The Opacity of Backbones*

Lane A. Hemaspaandra[a,1,*], David E. Narváez[a,2]

[a]*Department of Computer Science, University of Rochester, Rochester, NY, 14627, USA*

Abstract

This paper approaches, using structural complexity theory, the question of whether there is a chasm between knowing an object exists and getting one's hands on the object or its properties. In particular, we study the nontransparency of so-called backbones. A backbone of a boolean formula F is a collection S of its variables for which there is a unique partial assignment a_S such that $F[a_S]$ is satisfiable [22, 29]. We show that, under the widely believed assumption that integer factoring is hard, there exist sets of boolean formulas that have obvious, nontrivial backbones yet finding the values, a_S, of those backbones is intractable. We also show that, under the same assumption, there exist sets of boolean formulas that obviously have large backbones yet producing such a backbone S is intractable. Furthermore, we show that if integer factoring is not merely worst-case hard but is frequently hard, as is widely believed, then the frequency of hardness in our two results is not too much less than that frequency. These results hold more generally, namely, in the settings where, respectively, one's assumption is that $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ or that some problem in $\mathrm{NP} \cap \mathrm{coNP}$ is frequently hard.

Keywords: backbones, frequency of hardness, $\mathrm{NP} \cap \mathrm{coNP}$, structural complexity theory

1. Introduction

An important concept in the study of the SAT problem is the notion of backbones. The term was first used by Monasson et al. [22], and the following formal definition was provided by Williams, Gomes, and Selman [29].

Definition 1.1. *Let F be a boolean formula. A collection S of the variables of F is said to be a* backbone *if there is a unique partial assignment a_S such that $F[a_S]$ is satisfiable.*

In that definition, a_S assigns a value (true or false) to each variable in S, and $F[a_S]$ is a shorthand meaning F except with each variable in S assigned the value specified for it in a_S. A backbone S is *nontrivial* if $S \neq \emptyset$. The *size* of a backbone S is the number of variables in S. For a backbone S (for formula F), we say that a_S is the *value* of the backbone S.

For example, every satisfiable formula has the trivial backbone $S = \emptyset$. The formula $x_1 \wedge \overline{x_2}$ has four backbones, $\emptyset$, $\{x_1\}$, $\{x_2\}$, and $\{x_1, x_2\}$, with respectively the values (listing values as

*A preliminary version of this paper appeared in AAAI 2017 [12].

*Corresponding author. URL: https://www.cs.rochester.edu/u/lane.

[1]Supported in part by NSF grant CCF-2006496.

[2]Supported in part by NSF grant CCF-2030859 to the Computing Research Association for the CIFellows Project. Work done in part while at the Rochester Institute of Technology.

bit-vectors giving the assignments in the lexicographical order of the names of the variables in S) ϵ, 1, 0, and 10. The formula $x_1 \vee \overline{x_2}$ has no nontrivial backbones. (Every formula that has a backbone will have a maximum backbone—a backbone that every other backbone is a subset of. This is clear since if S_1 and S_2 are backbones of F, then so is $S_1 \cup S_2$. Backbone variables have been called "frozen variables," because each of them is the same over all satisfying assignments.)

As Williams, Gomes, and Selman [29] note, "backbone variables are useful in studying the properties of the solution space of a... problem."[3]

And that surely is so. But it is natural to hope to go beyond that and suspect that if formulas have backbones, we can use those to help SAT solvers. After all, if one is seeking to get one's hands on a satisfying assignment of an F that has a backbone, one need but substitute in the value of the backbone to have put all its variables to bed as to one's search, and thus to "only" have all the other variables to worry about.

The goal of the present paper is to understand, at least in a theoretical sense, the difficulty of—the potential obstacles to doing—what we just suggested. We will argue that even for cases when one can quickly (i.e., in polynomial time) recognize that a formula has at least one nontrivial backbone, it can be intractable to find one such backbone. And we will argue that even for cases when one can quickly (i.e., in polynomial time) find a large, nontrivial backbone, it can be intractable to find the value of that backbone. In particular, we will show that if integer factoring is hard, then both the just-made claims hold. Integer factoring is widely believed to be hard; indeed, if it were in polynomial time, RSA (the Rivest-Shamir-Adleman cryptosystem) itself would fall.

In fact, integer factoring is even believed to be hard on average. And we will be inspired by that to go beyond the strength of the results mentioned above. Regarding our results mentioned above, one might worry that the "intractability" might be very infrequent, i.e., merely a rare, worst-case behavior. But we will argue that if integer factoring—or indeed any problem in NP $\cap$ coNP—is frequently hard, then the bad behavior types we mention above happen "almost" as often: If the frequency of hardness of integer factoring is $d(n)$ for strings up to length n, then for some $\epsilon > 0$ the frequency of hardness of our problems is $d(n^\epsilon)$. (We are not defining here what is being counted by "$d(\cdot)$," but that will be covered rigorously in Section 2.2. Very loosely put, we are speaking, within a framework such as "infinitely often" or "almost everywhere" as to which lengths n this holds for, about how many errors efficient heuristic algorithms inherently must make.)

None of this means that backbones are not an excellent, important concept. Rather, this is saying—proving, in fact, assuming that integer factoring is as hard as is generally believed—that although the definition of backbone is merely about a backbone existing, one needs to be aware that going from a backbone existing to finding a backbone, and going from having a backbone to knowing its value, can be computationally expensive challenges.

The following section presents our results, and then the section after that, which we have placed after the results section so that the reader is familiar with the results and proofs before the related-work discussion, covers the related work. After that is the conclusion and a technical appendix.

