f. Since $\mathsf{L}(f) \geq 2$, we know that $|\varphi| \geq k + 3$.

If φ had more than $(k + 1)$-many constant labelled leaves, it follows by a standard gate elimination argument (note here it is important that $|\varphi| \geq k + 3$) that there is a φ' that computes the same function as φ such that

$$|\varphi'| < |\varphi| - (k + 1) < \mathsf{L}(f).$$

But then φ would be a formula of size less than $\mathsf{L}(f)$ computing f which is a contradiction. ◁

We will also need the following definitions. Given a formula φ that can take z-variables as input, we define $\mathsf{Substitute}_{z=0}(\varphi)$ to be the formula φ' given by replacing the z-labeled leaves in φ with 0-labels and replacing the $(\neg z)$-labeled leaves in φ with 1-labels.

We note that the $\mathsf{Substitute}_{z=0}$ operation in some sense respects formula isomorphisms.

▷ Claim 27. Let φ be a formula of size s that takes a z-variable as input. Let $\sigma : [2s - 1] \to [2s - 1]$ be a permutation. Then

$$\sigma \circ \mathsf{Substitute}_{z=0}(\varphi) = \mathsf{Substitute}_{z=0} \circ \sigma(\varphi)$$

Proof. The proof is essentially just applying the definition to both sides and seeing that the resulting edge sets and labelling functions are equal. ◁

We can also define a reverse operation to $\mathsf{Substitute}_{z=0}$ as follows. Given a formula φ and a subset S of leaf nodes in φ that are labelled by constants $\{0, 1\}$, define $\mathsf{Unsubstitute}_{z=0}(\varphi', S)$ to be the formula φ given by replacing 0-labeled leaves in S with z-labels leaves and by replacing 1-labeled leaves in S with $(\neg z)$-labels.

Indeed, the following claim whose proof we omit is easy to see.

▷ Claim 28. For all formulas φ there exists a set S such that

$$\varphi = \mathsf{Unsubstitute}_{z=0}(\mathsf{Substitute}_{z=0}(\varphi), S).$$

Being more precise, S is a subset of the leaf nodes in $\mathsf{Substitute}_{z=0}(\varphi)$ that are labelled by constants.

Now we are ready to describe our injection from $TestPassers \to P \times [2^{k+1}]$ on an input $g \in TestPassers$. Since $g \in TestPassers$, there is a φ of size $\mathsf{L}(f) + k + 1$ computing $\mathsf{Select}[f, g]$. Let $\varphi' = \mathsf{Substitute}_{z=0}(\varphi)$. Clearly φ' computes f since φ computes $\mathsf{Select}[f, g]$. Let $\overline{\varphi'}$ denote the canonical formula isomorphic to φ'. Then there exists a permutation σ such that

$$\begin{aligned}\overline{\varphi'} &= \sigma(\varphi') \\ &= \sigma \circ \mathsf{Substitute}_{z=0}(\varphi) \\ &= \mathsf{Substitute}_{z=0} \circ \sigma(\varphi)\end{aligned}$$

where the last equality comes from Claim 27. Thus, using Claim 28, we know that there exists a subset S of the leaf nodes of $\overline{\varphi'}$ labelled by constants such that

$$\mathsf{Unsubstitute}_{z=0}(\overline{\varphi'}, S) = \sigma(\varphi).$$

Moreover, the set S can be viewed as an element of $[2^{k+1}]$ because $\overline{\varphi'}$ has at most $k+1$ leaf nodes labelled by constants. In particular, by construction, we have that

$$|\overline{\varphi'}| = |\varphi|' = |\varphi| = \mathsf{L}(f) + k + 1,$$

and that $\overline{\varphi'}$ computes f, so Claim 26 ensures that φ' has at most $k+1$ many leaf nodes labelled by constants.

Hence, we define the output of our injection from $TestPassers$ to $P \times [2^{k+1}]$ on input $g \in TestPassers$ to be $(\overline{\varphi'}, S)$.

We must prove that this is indeed an injection. Towards this end, we claim that $\mathsf{Unsubstitute}_{z=0}(\overline{\varphi'}, S)$ is a formula computing $\mathsf{Select}[f,g]$. From this claim it is easy to see that this must be an injection.

▷ Claim 29. $\mathsf{Unsubstitute}_{z=0}(\overline{\varphi'}, S)$ is a formula computing $\mathsf{Select}[f,g]$.

Proof. S was chosen so that

$$\mathsf{Unsubstitute}_{z=0}(\overline{\varphi'}, S) = \sigma(\varphi).$$

Thus, $\mathsf{Unsubstitute}_{z=0}(\overline{\varphi'}, S)$ computes the same function as $\sigma(\varphi)$ which in turn computes the same function as φ, which computes $\mathsf{Select}[f,g]$ as desired. ◁

From this injection, we get that

$$|TestPassers| \leq 2^{k+1} \cdot |P|.$$

Next, we give an injection from P to the set

$$\mathsf{CanonOpt}_k\mathsf{Formulas}(f) \times [2\mathsf{L}(f)] \times \{\wedge, \vee\} \times \{0, 1, x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}.$$

To do this we will define an operation $\mathsf{DropLeaf}(\varphi, i)$ that takes as input a formula φ of size $s \geq 2$ and a leaf node $i \in [2s-1]$ from φ and outputs the formula φ' given as follows. We will first describe φ' informally and then give the formal description. φ' is obtained by deleting the leaf node i and making the output of the node i_p that i fed into simply the other node that was being fed into i_p.

Now, we formally describe φ'. Let $i_p \in [2s-1]$ be the internal node that i has an edge to in φ (we know this exists because $|\varphi| \geq 2$). If needed, apply a permutation to φ so $i = 2s-1$ and $i_p = 2s-2$. Let $u \in [2s-3]$ be the other node in φ that feeds into v_p. Let φ' be the formula given by the edge set

$$E' = (E \cap ([2s-3] \times [2s-3])) \cup \{\ (u, v_{pp}) : v_p \text{ feeds into } v_{pp} \text{ in } \varphi\}$$

and the labelling function

$$\tau' = \tau|_{[2s-3]}.$$

For example if $\varphi = (x_1 \vee x_2) \wedge x_3$ and 1 was then index of the x_1 leaf, then $\mathsf{DropLeaf}(\varphi, 1) = x_2 \wedge x_3$.

We show this operation in some sense commutes with formula isomorphisms.

▷ Claim 30. Let φ be formula of size s. Let i be the index a leaf node in φ, and let $\sigma : [2s-3] \to [2s-3]$ be a permutation. Then there exists an integer i' and a permutation $\sigma' : [2s-1] \to [2s-1]$ such that

$$\mathsf{DropLeaf}(\sigma'(\varphi), i') = \sigma \circ \mathsf{DropLeaf}(\varphi, i)$$

Proof. From our definition of $\mathsf{DropLeaf}(\cdot,\cdot)$, we can assume without loss of generality that $i = 2s-1$ and that the internal node that i feeds into in φ is $i_p = 2s-2$.

Then the claim follows from letting σ be equal to σ' on $[2s-3]$ and letting σ be the identity on $\{2s-2, 2s-1\}$ and applying the various definitions. ◁

We can also define a kind of inverse operation AddLeaf function that takes the following four inputs

- a formula φ' on n-inputs of size s,
- a node i in the tree given by φ',
- a gate $\triangledown \in \{\wedge, \vee\}$, and
- a leaf label $\ell \in \{0, 1, x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}$,

and outputs the formula φ of size $s+1$ given as follows. First, we give an informal description and then given a formal definition.

Intuitively, AddLeaf is adding a new $\triangledown$-gate into φ' between the i-node and wherever i was being output to (if i has an output), whose other input is a new ℓ-labeled leaf.

We define φ formally as follows. We will use $2s+1$ to add in our new leaf and $2s$ to add in our new gate. The edge set E_φ of φ is given by taking $E_{\varphi'}$ and adding in the edges $(2s+1, 2s)$ and $(i, 2s)$ and then, if there is a node i_p that i feeds into in φ', adding in an edge $(2s, i_p)$ and removing the (i, i_p) edge. The node labelling τ_φ of φ is given by

$$\tau_\varphi(i) = \begin{cases} \tau_{\varphi'} & \text{, if } i \in [2s-1] \\ \triangledown & \text{, if } i = 2s \\ \ell & \text{, if } i = 2s+1 \end{cases}$$

It is easy to see that AddLeaf can reverse a $\mathsf{DropLeaf}(\cdot,\cdot)$ operation.

▷ Claim 31. Let φ be a formula of size at least two. Let i be the index of a leaf node in φ. Then there exists an integer j, a gate $\triangledown \in \{\wedge, \vee\}$, and a leaf label $\ell \in \{0, 1, x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}$ such that

$$\mathsf{AddLeaf}(\,\mathsf{DropLeaf}(\varphi, i), j, \triangledown, \ell) = \varphi$$

One of the main steps in our injection will be provided by the following claim.

▷ Claim 32. Let $\varphi \in P$. Then there is an i such that $\varphi' = \mathsf{DropLeaf}(\varphi, i)$ is a size $(\mathsf{L}(f) + k)$ formula for computing f.

Proof. Let $\varphi \in P$. Then there is some internal node j in φ that takes as input a leaf node indexed by i satisfying $\tau_\varphi(i) = b \in \{0,1\}$. We will assume $b = 0$ (the proof in the $b = 1$ case is similar).

Let φ_j be the the subformula computed at node j, and let $\triangledown_j = \tau_\varphi(j) \in \{\wedge, \vee\}$ be the gate label of j. We already know that the ith leaf node feeds into φ_j. Let k be the other node feeding into j, and let φ_k be the subformula computed at node k. We split into cases depending on whether $\triangledown_j$ is an $\wedge$-gate or a $\vee$-gate.

First, let us suppose $\triangledown_j$ is a $\vee$-gate. Then the formula φ_j as a function is equivalent to the formula $0 \vee \varphi_k$ which is equivalent as a function to φ_k. Hence it follows that $\varphi' = \mathsf{DropLeaf}(\varphi, i)$ is an $(\mathsf{L}(f) + k)$-size formula (since we removed the ith leaf) computing f (since φ_j and φ_k compute the same function).

Now, suppose that it is a $\wedge$-gate. Then the output of φ_j is always zero. Since $|\varphi_j| \geq 2$, there exists some subformula φ_2 of φ_j of size 2 (i.e. there is some node in φ_v that has two leaves as children and φ_2 is the subformula computed at that node). Since φ_2 has two leaves, there exists one leaf index i' such that $i' \neq i$ (i.e. this leaf node is not the ith leaf node we were considering before).

Then we claim that $\varphi' = \mathsf{DropLeaf}(\varphi, i')$ is an $(\mathsf{L}(f)+k)$-size formula computing f. It is easy to see that $|\varphi'|$ is an $(\mathsf{L}(f)+k)$-size formula since we removed the i'th leaf node. To see that φ' still computes f, note that the 0-labeled ith leaf node in φ still exists in φ'. If the gate node j was removed by the $\mathsf{DropLeaf}(\cdot,\cdot)$ operation, then the output wire of φ_j that computed the 0 function in φ has been replaced by the 0-leaf i in φ' which still computes the 0 function, so φ' must still compute f. If the gate node v was not removed by the $\mathsf{DropLeaf}(\cdot,\cdot)$ operation, then the output corresponding gate to v in φ' is still computing 0 (since it is an $\wedge$-gate with a 0 input), so φ' still computes f. ◁

Now we can finally describe the injection from P to the set

$$\mathsf{CanonOpt}_k\mathsf{Formulas}(f) \times [2\mathsf{L}(f)+2k-1] \times \{\wedge, \vee\} \times \{0,1,x_1,\dots,x_n,\neg x_1,\dots,\neg x_n\}.$$

Given an input φ in P, we have by Claim 32 that there exists a least i-value such that $\varphi' = \mathsf{DropLeaf}(\varphi, i)$ is a formula computing f of size $\mathsf{L}(f)+k$. Let $\overline{\varphi'}$ be the canonical formula isomorphic to φ'. Then we have that $\overline{\varphi'} \in \mathsf{CanonOpt}_k\mathsf{Formulas}(f)$ and we have that there are permutations σ and σ' and an integer i' such that

$$\begin{aligned}\overline{\varphi'} &= \sigma(\varphi')\\ &= \sigma \circ \mathsf{DropLeaf}(\varphi\ , i)\\ &= \mathsf{DropLeaf}(\sigma'(\varphi), i')\end{aligned}$$

where the last equality comes from Claim 30.

Hence, by Claim 31, we have that there exists a gate index $j \in [2\mathsf{L}(f)+2k-1]$, a gate $\triangledown \in \{\wedge,\vee\}$, and a leaf label $\ell \in \{0,1,x_1,\dots,x_n,\neg x_1,\dots,\neg x_n\}$ such that

$$\mathsf{AddLeaf}(\overline{\varphi'}, j, \triangledown, \ell, D) = \sigma'(\varphi).$$

In other words, $\mathsf{AddLeaf}(\overline{\varphi'}, j, \triangledown, \ell, D)$ outputs a formula isomorphic to φ.

Thus, we set the output of the injection on input φ to be $(\overline{\varphi'}, j, \triangledown, \ell, D)$. The fact that this is an injection from P is ensured by the fact that $\mathsf{AddLeaf}(\overline{\varphi'}, j, \triangledown, \ell, D)$ is isomorphic to φ and the fact that P contains only canonical formulas.

Hence, we get that

$$\begin{aligned}|P| &\le |\mathsf{CanonOpt}_k\mathsf{Formulas}(f)| \cdot 2(\mathsf{L}(f)+k)\cdot 2\cdot(2n+1)\\ &\le O(|\mathsf{CanonOpt}_k\mathsf{Formulas}(f)|(N+k)\log N).\end{aligned}$$

Combining this with upper bound on $TestPassers$ in terms of P, we get that

$$|TestPassers| \le O(|\mathsf{CanonOpt}_k\mathsf{Formulas}(f)| \cdot 2^k N \log N) \qquad ◀$$

5 A deterministic reduction that works on average

We will now use the tools developed in Section 3 and Section 4 to give a search to decision reduction that is efficient on functions with few near-optimal formulas.

▶ **Theorem 33.** *There is a deterministic algorithm solving* Search-MFSP *on inputs of length* N *given access to an oracle that solves* MFSP *on instances of length* $2N$ *that runs in time* $O(|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)|^2 \cdot N^6 \log^2 N)$ *where* $n = \log N$*.*

Before we prove Theorem 33, we state a corollary that follows from the bound on the size of $\mathsf{CanonOpt}_k\mathsf{Formulas}(f)$ for a random function given in Proposition 14.

▶ **Corollary 34.** *There is a deterministic algorithm solving* Search-MFSP *on inputs of length N given access to an oracle that solves* MFSP *on instances of length $2N$ that runs in time $2^{O(\frac{N}{\log\log N})}$ on all but a $o(1)$-fraction of instances.*

Proof of Corollary 34. This corollary follows by combining the algorithm in Theorem 33 with the recursive algorithm for Search-MFSP given in Theorem 20 and using Proposition 14 to bound $|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)|$ by $2^{O(\frac{N}{\log\log N})}$ for a random function.