[3]We mention in passing that backbones also are very interesting for problems even harder than SAT, such as model counting (see Gomes, Sabharwal, and Selman, [8])—i.e., #SAT [28]: counting the number of satisfying assignments of a propositional boolean formula—especially given the currently huge runtime differences between SAT-solvers and model counters. After all, a backbone S for a formula means all the solutions that one is seeking to count are trapped within some subspace. That is, if a formula F has k variables, we can view the assignments to its variables as the points on a k-dimensional hypercube H. If that formula F has a backbone S, then all of F's solutions fall within the $(k - \|S\|)$-dimensional subhypercube of H induced by the assignment (we here use the notation of Definition 1.1) a_S.

2. Results

Section 2.1 will formulate our results without focusing on density. Then in Section 2.2 we will discuss how the frequency of hardness of sets of the type we have discussed is related to that of the sets in NP ∩ coNP having the highest frequencies of hardness.

2.1. Core results

We first look at whether there can be simple sets of formulas for which one can easily compute/obtain a nontrivial backbone, yet one cannot easily find the value of that backbone.

Our basic result on this is stated below as Theorem 2.1. In this and most of our results, we state as our hypothesis not that "integer factoring cannot be done in polynomial time," but rather that "P $\neq$ NP ∩ coNP." This in fact makes our claims stronger ones than if they had as their hypotheses "integer factoring cannot be done in polynomial time," since it is well-known (because the decision version of integer factorization is itself in NP ∩ coNP) that "integer factoring cannot be done in polynomial time" implies "P $\neq$ NP ∩ coNP." SAT will, as usual, denote the set of satisfiable (propositional) boolean formulas. (We do not assume that SAT by definition is restricted to CNF formulas.)

Theorem 2.1. *If* P $\neq$ NP ∩ coNP*, then there exists a set* $A \in$ P*,* $A \subseteq$ SAT*, of boolean formulas such that:*

1. *There is a polynomial-time computable function* f *such that* $(\forall F \in A)[f(F)$ *outputs a nontrivial backbone of* $F]$*.*
2. *There does not exist any polynomial-time computable function* g *such that* $g(F)$ *computes the value of backbone* $f(F)$*.*

Theorem 2.1 remains true even if one restricts the backbones found by f to be of size 1. We state that, in a slightly more general form, as follows.

Theorem 2.2. *Let* $k \in \{1, 2, 3, \ldots\}$*. If* P $\neq$ NP ∩ coNP*, then there exists a set* $A \in$ P*,* $A \subseteq$ SAT*, of boolean formulas such that:*

1. *There is a polynomial-time computable function* f *such that* $(\forall F \in A)[f(F)$ *outputs a size-*k *backbone of* $F]$*.*
2. *There does not exist any polynomial-time computable function* g *such that* $g(F)$ *computes the value of backbone* $f(F)$*.*

We defer the proof of Theorem 2.2 (which itself implies Theorem 2.1) until after the statement of Theorem 2.3.

Now let us turn to the question of whether, when it is obvious that there is at least one nontrivial backbone, it can be hard to efficiently produce a nontrivial backbone. The following theorem shows that, if integer factoring is hard, the answer is yes.

Theorem 2.3. *If* P $\neq$ NP ∩ coNP*, then there exists a set* $A \in$ P*,* $A \subseteq$ SAT*, of boolean formulas (each having at least one variable) such that:*

1. *Each formula* $F \in A$ *has a backbone whose size is at least 49% of* F*'s total number of variables.*
2. *There does not exist any polynomial-time computable function* g *such that, on each* $F \in A$*,* $g(F)$ *outputs a backbone whose size is at least 49%—or even at least 2%—of* F*'s variables.*

PROOF OF THEOREMS 2.2 AND 2.3. Since they share a proof structure, we will prove Theorems 2.2 and 2.3 hand-in-hand, and in a rather narrative fashion. Each of those theorems starts with the assumption that P $\neq$ NP $\cap$ coNP. So let B be some set instantiating that, i.e., $B \in (\text{NP} \cap \text{coNP}) - \text{P}$. As all students learn when learning that SAT is NP-complete, we can efficiently transform the question of whether a machine accepts a particular string into a question about whether a certain boolean formula is satisfiable [3, 17, 20]. The original work that did that did not require (and did not *need* to require) that the thus-created boolean formula transparently revealed what machine and input had been the input to the transformation. But it was soon explicitly noted that one can easily ensure—in fact, the natural ways one would write the reduction typically tacitly achieve this—that the formula mapped to transparently reveals the machine and input that were the input to the transformation; see Galil [7] or our appendix.

Galil's observation can be summarized in the following strengthened version of the standard claim regarding the so-called Cook-Karp-Levin Reduction. Here and throughout this paper, as is standard, for any machine N we will use $L(N)$ to denote the language accepted by machine N, i.e., the set of strings accepted by machine N. Let N_1, N_2, ... be a fixed, standard enumeration of clocked, polynomial-time, nondeterministic Turing machines, and w.l.o.g. assume that N_i runs within time $n^i + i$ on inputs of length n, and that N_i and i are polynomially related in size and easily obtained from each other (note: by the size of a natural number i we mean the number of bits in the binary representation of i). There is a function $r_{\textit{Galil-Cook}}$ (for conciseness, we are writing Galil-Cook rather than Galil-Cook/Karp/Levin, although this version is closer to the setting of Karp and Levin than to that of Cook, since Cook used Turing reductions rather than many-one reductions) such that

1. For each N_i and x: $x \in L(N_i)$ if and only if $r_{\textit{Galil-Cook}}(N_i, x) \in \text{SAT}$.
2. There is a polynomial p such that $r_{\textit{Galil-Cook}}(N_i, x)$ runs within time polynomial (in particular, with p being the polynomial) in $|N_i|$ and $|x|^i + i$.
3. There is a polynomial-time function s such that for each N_i and x, $s(r_{\textit{Galil-Cook}}(N_i, x))$ outputs the pair (N_i, x).

We will be using two separate applications of the r function in our construction. But we need those two applications to be variable-disjoint. We will need this as otherwise we'd have interference with some of our claims about sizes of backbones and which variables are fixed and how many variables we have. These are requirements not present in any previous work that used the r function of Galil-Cook. We also will want to be able to have some literal names (in particular, literal names using "z" that will be of the form z_ℓ, z'_ℓ, $\overline{z_\ell}$, or $\overline{z'_\ell}$, for all ℓ) available to us that we know are not part of the output of any application of the Galil-Cook r function; we need them as our construction involves not just two applications of the r function but also some additional variables. We can accomplish all the special requirements just mentioned as follows. We will, w.l.o.g., assume that in the output of the Galil-Cook function $r_{\textit{Galil-Cook}}(N_i, x)$, every variable is of the form x_j (the x there is not a generic example of a letter, but really means the letter "x" just as "z" earlier really means the letter "z"), where j itself, when viewed as a pair of positive integers via the standard fixed correspondence between $\mathbb{N}^+$ and $\mathbb{N}^+ \times \mathbb{N}^+$, has N_i as its first component or actually, to be completely precise, the positive integer corresponding to N_i in the standard fixed correspondence between positive integers and strings. Though not all implementations of the Galil-Cook r function need have this property (and in fact, none has previously satisfied it as far as we know), we claim that one can implement a legal Galil-Cook r function in such a way that it has this property yet still has the property that this r function will have a polynomial-time inversion function s satisfying

the behavior for s mentioned above. (For those wanting more information on how such a function $r_{\textit{Galil-Cook}}(N_i, x)$ can be implemented that has all the properties claimed above, we have included as a technical appendix, a detailed construction we have built that accomplishes this.)

We now can specify the sets A needed by Theorems 2.2 and 2.3. Recall we have (thanks to the assumptions of the theorems) fixed a set $B \in (\mathrm{NP} \cap \mathrm{coNP}) - \mathrm{P}$. $B \in \mathrm{NP}$, so let i be a positive integer such that N_i is a machine from the abovementioned standard enumeration such that $L(N_i) = B$. $\overline{B} \in \mathrm{NP}$, so let j be a positive integer such that N_j is a machine from the abovementioned standard enumeration, such that $L(N_j) = \overline{B}$. Fix any positive integer k. Then for the case of that fixed value k, the set A of Theorem 2.2 is as follows:

$$A_{3,k} = \{(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{\textit{Galil-Cook}}(N_j, x))) \mid x \in \Sigma^*\}.$$

One must keep in mind in what follows that, as per the previous paragraph, $r_{\textit{Galil-Cook}}$ never outputs literals with names involving subscripted zs or z's and the outputs of $r_{\textit{Galil-Cook}}(N_i, x)$ and $r_{\textit{Galil-Cook}}(N_j, x)$ share no variable names (since $i \neq j$).

Let us argue that $A_{3,k}$ indeed satisfies the requirements of the A for the "k" case of Theorem 2.2.

$\boldsymbol{A \in \mathbf{P}}$**:** Given a string y whose membership in A we are testing, we make sure y syntactically matches the form of the elements of A (i.e., elements of $A_{3,k}$). If it does, we then check that its k matches our k, and we use s to get decoded pairs (i', x') and (j'', x'') from the places in our parsing of y where we have formulas—call them $F_{\textit{left}}$ and $F_{\textit{right}}$—that we are hoping are the outputs of the r function. That is, if our input parses as

$$(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (F_{\textit{left}})) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (F_{\textit{right}}))\,,$$

then if $s(F_{\textit{left}})$ gives $(N_{i'}, x')$ our decoded pair is (i', x'), and $F_{\textit{right}}$ is handled analogously. We also check to make sure that $x' = x''$, $i = i'$, and $j = j''$. If anything mentioned so far fails, then $y \notin A$. Otherwise, we check to make sure that $r_{\textit{Galil-Cook}}(N_i, x') = F_{\textit{left}}$ and $r_{\textit{Galil-Cook}}(N_j, x') = F_{\textit{right}}$, and reject if either equality fails to hold. (Those checks are *not* superfluous. s by definition has to correctly invert on strings that are the true outputs of $r_{\textit{Galil-Cook}}$, but we did not assume that s might not output sneaky garbage when given other input values, and since $F_{\textit{left}}$ and $F_{\textit{right}}$ are coming from our arbitrary input y, they could be anything. However, the check we just made defangs the danger just mentioned.) If we have reached this point, we indeed have determined that $y \in A$, and for each $y \in A$ we will successfully reach this point.

$\boldsymbol{A \subseteq \mathbf{SAT}}$**:** For each x, either $x \in B$ or $x \notin B$. In the former case ($x \in B$), $r_{\textit{Galil-Cook}}(N_i, x) \in \mathrm{SAT}$ and so the left disjunct of $(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{\textit{Galil-Cook}}(N_j, x)))$ can be made true using that satisfying assignment and setting each z_ℓ to true. On the other hand, if $x \notin B$, then $r_{\textit{Galil-Cook}}(N_j, x) \in \mathrm{SAT}$ and so the whole formula can be made true using that satisfying assignment and setting each z_ℓ to false.