There is actually a subtlety in appealing to Theorem 20 in that the running time of the algorithm in Theorem 33 has a dependence on the number near-optimal formulas of its input. Hence, we need to argue that when the algorithm in Theorem 20 makes recursive calls to the algorithm in Theorem 20 on functions g other than the original input f that g also has few near-optimal formulas. However, this is not a problem since it is easy to see that if a function g is computed by some gate in an optimal formula for f (as it must be if a recursive call is made to g), then the number of near-optimal formulas for g is at most the number of near-optimal formulas for f (since one can create a near optimal formula for f by taking the optimal formula for f that computes g at some gate and replacing the subformula at that gate with a near-optimal formula for g). ◀

Proof of Theorem 33. We provide the pseudocode of our DecompProblem algorithm in Algorithm 2, which we recommend the reader look at before proceeding.

5.1 Correctness of Algorithm 2

In this subsection we show that Algorithm 2 has the desired input/output behavior.

Fix some function f with n-inputs satisfying $\mathsf{L}(f) \geq 2$. Let $N = 2^n$.

Part 1: building *Candidates*. First, we will prove some loop invariants that will help us show that *Candidates* and $PartialCandidates_{(i)}$ contain those functions we are interested in and do not contain many more things.

The following claim shows that the $x^\star$ described on Line 10 always exists and that the ?-values of partial functions in $PartialCandidates_{(i)}$ always have an easily computable structure.

▷ Claim 35. Before and after each iteration of the while loop, it is true that if $\gamma \in PartialCandidates_{(i)}$, then

- $\gamma(x) = ? \iff x \geq i$ (interpreting i as a binary string in $\{0,1\}^n$ in the natural way),
- and consequently $|\gamma^{-1}(\{0,1\})| = i$.

Proof. Clearly the claim is satisfied before the first iteration of the while loop when $i = 0$ and $PartialCandidates_{(i)} = \{AllUnknown\}$.

Now, we must argue inductively. Suppose $1 \leq i \leq N$ and $\gamma' \in PartialCandidates_{(i)}$. Then, it follows that there is some $\gamma \in PartialCandidates_{(i-1)}$ and some $b \in \{0,1\}$ such that $\gamma' = \gamma_b$ where γ_b is as defined in the pseudocode. That is, γ_b is equal to γ except that the first ?-value (which occurs at $x^\star_{old} = i - 1$ by the inductive hypothesis) is replaced by a b. Thus, we have

$$\gamma'(x) = ? \iff \gamma(x) = ? \wedge (x \neq x^\star_{old}) \iff x > x^\star_{old} \iff x \geq i$$

where the first equivalence comes from the definition of $\gamma_b = \gamma'$ and the second equivalence comes from the fact that $x^\star_{old} = i - 1$. ◁

Algorithm 2 A deterministic search to decision reduction for MFSP whose run time depends on the number of "near-optimal formulas".

```
procedure OptimalSubcomputation($f$)
    ▷ Given the length-$N$ truth table of a function $f$ that takes $n$-inputs with $\mathsf{L}(f) \geq 2$, this procedure returns an element $\{g, \triangledown, h\}$ of $\mathsf{OptSubcomps}(f)$.

Part 1: Building a $Candidates$ list
    Let $allUnknown : \{0,1\}^n \to \{0,1,?\}$ be given by $allUnknown(x) = ?$ for all $x$.
    Set $PartialCandidates_{(0)} = \{allUnknown\}$.
    Set $i = 0$.
    while $i < N$ do
        Set $PartialCandidates_{(i+1)} = \emptyset$.
        for all $\gamma \in PartialCandidates_{(i)}$ and for all $b \in \{0,1\}$ do
            Let $x^\star$ be the lexicographically first input satisfying $\gamma(x^\star) = ?$.
            Let $\gamma_b : \{0,1\}^n \to \{0,1,?\}$ be given by $\gamma_b(x) = \begin{cases} b & \text{, if } x = x^\star \\ \gamma(x) & \text{, otherwise.} \end{cases}$
            Let $g_{\gamma_b}$ be the (total) function given by $g_{\gamma_b}(x) = \begin{cases} 1 & \text{, if } \gamma_b(x) = ? \\ \gamma_b(x) & \text{, otherwise.} \end{cases}$
            if $\mathsf{L}(\mathsf{Select}[f, g_{\gamma_b}]) \leq \mathsf{L}(f) + n + 2$ then
                Add $\gamma_b$ to $PartialCandidates_{(i+1)}$.
            end if
        end for
        Set $i = i + 1$.
    end while
    Set $Candidates = PartialCandidates_{(N)}$.

Part 2: Finding an optimal pair within $Candidates$
    for all pairs $g, h \in Candidates$ and for all gates $\triangledown \in \{\wedge, \vee\}$ do
        if $\mathsf{L}(g) + \mathsf{L}(h) = \mathsf{L}(f)$ and $f = g \triangledown h$ then
            return $\{g, \triangledown, h\}$ .
        end if
    end for
end procedure
```

Next, we show that the $PartialCandidates_{(i)}$ never contains "redundant" partial functions.

▷ Claim 36. Before and after each iteration of the while loop, it is true that if γ' and γ'' are distinct elements of $PartialCandidates_{(i)}$, then no total function agrees with both γ' and γ''.

Proof. Before the first iteration of the while loop runs, $i = 0$ and $PartialCandidates_{(0)}$ only contains the single partial function $AllUnknown$, so the claim clearly holds.

Now we must show that the claim holds inductively. Assume $1 \leq i \leq N$. For contradiction, suppose there was some total function q that agrees with distinct elements μ and μ' from $PartialCandidates_{(i)}$. It follows that there exists some $b, b' \in \{0,1\}$ and some (possibly not distinct) $\gamma, \gamma' \in PartialCandidates_{(i-1)}$ such that $\mu = \gamma_b$ and $\mu' = \gamma'_{b'}$ (using the notation from the pseudocode where these functions γ_b and $\gamma'_{b'}$ are given by replacing the output of the first ?-valued input in γ or γ' respectively with a b-value or b'-value respectively). It follows that q must also agree with γ and γ'.

Either $\gamma \neq \gamma'$ or not. If $\gamma \neq \gamma'$, then q agrees with two distinct elements from $PartialCandidates_{(i-1)}$ which contradicts the inductive hypothesis.

Now suppose that $\gamma = \gamma'$. Then it must be that $b \neq b'$ (otherwise, $\mu = \mu'$ and we assumed they are distinct). But then, we have then γ and γ' have the same first ?-valued input $x^\star$, so

$$b = \mu(x^\star) = q(x^\star) = \mu'(x^\star) = b'$$

which contradicts that $b \neq b'$. ◁

Moreover, $PartialCandidates_{(i)}$ only contains partial functions that can be completed to total functions that pass a certain test.

▷ Claim 37. Before and after each iteration of the while loop, it is true that if $\gamma \in PartialCandidates_{(i)}$ then there exists a function g on n-inputs that agrees with γ such that

$$\mathsf{L}(\mathsf{Select}[f, g]) \leq \mathsf{L}(f) + n + 2.$$

Proof. Before the first iteration of the while loop runs, $i = 0$ and $PartialCandidates_{(0)}$ only contains one partial function ($AllUnknown$). The function f clearly agrees with $AllUnknown$, and it is easy to see that $\mathsf{L}(\mathsf{Select}[f, f]) = \mathsf{L}(f) \leq \mathsf{L}(f) + n + 1$, as desired. Thus, the claim holds before the first iteration of the while loop.

Moreover, the claim clearly continues holding inductively because before any γ_b is added to $PartialCandidates_{(i)}$, we check to see if the function g_{γ_b} satisfies

$$\mathsf{L}(\mathsf{Select}[f, g_{\gamma_b}]) \leq \mathsf{L}(f) + n + 2$$

and g_{γ_b} agrees with γ_b by construction. ◁

Finally, we show that $PartialCandidates_{(i)}$ always contains the partial functions we want.

▷ Claim 38. Suppose some function q is in an optimal subcomputation for f. Then before and after each iteration of the while loop there is a $\gamma \in PartialCandidates_{(i)}$ such that q agrees with γ. Moreover, once part 1 is finished, $q \in Candidates$

Proof. Fix some q as in the statement of the claim.

Before the first iteration of the while loop runs, $i = 0$ and $PartialCandidates_{(0)}$ contains the all-? partial function $AllUnknown$, so q agrees with $AllUnknown$ and the claim holds.

Now, we must show the claim holds inductively. Assume $1 \leq i \leq N$. Then by induction there exists a $\gamma \in PartialCandidates_{(i-1)}$ such that q agrees with γ. Let $b = q(i-1)$. Then q agrees with γ_b as defined in the pseudocode (replacing the first ?-value in γ with a b-value) since Claim 35 implies that

$$\gamma_b(x) = \begin{cases} b & \text{, if } x = i - 1 \\ \gamma(x) & \text{, otherwise.} \end{cases}.$$

Thus, if we could show $\gamma_b \in PartialCandidates_{(i)}$, we would be done with showing the first part of the claim. From the pseudocode, it is clear $\gamma_b \in PartialCandidates_{(i)}$ if

$$\mathsf{L}(\mathsf{Select}[f, g_{\gamma_b}]) \leq \mathsf{L}(f) + n + 2,$$

where g_{γ_b} is as defined in the code (the function given by replacing the ?-values in γ_b with ones) which we now prove.

We already noted that

$$\gamma_b(x) = \begin{cases} b & \text{, if } x = i-1 \\ \gamma(x) & \text{, otherwise.} \end{cases}.$$

Thus, appealing to Claim 35, we know that $\gamma_b(x) = ? \iff x > x^\star$ where $x^\star \in \{0,1\}^n$ is the binary string equivalent to $i-1$ (note that $0 \leq i-1 \leq N-1$ so this makes sense). Hence, since q agrees with γ_b, we have that $g_{\gamma_b}(x) = q(x) \vee \mathit{GrtrThan}_{x^\star}(x)$ where $\mathit{GrtrThan}_{x^\star}(x) = 1$ if and only if $x > x^\star$.

Thus, we have that

$$\begin{aligned} \mathsf{Select}[f, g_{\gamma_b}](x,z) &= \begin{cases} f(x) & \text{, if } z = 0 \\ g_{\gamma_b}(x) & \text{, if } z = 1 \end{cases} \\ &= \mathsf{Select}[f,q](x,z) \vee (z \wedge \mathit{GrtrThan}_{x^\star}(x)) \end{aligned}$$

Since $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$, we know that $\mathsf{L}(\mathsf{Select}[f,q]) = \mathsf{L}(f) + 1$ by Lemma 23, and Proposition 15 implies that $\mathsf{L}(\mathit{GrtrThan}_{x^\star}) \leq n$. Hence, we have that

$$\mathsf{L}(\mathsf{Select}[f, g_{\gamma_b}]) \leq \mathsf{L}(f) + n + 2.$$

Finally, we show that $q \in \mathit{Candidates}$ after part 1 finishes. Clearly, it suffices to show that $q \in \mathit{PartialCandidates}_{(N)}$ after part 1 finishes. We have already shown that there is a $\gamma \in \mathit{PartialCandidates}_{(N)}$ such that γ agrees with q. However, Claim 35 implies that γ is a total function and hence it equals q, so $q \in \mathit{PartialCandidates}_{(N)}$. ◁

Part 2: Finding a g, h pair within $\mathit{Candidates}$. First, we note that any output by Algorithm 2 must be correct.

▷ Claim 39. Any value Algorithm 2 outputs must be an element of $\mathsf{OptSubcomps}(f)$.

Proof. Any output $\{g, \triangledown, h\}$ of Algorithm 2 must satisfy $f = g \triangledown h$ and $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ which implies $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$ by Lemma 21. ◁

Finally, we show that Algorithm 2 must output a value.

▷ Claim 40. Algorithm 2 must output a value (on input f).

Proof. Since $\mathsf{L}(f) \geq 2$, we have that $\mathsf{OptSubcomps}(f)$ is non-empty. Let $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$.

Claim 38 implies that $\{g, h\} \subseteq \mathit{Candidates}$. On the other hand, Lemma 21 implies that $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ and $f = g \triangledown h$. Thus, it is clear that part 2 will either output $\{g, \triangledown, h\}$ or output a value before that. ◁

5.2 Running Time of Algorithm 2

Fix some function f with n-inputs satisfying $\mathsf{L}(f) \geq 2$. Let $N = 2^n$. We break the running time analysis into the two pieces of the algorithm.

Part 1. It is easy to see that the run time of part 1 can be bounded by

$$O(N + \sum_{i \in [N]} N \cdot |PartialCandidates_{(i)}|)$$

where $|PartialCandidates_{(i)}|$ indicates the size of $PartialCandidates_{(i)}$ after Algorithm 2 is finished adding elements to it.

Moreover, we can bound the quantity $|PartialCandidates_{(i)}|$ as follows. Claim 37 implies that every partial function in $PartialCandidates_{(i)}$ must be consistent with some total function g on n-inputs satisfying

$$\mathsf{L}(\mathsf{Select}[f,g]) \leq \mathsf{L}(f) + n + 2.$$

On the other hand, Claim 36 implies that any single (total) function can agree with at most partial function in $PartialCandidates_{(i)}$. Hence, we have that

$$|PartialCandidates_{(i)}| \leq |\{g : \mathsf{L}(\mathsf{Select}[f,g]) \leq \mathsf{L}(f) + n + 2\}|$$

and Lemma 24 implies that

$$|\{g : \mathsf{L}(\mathsf{Select}[f,g]) \leq \mathsf{L}(f) + n + 2\}| \leq O(|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)| \cdot N^2 \log N).$$

Thus, we have that part 1 runs in time at most $O(|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)| \cdot N^4 \log N)$. Moreover, part 1 only makes oracle calls of length at most $2N$ (to calculate $\mathsf{L}(\mathsf{Select}[f, g_{\gamma_b}])$).

Part 2. It is easy to see that this part runs in time $O(N \cdot |Candidates|^2)$. Hence, since $Candidates = PartialCandidates_{(N)}$, the analysis in part 1 above implies that

$$|Candidates| \leq O(|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)| \cdot N^2 \log N).$$

Thus, part 2 runs in time at most

$$O(|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)|^2 \cdot N^5 \log^2 N).$$

Moreover, part 2 only makes oracle calls of length N.

In total. Putting it all together, we have that Algorithm 2 runs in time at most

$$O(|\mathsf{CanonOpt}_{n+1}\mathsf{Formulas}(f)|^2 \cdot N^5 \log^2 N)$$

and only makes oracle queries of length $2N$. ◀

6 A worst-case randomized reduction

We now present a worst-case search to decision reduction for MFSP.

▶ **Theorem 41.** *There is a randomized algorithm solving* Search-MFSP *on inputs of length N in time $O(2^{.67N})$ given access to an oracle that solves* MFSP *on instances of length $2N$.*

Proof. We prove this theorem by giving an oracle algorithm solving DecompProblem and appealing to Theorem 20. We provide the pseudocode of our algorithm in Algorithm 3, which we recommend the reader look at before proceeding.