There is a polynomial-time computable function $\boldsymbol{f}$ such that $(\forall \boldsymbol{F} \in \boldsymbol{A})[\boldsymbol{f}(\boldsymbol{F})$ outputs a nontrivial backbone of $\boldsymbol{F}$]: On input $F \in A$, f will simply output $\{z_1, z_2, \ldots, z_k\}$, which is a nontrivial backbone of F. Why is it a nontrivial backbone? If the x embedded in F satisfies $x \in B$, then not only does $r_{\textit{Galil-Cook}}(N_i, x) \in \mathrm{SAT}$ hold, but also $r_{\textit{Galil-Cook}}(N_j, x) \notin \mathrm{SAT}$ must hold, since otherwise we would have $x \notin B \wedge x \in B$, an impossibility. So if the x embedded in F satisfies $x \in B$, then there are satisfying assignments of $(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{\textit{Galil-Cook}}(N_j, x)))$, and every one of them has each z_ℓ set to true. Similarly, if the x embedded in F satisfies $x \notin B$, then our long formula has satisfying assignments, and every one of them has each z_ℓ set to false. Thus $\{z_1, z_2, \ldots, z_k\}$ indeed is a size-k backbone.

There does not exist any polynomial-time computable function g such that $g(F)$ computes the value of backbone $f(F)$: Suppose by way of contradiction that such a polynomial-time computable function g does exist. Then we would have that $B \in \mathrm{P}$, by the following algorithm. Let f be the function constructed in the previous paragraph, i.e., the one that outputs $\{z_1, z_2, \ldots, z_k\}$ when $F \in A$. Given x, in polynomial time—g and f are polynomial-time computable, and although r in general is not since its running time's polynomial degree varies with its first argument and so is not uniformly polynomial, r here is used only for the first-component values N_i and N_j and under that restriction it indeed is polynomial-time computable—compute

$$g(f((z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{\textit{Galil-Cook}}(N_j, x))))).$$

This must either tell us that the z_ℓs are true in all satisfying assignments, which tells us that it is the left disjunct that is satisfiable and thus $x \in B$, or it will tell us that the z_ℓs are false in all satisfying assignments, from which we similarly can correctly conclude that $x \notin B$. So $B \in \mathrm{P}$, yet we chose B so as to satisfy $B \in (\mathrm{NP} \cap \mathrm{coNP}) - \mathrm{P}$. Thus our assumption that such a g exists is contradicted.

That ends our proof of Theorem 2.2—and so implicitly also of Theorem 2.1, since Theorem 2.1 follows immediately from Theorem 2.2.

Having seen the above proof, the reader will not need a detailed treatment of the proof of Theorem 2.3. Rather, we describe how to convert the above construction into one that proves Theorem 2.3. Recall that for the "k" case of Theorem 2.2 our set A was

$$\{(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{\textit{Galil-Cook}}(N_j, x))) \mid x \in \Sigma^*\}.$$

For Theorem 2.3, let us use almost the same set. Except we will make two types of changes. First, in the above, replace the two occurrences of k each with the smallest positive integer m' satisfying

$$\frac{m'}{\textit{numvars}(r_{\textit{Galil-Cook}}(N_i, x)) + \textit{numvars}(r_{\textit{Galil-Cook}}(N_j, x)) + 2m'} \geq \frac{49}{100},$$

where *numvars* counts the number of variables in a formula, e.g., $\textit{numvars}(\overline{x_1} \wedge x_2 \wedge \overline{x_2}) = 2$, due to the variables x_1 and x_2. Let m henceforward denote that value, i.e., the smallest (positive integer) m' that satisfies the above equation. Second, in the right disjunct, change each $\overline{z_\ell}$ to $\overline{z'_\ell}$.

Note that if $x \in B$, then $\{z_1, z_2, \ldots, z_m\}$ is a backbone whose value is the assignment of true to each variable, and that contains at least 49% of the variables in the formula that x put into A. Similarly, if $x \notin B$, then $\{z'_1, z'_2, \ldots, z'_m\}$ is a backbone whose value is the assignment of false to each variable, and that contains at least 49% of the variables in the formula that x put into A. It also is straightforward to see that our thus-created set A belong to P and satisfies $A \subseteq \mathrm{SAT}$.

So the only condition of Theorem 2.3 that we still need to show holds is the claim that, for the just-described A, there does not exist any polynomial-time computable function g such that, on each $F \in A$, $g(F)$ outputs a backbone whose size is at least 2% of F's variables. Suppose by way of contradiction that such a function g does exist. We claim that would yield a polynomial-time algorithm for B, contradicting the assumption that $B \notin \mathrm{P}$. Let us give such a polynomial-time algorithm. To test whether $x \in B$, in polynomial time we create the formula in A that is put there by x, and we run our postulated polynomial-time g on that formula, and thus we get a backbone, call it S, that contains at least 2% of F's variables. Note that we ourselves do not get

to choose which large backbone g outputs, so we must be careful as to what we assume about the output backbone. We in particular certainly cannot assume that g happens to always output either $\{z_1, z_2, \ldots, z_m\}$ or $\{z'_1, z'_2, \ldots, z'_m\}$. But we don't need it to. Note that the two backbones just mentioned are variable-disjoint, and each contains 49% of F's variables.

Now, there are two cases. One case is that S contains at least one variable of the form z_ℓ or z'_ℓ. In that case we are done. If it contains at least one variable of the form z_ℓ then $x \in B$. Why? If $x \in B$, then the left-hand disjunct of the formula x puts into A is satisfiable and the right-hand disjunct is not. From the form of the formula, it is clear that each z_ℓ is always true in each satisfying assignment in this case, yet that for each z'_ℓ there are satisfying assignments where z'_ℓ is true and there are satisfying assignments where z'_ℓ is false. So if $x \in B$, no z'_ℓ can belong to any backbone.

By analogous reasoning, if S contains at least one variable of the form z'_ℓ then $x \notin B$. It follows from this and the above that S cannot possibly contain at least one variable that is a subscripted z and at least one variable that is a subscripted z', since then x would have to simultaneously belong and not belong to B.

The final case to consider is the one in which S does not contain at least one variable of the form z_ℓ or z'_ℓ. We argue that this case cannot happen. If this were to happen, then every variable of F other than the variables $\{z_1, z_2, \ldots, z_m, z'_1, z'_2, \ldots, z'_m\}$ must be part of the backbone, since S must involve at least 2% of the variables and $\{z_1, z_2, \ldots, z_m, z'_1, z'_2, \ldots, z'_m\}$ comprise at least 98% of the variables. But that is impossible. We know that the variables used in $r_{\textit{Galil-Cook}}(N_i, x)$ and $r_{\textit{Galil-Cook}}(N_j, x)$ are disjoint. So the variables in the one of those two that is not the one that is satisfiable can and do take on any value in some satisfying assignment, and so cannot be part of any backbone. The only remaining worry is the case where one of $r_{\textit{Galil-Cook}}(N_i, x)$ or $r_{\textit{Galil-Cook}}(N_j, x)$ contains no variables. However, the empty formula is by convention considered illegal, in cases such as here where the formulas are not considered to be trapped into DNF or CNF. There is a special convention regarding empty DNF and CNF formulas, but that is not relevant here.

We have thus established Theorem 2.3.

The 49% and 2% used in Theorem 2.3's statement and proof are not at all magic, but are just for concreteness. It is immediately clear that our above proof that establishes Theorem 2.3 is in fact tacitly (namely, if one changes—from the fixed constants we used in that proof—to instead the more flexible constants below) establishing the more general theorem that we now state as Theorem 2.4. Theorem 2.3 is the $\epsilon = 1$ special case of Theorem 2.4. Note that the smaller the ϵ the stronger the claim, and so the fact that Theorem 2.4 speaks only of $\epsilon \leq 1$ is not a weakness.

Theorem 2.4. *For each fixed ϵ, $0 < \epsilon \leq 1$, the following claim holds. If $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$, then there exists a set $A \in \mathrm{P}$, $A \subseteq \mathrm{SAT}$, of boolean formulas (each having at least one variable) such that:*

1. *Each formula $F \in A$ has a backbone whose size is at least $(50 - \epsilon)\%$ of F's total number of variables.*
2. *There does not exist any polynomial-time computable function g such that, on each $F \in A$, $g(F)$ outputs a backbone whose size is at least $(2\epsilon)\%$ of F's variables.*

2.2. Frequency of hardness

A skeptical person might worry about results of the previous section in the following way. (Here, $|F|$ will denote the number of bits in the representation of F.) "Just because something is hard, doesn't mean it is hard *often*. For example, consider Theorem 2.3. Perhaps there is a

polynomial-time function g' that, though it on infinitely many $F \in A$ fails to compute the value of the backbone $f(F)$, has the property that for each $F \in A$ for which it fails it then is correct on the (in lexicographical order) next $2^{2^{2^{|F|}}}$ elements of A. In this case, the theorem is indeed true, but it is a worst-case extreme that doesn't recognize that in reality the errors may be few and far—very, very far—between."

In this section, we address that completely reasonable and important worry. We show that if *even one problem* in NP ∩ coNP is frequently hard, then the sets in our previous sections can be made "almost" as frequently hard, in a sense of "almost" that we will make formal and specific. Since it is generally believed—for example due to the generally believed typical-case hardness of integer factoring—that there are sets in NP ∩ coNP that are quite frequently hard, it follows that the $2^{2^{2^{|F|}}}$ behavior our skeptic was speculating about cannot happen. Or at least, if that behavior did happen, then that would imply that every single problem in NP ∩ coNP has polynomial-time heuristic algorithms that make extraordinarily few errors.

Note that no one currently knows for sure how frequently-hard problems in NP ∩ coNP can be. But our results are showing that, whatever that frequency is, sets of the sort we've been constructing are hard "almost" as frequently. This can at first seem a bit of a strange notion to get one's head around, especially as complexity theory often doesn't pay much attention to frequency-of-hardness issues (though such issues in complexity theory can be traced back at least as far as the work of Schöning [26]). But this actually is analogous to something every computer science researcher knows well, namely, NP-completeness. No one today knows for sure whether any NP problems are not in P. But despite that the longstanding NP-completeness framework lets one right now, today, prove clearly for specific problems that *if* any NP problem is not in P then that specific problem is not in P. The results of this section are about an analogous type of argument, except regarding frequency of hardness.

We now give our frequency-of-hardness version of Theorem 2.1. A claim is said to hold *for almost every n* if there exists an n_0 beyond which the claims always holds, i.e., the claim fails at most at a finite number of values of n. (In the theorems of this section, n's universe is the natural numbers, $\{0, 1, 2, \ldots\}$. And we will defer the proving of this section's theorems until the end of this section, where we will argue that the results in effect follow from the constructions of the previous section.)

Theorem 2.5. *If h is any nondecreasing function and for some $B \in \mathrm{NP} \cap \mathrm{coNP}$ it holds that each polynomial-time algorithm, viewed as a heuristic algorithm for testing membership in B, for almost every n (respectively, for infinitely many n) errs on at least $h(n)$ of the strings whose length is at most n, then there exist an $\epsilon > 0$ and a set $A \in \mathrm{P}$, $A \subseteq \mathrm{SAT}$, of boolean formulas such that:*

1. *There is a polynomial-time computable function f such that $(\forall F \in A)[f(F)$ outputs a nontrivial backbone of $F]$.*
2. *Each polynomial-time computable function g will err (i.e., will fail to compute the value of backbone $f(F)$), for almost every n (respectively, for infinitely many n), on at least $h(n^\epsilon)$ of the strings in A of length at most n.*

The precisely analogous result holds for Theorem 2.2. The analogous results also hold for Theorems 2.3 and 2.4, and to be explicit, we state that for Theorem 2.3 as the following theorem (and from this the analogue for Theorem 2.4 will be implicitly clear).