Algorithm 3 A randomized worst-case search to decision reduction for MFSP.

```
procedure WorstCaseOptimalSubcomputation($f$)
    ▷ Given the length-$N$ truth table of a function $f$ that takes $n$-inputs with $\mathsf{L}(f) \geq 2$, this procedure returns an element $\{g, \triangledown, h\}$ of $\mathsf{OptSubcomps}(f)$.

    Set $s = \frac{2}{3} \cdot \frac{2^n}{\log n}$
    Set $t = 2^{2N/3}$

Part 1: Try random formulas
    for $i = 1, \ldots, t$ do
        Let $G_i$ be a uniformly random binary tree with $\mathsf{L}(f)$-leaves. (Section 6.2 discusses how to sample $G_i$.)
        Turn $G_i$ into a uniformly random formula $\varphi_i$ by picking uniformly random gates from $\{\wedge, \vee\}$ and uniformly random input leaves from $\{0, 1, x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$.
        if $\varphi_i$ computes $f$ then
            Write $\varphi_i = \varphi_{i,1} \triangledown \varphi_{i,2}$.
            Let $g$ and $h$ be the function computed by $\varphi_{i,1}$ and $\varphi_{i,2}$ respectively.
            if $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ then
                return $\{g, \triangledown, h\}$.
            end if
        end if
    end for

Part 2: Generate a small list of candidates for $g$
    Set $\mathit{SmallFuncs} = \{g : g \text{ is a Boolean function with } n\text{-inputs and } \mathsf{L}(g) \leq s\}$.
    Set $\mathit{Candidates} = \{g \in \mathit{SmallFuncs} : \mathsf{L}(\mathsf{Select}[f, g]) \leq \mathsf{L}(f) + 1\}$.

Part 3: Try to find a $g, h$ pair within Candidates
    for each pair of functions $(g, h) \in \mathit{Candidates}$ and for each gate $\triangledown \in \{\wedge, \vee\}$ do
        if $f = g \triangledown h$ and $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ then
            return $\{g, \triangledown, h\}$.
        end if
    end for

Part 4: Try to find a $g, h$ pair by looking at functions $h$ satisfying $f = g \triangledown h$
    Set $\mathit{SmallCandidates} = \{g \in \mathit{Candidates} : \mathsf{L}(g) \leq \mathsf{L}(f) - s\}$.
    for each function $g \in \mathit{SmallCandidates}$ and for each $\triangledown \in \{\wedge, \vee\}$ do
        if $\forall x \in \{0,1\}^n \; \exists b \in \{0,1\}$ such that $g(x) \triangledown b = f(x)$ then
            Let $h_{?,g} : \{0,1\}^n \to \{0, 1, ?\}$ be the unique partial function on $n$-inputs such that $\forall\, h,\; f = g \triangledown h \iff h$ agrees with $h_{?,g}$.
            for each total function $h$ that agrees with $h_{?,g}$ do
                if $f = g \triangledown h$ and $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ then
                    return $\{g, \triangledown, h\}$.
                end if
            end for
        end if
    end for
end procedure
```

6.1 Correctness of Algorithm 3

In this section, we prove that Algorithm 3 has the desired input/output behavior. In our analysis, we will use s and t as parameters which we will set to the optimal values (which are written in the pseudocode) in Section 6.2 where we do the running time analysis for Algorithm 3.

Fix some function f on n-inputs with $\mathsf{L}(f) \geq 2$. We analyze the algorithm in parts.

Part 1. Since φ_i is chosen to have $\mathsf{L}(f)$ leaves and the algorithm in part 1 checks if φ_i computes f before returning any value, the following claim is clear.

▷ Claim 42. Any output by Algorithm 3 returned in part 1 must be an element of $\mathsf{OptSubcomps}(f)$.

Moreover, we can lower bound the probability that Algorithm 3 returns a value in part 1 as follows. Recall that $\mathsf{CanonOpt_0Formulas}(f)$ is the set of optimal canonical formulas for f. We will show that part 1 succeeds if this set is large.

▷ Claim 43. If $t \geq 5 \cdot \frac{2^{N(1+o(1))}}{|\mathsf{CanonOpt_0Formulas}(f)|}$, then part 1 of Algorithm 3 will return a value at least 99% of time.

Proof. Since we are picking each s-leaf formula φ_i uniformly at random, the probability that any fixed formula computes f is at least

$$\frac{|\mathsf{CanonOpt_0Formulas}(f)|}{\text{the total number of formulas with } \mathsf{L}(f) \text{ leaves}}$$

Combining Theorem 13 with Proposition 12 upper bounds the denominator by $2^{N(1+o(1))}$, so

$$\Pr[\varphi_i \text{ computes } f] \geq \frac{|\mathsf{CanonOpt_0Formulas}(f)|}{2^{N(1+o(1))}}.$$

Since each of these φ_i are chosen independently, we have that

$$\begin{aligned}\Pr[\exists i \in [t] \text{ such that } \varphi_i \text{ computes } f] &\geq 1-(1-\frac{|\mathsf{CanonOpt_0Formulas}(f)|}{2^{N(1+o(1))}})^t\\ &\geq 1-e^{-t\cdot\frac{|\mathsf{CanonOpt_0Formulas}(f)|}{2^{N(1+o(1))}}}\\ &\geq 1-e^{-5}\\ &\geq .99\end{aligned}$$

Hence, with probability at least 99%, part 1 will find a φ_i computing f at which point it will clearly return a value. ◁

Part 2. In part 2, Algorithm 3 constructs the *Candidates* set. We prove two claims about this set. First, that it contains the functions we care about, and second that its size can be bounded using the size of the $\mathsf{CanonOpt_0Formulas}(f)$ set.

▷ Claim 44. Suppose g is in an optimal subcomputation for f. Then $\mathsf{L}(g) \leq s \implies g \in$ *Candidates*.

Proof. Since $\mathsf{L}(g) \leq s$, we know that $g \in$ *SmallFuncs*. Next, since g is in an optimal subcomputation for f, we have by Lemma 23 that

$$\mathsf{L}(\mathsf{Select}[f,g]) \leq \mathsf{L}(f)+1,$$

so g is an element of *Candidates*. ◁

▷ Claim 45.

$$|Candidates| = O(|\mathsf{CanonOpt}_0\mathsf{Formulas}(f)| \cdot N \log N)$$

Proof. By construction, we have that

$$Candidates \subseteq \{g : \mathsf{L}(\mathsf{Select}[f, g]) = \mathsf{L}(f) + 1\}.$$

On the other hand, Lemma 24, we have that

$$|\{g : \mathsf{L}(\mathsf{Select}[f, g]) = \mathsf{L}(f) + 1\}| \leq O(|\mathsf{CanonOpt}_0\mathsf{Formulas}(f)| \cdot N \log N)$$ ◁

Part 3. In Part 3, Algorithm 3 tries to find a g, h pair by looking within the $Candidates$ set. We show this works as long as there is a $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$ where $\mathsf{L}(g)$ and $\mathsf{L}(h)$ are small.

First, we note that part 3 can only return correct answers.

▷ Claim 46. Any output returned by Algorithm 3 in part 3 will be an element of $\mathsf{OptSubcomps}(f)$.

Proof. In order for a $\{g, \triangledown, h\}$ value to be returned in part 3 it must satisfy $f = g \triangledown h$ and $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$. Thus, by Lemma 21, we know $\{g, \triangledown, h\} \in \mathsf{OptSubcomps}(f)$. ◁

Next, we give sufficient conditions on which part 3 will return an answer.

▷ Claim 47. If there exists an element $\{g', \triangledown, h'\}$ of $\mathsf{OptSubcomps}(f)$ such that

$$\max\{\mathsf{L}(g'), \mathsf{L}(h')\} \leq s,$$

then Algorithm 3 will return a value in part 3 or before.

Proof. Suppose there exists an element $\{g', \triangledown, h'\}$ of $\mathsf{OptSubcomps}(f)$ such that

$$\max\{\mathsf{L}(g'), \mathsf{L}(h')\} \leq s,$$

and assume that this procedure has not returned a value before it reaches part 3. Then by Claim 44, we have that both g' and h' are in $Candidates$. Moreover, since $\{g', \triangledown, h'\} \in \mathsf{OptSubcomps}(f)$, we know that $f = g' \triangledown' h'$ and $\mathsf{L}(f) = \mathsf{L}(g) + \mathsf{L}(h)$ by Lemma 21. Hence it is clear there are value the for loop will return an output if it reaches $g = g', h = h'$, and $\triangledown = \triangledown'$ (although it could return a value before that). ◁

Part 4. In the final part of Algorithm 3, we look for matching h functions for g candidates with small complexity.