Theorem 2.6. *If h is any nondecreasing function and for some $B \in \mathrm{NP} \cap \mathrm{coNP}$ it holds that each polynomial-time algorithm, viewed as a heuristic algorithm for testing membership in B, for almost every n (respectively, for infinitely many n) errs on at least $h(n)$ of the strings whose length is at most n, then there exist an $\epsilon > 0$ and a set $A \in \mathrm{P}$, $A \subseteq \mathrm{SAT}$, of boolean formulas such that:*

1. *Each formula $F \in A$ has a backbone whose size is at least 49% of F's total number of variables.*
2. *Each polynomial-time computable function g will err (i.e., will fail to compute a set of size at least 2% of F's variables that is a backbone of F), for almost every n (respectively, for infinitely many n), on at least $h(n^\epsilon)$ of the strings in A of length at most n.*

What the above theorems say, looking at the contrapositives to the above results, is that if any of our above cases have polynomial-time heuristic algorithms that don't make errors too frequently, then *every single set in* $\mathrm{NP} \cap \mathrm{coNP}$ (even those related to integer factoring) has polynomial-time heuristic algorithms that don't make errors too frequently.

To make the meaning of the above results clearer, and to be completely open with our readers, it is important to have a frank discussion about the effect of the "ϵ" in the above results. Let us do this in two steps. First, we give as concrete examples two central types of growth rates that fall between polynomial and exponential. And second, we discuss how innocuous or noninnocuous the "ϵ" above is.

As to our examples, suppose that for some fixed $c > 0$ a particular function $h(n)$ satisfies $h(n) = 2^{\Omega((\log n)^c)}$. Note that for each fixed $\epsilon > 0$, it holds that the function $h'(n)$ defined as $h(n^\epsilon)$ itself satisfies the same bound, $h'(n) = 2^{\Omega((\log n)^c)}$. (Of course, the constant implicit in the "Ω" potentially has become smaller in the latter case.) Similarly, suppose that a particular function $h(n)$ satisfies $h(n) = 2^{n^{\Omega(1)}}$. Then for each fixed $\epsilon > 0$ it will hold that $h(n^\epsilon) = 2^{n^{\Omega(1)}}$.

The above at a casual glance might suggest that the weakening of the frequency claims between the most frequently hard problems in $\mathrm{NP} \cap \mathrm{coNP}$ and our problems is a "mere" changing of a constant. In some sense it is, but constants that are standing on the shoulders of exponents have more of a kick than constants sitting on the ground floor. And so as a practical matter, the difference in the actual numbers when one substitutes in for them can be large. On the other hand, polynomial-time reductions sit at the heart of computer science's formalization of its problems, and density distortions from n to n^ϵ based on the stretching of reductions are simply inherent in the standard approaches of theory, since those are the distortions one gets due to polynomial-time reductions being able to stretch their inputs to length $n^{1/\epsilon}$, i.e., polynomially. For example, it is well known that if B is an NP-complete set, then for every $\epsilon > 0$ it holds that B is polynomial-time isomorphic (which is theoretical computer science's strongest standard notion of them being "essentially the same problem") to some set B' that contains at most 2^{n^ϵ} strings at each length n.

Simply put, the "almost" in our "almost as frequent" claims is the natural, strong claim, judged by the amounts of slack that in theoretical computer are considered innocuous. And the results do give insight into how much the density does or does not change, e.g., the first above example shows that quasi-polynomial lower bounds on error frequency remain quasi-polynomial lower bounds on error frequency. However, on the other hand, there is a weakening, and even though it is in a "constant," that constant is in an exponent and so can alter the numerical frequency quite a bit.[4]

[4]In reality, how nastily small will the "ϵ" be? From looking inside the proofs of the results and thinking hard about the lengths of the formulas involved in the proofs (and resulting from the "Galil" version of Cook-Karp-Levin's Theorem that we will be discussing in the next section), one can see that ϵ's value is primarily controlled by the

Our use of B and A here reflects that of the theorems in this section. The crucial thing to note is that the mapping from strings x (as to whether they belong to B) into the string that x puts into A is (a) polynomial-time computable (and so the one string that x puts into A is at most polynomially longer than x), and (b) one-to-one.

So any collection of m instances up to a given length n that fool a particular polynomial-time algorithm for B are associated with at least m distinct instances in A all of length at most n^q (where the polynomial bound on the length of the formula that x, $|x| = n$, puts into A is that its length is n^q or less[5]). So if one had an algorithm for the "A" set such that the algorithm had at most m' errors on the strings up to length n^q, it would certainly imply an algorithm for B that up to length n made at most m' errors. Namely, one's heuristic of that form for B would be to take x, map it to the string it put into A, and then run the heuristic for A on that string.

The results of this section are the immediate consequences of this observation, applied to the constructions/results of the previous section. To make completely clear that that is the case and why it is the case, we now provide a more detailed explanation of the proof of one of this section's theorems, namely, Theorem 2.5.

Proof of Theorem 2.5. Let A be defined as $A_{3,1}$ in Section 2.1, i.e.:

$$A = A_{3,1} = \{(z_1 \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge (r_{\textit{Galil-Cook}}(N_j, x))) \mid x \in \Sigma^*\}$$

where N_i and N_j are Turing machines such that $L(N_i) = B$ and $L(N_j) = \overline{B}$. We know from our discussion in Section 2.1 that $A \in \mathrm{P}$ and that, given any formula $F \in A$, $\{z_1\}$ is a nontrivial backbone of that formula, thus the function $f(F) = \{z_1\}$ satisfies the requirements of the first part of Theorem 2.5.