First, we note that any output returned by part 4 must be correct by essentially the same proof as Claim 46 in part 3.

▷ Claim 48. Any output returned by Algorithm 3 in part 4 will be an element of $\mathsf{OptSubcomps}(f)$.

Next, we show sufficient conditions for part 4 returning an answer.

▷ Claim 49. If $s \geq \mathsf{L}(f)/2$ and there exists a $\{g', \triangledown, h'\} \in \mathsf{OptSubcomps}(f)$ such that $\mathsf{L}(h') \geq s$, then Algorithm 3 will return a value in part 4 or earlier.

Proof. Using Lemma 21, we have $\mathsf{L}(f) = \mathsf{L}(g') + \mathsf{L}(h')$. Thus, since $s \geq \mathsf{L}(f)/2$ and $\mathsf{L}(h') \geq s$, we have that

$$\mathsf{L}(g') = \mathsf{L}(f) - \mathsf{L}(h') \leq \mathsf{L}(f) - s \leq 2s - s = s.$$

Hence, by Claim 44, we have that $g' \in \mathit{Candidates}$. Moreover, since $\mathsf{L}(g') \leq \mathsf{L}(f) - s$, it follows that $g' \in \mathit{SmallCandidates}$. Thus, it is clear that part 4 will return a value if its for loop ever reaches $g = g', \triangledown = \triangledown'$, and $h = h'$ (though it could return a value before that). ◁

In total. Finally, we can prove the correctness of the input/output behavior of Algorithm 3.

▷ Claim 50. If $s \geq \mathsf{L}(f)/2$, then Algorithm 3 (run on input f) returns an element of $\mathsf{OptSubcomps}(f)$.

Proof. Put together, Claim 42, Claim 46, and Claim 48 ensures that any output returned by Algorithm 3 must be an element of $\mathsf{OptSubcomps}(f)$.

Hence, it suffices to show that Algorithm 3 will always output a value. We divide into two cases. Either there exists an element $\{g', \triangledown, h'\} \in \mathsf{OptSubcomps}(f)$ such that $\max\{\mathsf{L}(g'), \mathsf{L}(h')\} \leq s$ or not.

If there exists such an element, then Claim 47 ensures that Algorithm 3 will output a value.

Now suppose that for all $\{g', \triangledown, h'\} \in \mathsf{OptSubcomps}(f)$ we have $\max\{\mathsf{L}(g'), \mathsf{L}(h')\} \geq s$. Since $\mathsf{L}(f) \geq 2$, we know $\mathsf{OptSubcomps}(f)$ is non-empty by Proposition 18. Hence we can fix some $\{g', \triangledown, h'\} \in \mathsf{OptSubcomps}(f)$ satisfying $\max\{\mathsf{L}(g'), \mathsf{L}(h')\} \geq s$. Without loss of generality we can assume that $\mathsf{L}(g') \leq \mathsf{L}(h')$. Thus, we have that $\mathsf{L}(h') \geq s$, and by hypothesis $s \geq \mathsf{L}(f)/2$, so Claim 49 ensures Algorithm 3 outputs a value. ◁

Thus Algorithm 3 is correct as long as $s \geq \mathsf{L}(f)/2$. Indeed, in the next section we will set s so that $s \geq \max\{\mathsf{L}(f)/2 : f \text{ takes } n\text{-inputs}\}$.

6.2 Runtime of Algorithm 3

In this subsection, we bound the runtime of Algorithm 3 and set s and t to the optimal values. We analyze Algorithm 3 in its parts.

Part 1. The for loop in part 1 clearly runs t times, so we just need to bound the running time of each iteration. Generating a uniformly random binary with $\mathsf{L}(f)$-leaves can be done in linear time (see [15] for a survey of various approaches). The other operations in the for loop can clearly be done in time $O(N + \mathsf{L}(f))$. Hence, all of part 1 runs in time $O(t \cdot (N + \mathsf{L}(f))$ which is $O(t \cdot N)$ using the worst-case formula upper bound from Theorem 13.

Moreover, part 1 only makes oracle calls of length N (to calculate $\mathsf{L}(f)$).

Part 2. Building the $\mathit{SmallFuncs}$ set requires iterating through all formulas of size s (which is bounded by $2^{(1+o(1))\cdot s \log n}$ using Proposition 12) and then computing the truth table of each of these size s formulas (which can be done in time $O(Ns)$). Hence, computing $\mathit{SmallFuncs}$ can be done in $O(N \cdot 2^{(1+o(1))\cdot s \log n})$ time. Moreover, $|\mathit{SmallFuncs}|$ is clearly upper bounded by the upper bound on the number of formulas of size s: $2^{(1+o(1))\cdot s \log n}$.

Next, building the $\mathit{Candidates}$ set can be done in time $O(|\mathit{SmallFuncs}| + N) = O(N \cdot 2^{(1+o(1))\cdot s \log n})$, and we use oracle calls of length $2N$ in this step (for $\mathsf{Select}[f, g]$).

Hence, part 2 runs in time $O(N \cdot 2^{(1+o(1))\cdot s\log n})$.

We will make use of the following claim later, which bounds the size of the *Candidates* set if part 1 did not return a value.

▷ Claim 51. Fix some function f. Then with 99% probability (over the algorithm's choice of random formulas) either Algorithm 3 on input f returns before reaching part two or

$$|Candidates| \le \frac{2^{N(1+o(1))}}{t}.$$

Proof. Suppose that *Algorithm* 3 reaches part two on input f and

$$|Candidates| > \frac{2^{N(1+o(1))}}{t}.$$

Then Claim 45 implies that

$$\frac{2^{N(1+o(1))}}{t} = O(|\mathsf{CanonOpt}_0\mathsf{Formulas}(f)|N\log n)$$

which implies that

$$|\mathsf{CanonOpt}_0\mathsf{Formulas}(f)| \ge \frac{2^{N(1+o(1))}}{t}.$$

Hence, Claim 43 implies that Algorithm 3 will return in part 1 with 99% probability. ◁

Part 3. It is easy to see that part 3 runs in time $O(|Candidates|^2 + N)$ and makes oracle calls of length N.

Thus, using Claim 51, we have that with 99% probability part 3 runs in time $\frac{2^{2N(1+o(1))}}{t^2}$.

Part 4. Computing *SmallCandidates* can be done in $O(|Candidates| + N)$ time, and the outer for loop runs at most $O(|Candidates|)$ many times.

It remains to bound the running time of each iteration of the outer for loop. The if condition can be checked in $O(N)$ time. Constructing $h_{?,g}$ also takes $O(N)$ time (similar to the if condition, just iterate through each input $x \in \{0,1\}^n$ and see which values of $b \in \{0,1\}$ satisfy $f(x) = g(x)\nabla b$). Each iteration of the inner for loop takes $O(N)$ time. Finally, the inner for loop runs $2^{|h_{?,g}^{-1}(?)|}$ many times.

Thus, the total running time for part 4 is

$$O(|Candidates| \cdot (N + N \cdot 2^{\max_g\{|h_{?,g}^{-1}(?)|\}}))$$

time.

Moreover, we can bound the quantity $\max_g\{|h_{?,g}^{-1}(?)|\}$ as follows.

▷ Claim 52. $\max_g\{|h_{?,g}^{-1}(?)|\} \le (1+o(1))(N - s\log n)$.

Proof. Since any function h that agrees with $h_{?,g}$ satisfies $g\nabla h = f$ and, we have that

$$\mathsf{L}(f) \le \mathsf{L}(g) + \mathsf{L}(h_{?,g}).$$

Since $\mathsf{L}(g) \le \mathsf{L}(f) - s$ (since $g \in SmallCandidates$), we have that

$$\mathsf{L}(h_{?,g}) \ge s.$$

On the other hand, the upper bound on the formula complexity of partial functions from Theorem 16 implies that

$$\mathsf{L}(h_{?,g}) \leq (1+o(1))(1-\frac{|h_{?,g}^{-1}(?)|}{N})\frac{N}{\log n}.$$

Hence

$$s \leq (1+o(1))(1-\frac{|h_{?,g}^{-1}(?)|}{N})\frac{N}{\log n}$$

so

$$|h_{?,g}^{-1}(?)| \leq N - \frac{s\log n}{(1+o(1))} \leq (1+o(1))(N - s\log n). \qquad \triangleleft$$

Thus, using Claim 51, we have that with 99% probability part 4 runs in $\frac{2^{2N(1+o(1))}}{t^2}$.

$$O(\frac{2^{N(1+o(1))}}{t}\cdot N\cdot 2^{(1+o(1))(N-s\log n)}) = \frac{2^{(1+o(1))(2N-s\log n)}}{t}$$

time.

In total. Thus, we get that with 99% probability Algorithm 3 runs in time

$$O(t\cdot N) + O(N\cdot 2^{(1+o(1))\cdot s\log n}) + \frac{2^{2N(1+o(1))}}{t^2} + \frac{2^{(1+o(1))(2N-s\log n)}}{t}.$$

Letting $s = \frac{2}{3}\frac{2^n}{\log n}$ and $t = 2^{\frac{2}{3}N}$, we get that the running time is bounded by

$$2^{(1+o(1))\frac{2}{3}N}.$$

Moreover, s will satisfy $s \geq \mathsf{L}(f)/2$ (as required for the correctness of the algorithm) for all f with n-inputs when n is sufficiently large by Theorem 13. ◀

7 A “bottom-up” reduction for DeMorgan Formulas

In this section, we provide another algorithm for solving Search-MFSP that is also efficient on average, though with worse guarantees than the one given by Theorem 33. Despite its worse guarantees, we present the algorithm because it uses a different “bottom-up” approach that we think is interesting.

We begin by proving a lemma that bounds the depth of optimal DeMorgan formulas.

▶ **Lemma 53** (Large optimal DeMorgan formulas have not too large depth). *Let $f : \{0,1\}^n \to \{0,1\}$. Let φ be an optimal DeMorgan formula for computing f. Then the depth of f is at most $\frac{10}{n}\cdot 2^n$ for sufficiently large n.*

Proof. Let d be a parameter we set later. For contradiction, suppose that φ is an optimal formula for computing f with depth greater than d. Clearly then $\mathsf{L}(f) > d$ as well. For φ to have depth greater than d, there must be gates $\triangledown_1,\ldots,\triangledown_{d-1} \in \{\wedge,\vee\}$ and subformulas $\varphi_1,\ldots,\varphi_d$ such that

$$\varphi = \varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \ \ldots \ \triangledown_{d-1} \varphi_d$$

where we evaluate gates from left to right, so this formula with parentheses would be

$$(\dots((\varphi_1 \triangledown_1 \varphi_2) \triangledown_2 \varphi_3)\dots) \triangledown_{d-1} \varphi_d,$$

and $|\varphi_i| \geq 1$ for all $i \in [d]$. In other words, consider a d-length path from some subformula φ_1 to the output gate g_{d-1} in the formula and let $\varphi_2, \dots, \varphi_d$ be all the subformulas in φ from bottom to top (viewing the output gate as the top) intersecting this path. Similarly, let $\triangledown_1, \dots, \triangledown_{d-1}$ be the gates in order from bottom to top along this path.

Then, we have that

$$\mathsf{L}(f) = |\varphi| \geq \sum_{i\in[d]} |\varphi_i|.$$

Thus, we have that

$$\mathbb{E}_{i\in[d]\setminus\{1\}}[|\varphi_i|] \leq \frac{\mathsf{L}(f)-1}{d-1}.$$

Hence by Markov's inequality, we have that there exists a subset $S \subseteq [d] \setminus \{1\}$ of size at least $\frac{d-1}{2}$ such that for all $i \in S$ we have $|\varphi_i| \leq 2 \cdot \frac{\mathsf{L}(f)-1}{d-1}$.

On the other hand the number of distinct formulas on n-inputs with size at most $2 \cdot \frac{\mathsf{L}(f)-1}{d-1}$ is bounded by

$$2^{2\cdot\frac{\mathsf{L}(f)-1}{d-1}\log n(1+o(1))}$$

according to Proposition 12. Assume that we have chosen d so that

$$|S| \geq \frac{d-1}{2} > 2^{\frac{\mathsf{L}(f)-1}{d-1}\log n(1+o(1))}.$$

Then, by the pigeonhole principle, there exists $i \leq j \in S$ such that φ_i and φ_j compute the same function. We can use this to get a contradiction to optimality as follows. Assume that $\triangledown_{i-1} = \triangledown_{j-1} = \wedge$ (the other cases are similar). Then, substituting in $\triangledown_{i-1} = \triangledown_{j-1} = \wedge$ we would have that the subformula of φ computed at $\triangledown_{i-1}$, that is

$$\varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \dots \triangledown_{j-2} \varphi_{j-1} \triangledown_{j-1} \varphi_j \triangledown_j \dots \triangledown_{i-1} \varphi_i,$$

equals the function computed by

$$\varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \dots \triangledown_{j-2} \varphi_{j-1} \wedge \varphi_j \triangledown_j \dots \wedge \varphi_i$$

However if $\varphi_i(x) = 0$, then intuitively this formula outputs 0 no matter what happens on the to the "left" of φ_i in the formula. Thus, we might as well assume on the "left" that $\varphi_i(x) = \varphi_j(x) = 1$. Thus we get that the function computed at gate $\triangledown_{i-1}$ is also computed by the following simplified formula:

$$\varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \dots \triangledown_{j-2} \varphi_{j-1} \wedge 1 \triangledown_j \dots \wedge \varphi_i.$$

which equals

$$\varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \dots \triangledown_{j-2} \varphi_{j-1} \triangledown_j \dots \wedge \varphi_i.$$

Thus, while the original subformula of φ computed at gate $\triangledown_{i-1}$ given by

$$\varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \dots \varphi_{j-1} \triangledown_{j-1} \varphi_j \dots \triangledown_{i-1} \varphi_i$$

had size $\sum_{k\in[i]} |\varphi_k|$, the new equivalent formula given by

$$\varphi_1 \triangledown_1 \varphi_2 \triangledown_2 \ \dots \ \varphi_{j-1} \triangledown_j \varphi_{j+1} \dots \ \wedge \ \varphi_i$$

has the smaller size $\sum_{k\in[i]} |\varphi_k| - |\varphi_j| < \sum_{k\in[i]} |\varphi_k|$ which contradicts the optimality of φ for f.

It remains to chose a value for d. We need to satisfy that

$$\frac{d-1}{2} > 2^{2\cdot\frac{\mathsf{L}(f)-1}{d-1}\log n(1+o(1))}.$$

By Theorem 13, we have that $\mathsf{L}(f) \leq (1+o(1))\frac{2^n}{\log n}$. So setting $d = \frac{10}{n}\cdot 2^n$, we get that

$$2^{2\cdot\frac{\mathsf{L}(f)-1}{d-1}\log n(1+o(1))} \leq 2^{2n(1/10+o(1))} \leq d = \frac{10}{n}\cdot 2^n$$ ◀

Using this lemma, we prove a "bottom-up" search to decision reduction for Search-MFSP.