Since $r_{\textit{Galil-Cook}}$ runs in polynomial time, there exist polynomials p_i and p_j of equal degree q such that the length of the formula $r_{\textit{Galil-Cook}}(N_i, x)$ is at most $p_i(|x|)$ and the length of the formula $r_{\textit{Galil-Cook}}(N_j, x)$ is at most $p_j(|x|)$. Note that there will exist natural numbers k and N such that for all $n > N$,

$$k\, n^q \geq |\, (z_1 \wedge (r_{\textit{Galil-Cook}}(N_i, x))) \vee (\overline{z_1} \wedge (r_{\textit{Galil-Cook}}(N_j, x)))\,|$$

for all strings x whose length is at most n. Let $n_B > N$ be a natural number such that every polynomial-time algorithm, viewed as a heuristic for testing membership in B, errs on at least $h(n_B)$ of the strings whose length is at most n_B. We claim that every polynomial-time algorithm, viewed as a heuristic for computing the value of $f(F) = \{z_1\}$ for inputs $F \in A$, errs on at least

running time of the NP machines for the NP sets B and $\overline{B}$ from the theorems of this section. If those machines run in time $\mathcal{O}(n^q)$, then ϵ will vary, viewed as a function of q, roughly as (some constant times) the inverse of q. For the particular case of Theorem 2.5, for example, our coming proof will make it clear that, where q is as just stated, for any $\delta > 0$ one can make the ϵ in the theorem's $h(n^\epsilon)$ be $\frac{1}{q} - \delta$, i.e., the theorem's lower bound is $h(n^{\frac{1}{q}-\delta})$. Indeed, the penultimate paragraph of that proof will actually establish—in the text immediately before it simplifies by introducing δ—a slightly stronger bound, namely, that there will be a constant $c > 0$ (related to not just the degree of but also the multiplicative constant of the running-time bound on the NP machines for B and $\overline{B}$) such that the lower bound the theorem is speaking of can be taken as $h(c\, n^{\frac{1}{q}})$.

[5] We have for simplicity left out any lower-order terms and the leading-term constant, but that is legal except at $n \in \{0, 1\}$—since starting with $n = 2$ we can boost q if needed—and no finite set of values, such as $\{0, 1\}$ can cause problems to our theorem, as it is about the "infinitely-often" and "almost-everywhere" cases. However, such boosting does potentially interfere with the inverse-of-q relation mentioned in Footnote 4, and so if we wanted to maintain that, we would in this argument instead use a lowest-degree-possible monotonic polynomial bounding the growth rate.

$h(n_B)$ of the strings whose length is at most $k\,(n_B)^q$. Let us define n_A by $n_A = k\,(n_B)^q$. Making that variable substitution in our claim, we have that every polynomial-time algorithm, viewed as a heuristic for computing the value of $f(F) = \{z_1\}$ for inputs $F \in A$, errs on at least $h\left(k^{-\frac{1}{q}}\,(n_A)^{\frac{1}{q}}\right)$ of the strings whose length is at most n_A. For any $\delta > 0$ it certainly holds that, for almost all n, $k^{-\frac{1}{q}}\,n^{\frac{1}{q}} \geq n^{\frac{1}{q}-\delta}$, and thus, since h is nondecreasing, it certainly holds that, for almost all n, $h\left(k^{-\frac{1}{q}}\,n^{\frac{1}{q}}\right) \geq h\left(n^{\frac{1}{q}-\delta}\right)$. Depending on which part of the "respectively" in the theorem's statement one is speaking of, from the assumptions we have that almost every $n > N$ can take the role of n_B (respectively, infinitely many $n > N$ can take the role of n_B). Thus setting $\epsilon = \frac{1}{q} - \delta$ proves that A satisfies the second part of Theorem 2.5.

To prove our claim, notice that, by our choice of n_B, for all inputs x of length at most n_B the length of $(z_1 \wedge (r_{\mathit{Galil\text{-}Cook}}(N_i, x))) \vee (\overline{z_1} \wedge (r_{\mathit{Galil\text{-}Cook}}(N_j, x)))$ is at most $k\,(n_B)^q$. Assume, by contradiction, that there exists a polynomial-time algorithm g' that, viewed as a heuristic for computing the value of $f(F) = \{z_1\}$ for inputs $F \in A$, errs on fewer than $h(n_B)$ of the strings of length at most $k\,(n_B)^q$. Consider the following polynomial-time heuristic for testing membership in B: On input x, calculate $v = g'((z_1 \wedge (r_{\mathit{Galil\text{-}Cook}}(N_i, x))) \vee (\overline{z_1} \wedge (r_{\mathit{Galil\text{-}Cook}}(N_j, x))))$; if v sets z_1 to true then output $x \in B$ and if v sets z_1 to false output $x \notin B$. Based on our discussion in Section 2.1, this heuristic will err exactly when g' errs since, for instance, if v sets z_1 to true but the correct value sets z_1 to false that would imply $x \notin B$. But g' errs on fewer than $h(n_B)$ inputs of length at most $k\,n_B^q$, so the polynomial-time heuristic we just constructed errs on fewer than $h(n_B)$ inputs x of length at most n_B, a contradiction.

3. Related work

Our results can be viewed as part of a line of work that though interesting is, unfortunately, so underpopulated as to barely merit being called a line of work. The true inspiration for this work was an insightful structural complexity theory paper of Alan Demers and Allan Borodin [2] from the 1970s, which never appeared in any form other than as a technical report. Their paper in effect showed sufficient conditions for creating simple sets of satisfiable formulas such that it was unclear why they were satisfiable.