▶ **Theorem 54.** *There is a deterministic "bottom-up" algorithm solving* Search-MFSP *on inputs of length N given access to an oracle that solves* MFSP *on instances of length $2N$ that runs in time $O(N^3\cdot|\mathsf{CanonOpt}_{(\frac{10}{n}\cdot 2^n)}\mathsf{Formulas}(f)|^2)$ where f is the input truth table of length N.*

Algorithm 4 A bottom up search to decision reduction.

```
procedure OptimalFormula($f$)
    ▷ Given the length-$N$ truth table of a function $f$ that takes $n$-inputs, this procedure finds an optimal formula computing $f$
    Set $Candidates_{(1)} = \emptyset$.
    Let $OptForm$ be a empty lookup table.
    for each size one formula $\varphi$ on $n$-inputs do
        Let $q$ be the function computed by $\varphi$.
        Add $q$ to $Candidates_{(1)}$.
        Let $OptForm(q) = \varphi$.
    end for
    Set $s = 1$.
    while $s < \mathsf{L}(f)$ do
        Set $Candidates_{(s+1)} \leftarrow \emptyset$.
        for every pair $g, h$ in $Candidates$ and every gate $\triangledown \in \{\wedge, \vee\}$ do
            Let $q$ be the function computed by $g \triangledown h$.
            if $\mathsf{L}(q) = \mathsf{L}(g) + \mathsf{L}(h)$ and $L(\mathsf{Select}[f, q]) \leq \mathsf{L}(f) + \frac{10}{n}\cdot 2^n$ then
                Add $q$ to $Candidates_{(s+1)}$.
                Set $OptForm(q)$ to the formula given by $OptForm(g) \triangledown OptForm(h)$.
            end if
        end for
        Set $s = s + 1$.
    end while
    return $OptForm(f)$.
end procedure
```

Proof. The pseudocode for our reduction is presented in Algorithm 4.

Since this algorithm is weaker than the one presented in Theorem 33, we only sketch the main observation needed to see that the "test" implicit in Algorithm 4 that

$$\mathsf{L}(\mathsf{Select}[f,q]) - \mathsf{L}(f) \le d \le \frac{10}{n} \cdot 2^n$$

is passed by any function q that is computed by some gate in an optimal formula. The bound on the total number of functions that pass this test is given by Lemma 24.

Fix a function f on n-inputs and set $N = 2^n$. The correctness of this algorithm follows from showing that if φ is an optimal formula for f and q is an n-input function computed by the ith gate node in φ, then

$$\mathsf{L}(\mathsf{Select}[f,q]) - \mathsf{L}(f) \le d$$

where d is the depth of φ. If there were the case, then

$$\mathsf{L}(\mathsf{Select}[f,q]) - \mathsf{L}(f) \le \frac{10}{n} \cdot 2^n$$

using the depth bound on optimal DeMorgan formulas from Lemma 53.

We now show that

$$\mathsf{L}(\mathsf{Select}[f,q]) - \mathsf{L}(f) \le d$$

by producing a formula φ' for $\mathsf{L}(\mathsf{Select}[f,q])$ of size at most $\mathsf{L}(f) + d$.

Before, we give our formula construction of φ', we give an example of what our construction does that will hopefully be enough to convince the reader. To give an example,if $\varphi = x_1 \vee x_2 \wedge x_3$ and x_1 computes q, then $\varphi' = x_1 \vee (x_2 \wedge \neg z) \wedge (x_3 \vee z)$.

We formally construct φ' as follows. Recall we assumed that φ has depth d that q is the function computed by the ith gate in φ. Then, we can write

$$\varphi = \varphi_i \nabla_{i+1} \varphi_{i+1} \nabla_{i+2} \ldots \nabla_k \varphi_k$$

(associating from left to right) where $k \le d$ and $\varphi_i, \ldots, \varphi_{k+1}$ are subformulas of φ and $\nabla_i, \ldots, \nabla_k$ are the gates connecting those subformulas in φ and φ_i computes q.

We can then construct φ' by replacing each φ_j in φ for $i+1 \le j \le k$ with a new formula φ'_j given by

$$\varphi'_j = \begin{cases} \varphi_j \wedge \neg z, & \text{if } \nabla_j = \vee \\ \varphi_j \vee z, & \text{if } \nabla_j = \wedge \end{cases} .$$

Then

$$\varphi' = \varphi_i \nabla_{i+1} \varphi'_{i+1} \nabla_{i+2} \ldots \nabla_k \varphi'_k$$

computes $\mathsf{Select}[f,q]$ because these φ'_j are chosen so that ∇_j will always just output its other input when $z = 1$.

Hence,

$$\mathsf{L}(\mathsf{Select}[f,q]) - \mathsf{L}(f) \le d$$ ◀

References

1 Eric Allender. The new complexity landscape around circuit minimization. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings*, volume 12038 of *Lecture Notes in Computer Science*, pages 3–16. Springer, 2020. doi:10.1007/978-3-030-40608-0_1.

2 Eric Allender, Harry Buhrman, Michal Koucký, Dieter van Melkebeek, and Detlef Ronneburger. Power from random strings. *SIAM J. Comput.*, 35(6):1467–1493, 2006.

3 Eric Allender and Bireswar Das. Zero knowledge and circuit minimization. *Inf. Comput.*, 256:2–8, 2017.

4 Eric Allender, Michal Koucký, Detlef Ronneburger, and Sambuddha Roy. The pervasive reach of resource-bounded kolmogorov complexity in computational complexity theory. *Journal of Computer and System Sciences*, 77(1):14–40, 2011.

5 David Buchfuhrer and Christopher Umans. The complexity of boolean formula minimization. *J. Comput. Syst. Sci.*, 77(1):142–153, 2011.

6 Marco L. Carmosino, Russell Impagliazzo, Valentine Kabanets, and Antonina Kolokolova. Learning algorithms from natural proofs. In *Proceedings of the 31st Conference on Computational Complexity*, 2016.

7 Alexander Golovnev, Rahul Ilango, Russell Impagliazzo, Valentine Kabanets, Antonina Kolokolova, and Avishay Tal. Ac^0[p] lower bounds against MCSP via the coin problem. In *ICALP*, volume 132 of *LIPICS*, pages 66:1–66:15, 2019.

8 Shuichi Hirahara. Non-black-box worst-case to average-case reductions within NP. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 247–258, 2018.

9 Shuichi Hirahara, Igor Carboni Oliveira, and Rahul Santhanam. NP-hardness of minimum circuit size problem for OR-AND-MOD circuits. In *33rd Computational Complexity Conference, CCC*, volume 102, pages 5:1–5:31, 2018.

10 Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 73–79, 2000.

11 S. A. Lozhkin. Tighter bounds on the complexity of control systems from some classes. *Mat. Voprosy Kibernetiki 6*, pages 189–214 (in Russian), 1996.

12 Oleg B. Lupanov. Complexity of formula realization of functions of logical algebra. *Problemy Kibernetiki*, 3:61–80, 1960.

13 William J. Masek. Some NP-complete set covering problems. Unpublished Manuscript, 1979.

14 Cody D. Murray and R. Ryan Williams. On the (non) np-hardness of computing circuit complexity. *Theory of Computing*, 13(1):1–22, 2017.

15 Erkki Mäkinen. Generating random binary trees — a survey. *Information Sciences*, 115(1):123–136, 1999.

16 Nicholas Pippenger. Information theory and the complexity of boolean functions. *Mathematical Systems Theory*, 10:129–167, January 1977.

17 Alexander A. Razborov and Steven Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, 1997.

18 Michael Rudow. Discrete logarithm and minimum circuit size. *Inf. Process. Lett.*, 128:1–4, 2017.

19 Rahul Santhanam. Pseudorandomness and the minimum circuit size problem. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS*, volume 151 of *LIPIcs*, pages 68:1–68:26, 2020.

20 B. A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *IEEE Ann. Hist. Comput.*, 6(4):384–400, October 1984.