Borodin and Demers's work has been used only very rarely. In particular, it has been used to get characterizations regarding unambiguous computation [9], and Rothe and his collaborators have used it in various contexts to study the complexity of certificates [14, 25], see also Fenner et al. [6] and Valiant [27]. Also, one paper—Hemaspaandra, Hemaspaandra, and Menton [10]—shows that the work has a connection to an "applied" area, namely, that paper shows that some problems about the manipulation of elections have the property that if $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ then their search versions are not polynomial-time Turing reducible to their decision problems—a rare behavior among the most familiar seemingly hard sets in computer science, since so-called self-reducibility [21] is known to preclude that possibility for most standard NP-complete problems. The key issue that the 2020 paper of Hemaspaandra, Hemaspaandra, and Menton [10] left open is whether the type of techniques it used, descended from Borodin and Demers [2], might be relevant in other domains, or whether its results were a one-shot oddity. The present paper in effect is arguing that the former is the case. Backbones are a topic important in both theory and artificial intelligence. This paper shows that the inspiration of the line of work initiated by Borodin and Demers [2] can be used to establish the opacity of backbones. It is important to acknowledge that our proofs regarding

Section 2.1 are drawing on elements of the insights of Borodin and Demers [2], although in ways unanticipated by that paper.

This paper uses density transfer arguments in the context of Borodin-Demers arguments. To the best of our knowledge, the only paper to previously do that is the work—in the quite different context of computational social choice theory—of Hemaspaandra, Hemaspaandra, and Menton [10].

Section 2.2 is about frequency of hardness. Very loosely put, its theorems show—among other things—that the existence of a set in $\mathrm{NP} \cap \mathrm{coNP}$ on which all deterministic polynomial-time algorithms err, for almost every n, on at least $h(n)$ strings of length at most n implies that there exist an $\epsilon > 0$ and a set $A \in \mathrm{P}$, with $A \subseteq \mathrm{SAT}$, such that some particular, important task regarding backbones will, for each polynomial-time machine attempting to execute it correctly, fail for almost all n on at least $h(n^\epsilon)$ of the strings in A of length at most n. Note that in our results, all strings in A up to a given length are in some sense viewed as equally important. As a literature pointer to those interested in contrasting frequency issues, we mention that Erdélyi et al. [5] (see also [24]) showed that a broad range of NP-complete sets are in in so-called basic deterministic heuristic polynomial time relative to some so-called basic junta distribution; that claim, however, is about *distributional* decision problems. It is also known, from a long line of work in complexity theory, that if any NP-hard set has (under uniform weighting) a deterministic polynomial-time heuristic algorithm that asymptotically has a relatively low density of errors, then the polynomial hierarchy collapses (see the survey [15]).

This paper's work may seem somewhat of the flavor of a paper on the topic of "search versus decision," an important theme in complexity theory. However, one must be careful, if one says that about this paper, as to precisely what one means. For example, regarding Theorem 2.1, the theorem is certainly not saying that finding the asserted backbones the theorem is speaking of is harder than knowing whether they exist. In fact, both are easy. Rather, the theorem is saying that, although knowing that a formula in A has a nontrivial backbone is easy and finding such a backbone is easy, nonetheless finding the *value* of that backbone is, under the $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ hypothesis, not easy. Of course, it is true that stating that a value exists is easy here and finding it is hard. However, that is somewhat stretched as to being viewed as a typical case of search versus decision. Readers interested in work on search versus decision might wish to look for example at the paper of Bellare and Goldwasser [1] that shows a very nice search-versus-decision separation in cryptography. However, that paper requires as its supporting hypothesis an extremely strong assumption about double-exponential time classes separating. More recently, the work of Hemaspaandra, Hemaspaandra, and Menton [10] mentioned above shows search-versus-decision separations within the area of manipulative attacks on elections. There is also some coverage in the survey paper Hemaspaandra [11], and search versus decision is also of importance in the context of parallel computation (see, e.g., [18]), though parallel computation is not the subject of the present paper.

Finally, note that most of our results rely on the assumption that $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$, which as noted above is likely true, since if it is false then integer factoring is in P and the RSA encryption scheme falls. What results can one obtain if one allows oneself to assume only $\mathrm{P} \neq \mathrm{NP}$? That question is explored in a recent paper by the authors that was motivated by the present work [13]. However, that later paper's results, due to the weaker assumption, do not at all address the issues, central to the present paper, of outputting a backbone or outputting the value of a backbone. That paper studies not only backbones, but also another so-called "hidden structure" of formulas, namely, backdoors. As pointers to earlier studies of the complexity issues regarding backbones and

backdoors, we mention [23, 19, 4].

4. Conclusions

We argued, under assumptions widely believed to be true such as the hardness of integer factoring, that knowing a large backbone exists doesn't mean one can efficiently find a large backbone, and finding a nontrivial backbone doesn't mean one can efficiently find its value. Further, we showed that one can ensure that these effects are not very infrequent, but rather that they can be made to happen with "almost" the same density of occurrence as the error rates of the most densely hard sets in NP $\cap$ coNP.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

A preliminary version appeared in AAAI-17 [12]. We are grateful to the anonymous conference and journal reviewers for their helpful comments and suggestions.

Appendix: Construction of a Galil-Cook r function with the properties claimed in Section 2.1

For those who wish to be assured that a Galil-Cook "r" function can be implemented so as to have all the properties we have "without loss of generality" assumed in Section 2.1, we here provide such an implementation.

Let N_1, N_2, ... be as in Section 2.1. That is, it is a fixed, standard enumeration of clocked, polynomial-time, nondeterministic Turing machines, such that each N_i runs within time $n^i + i$ on inputs of length n, and N_i and i are polynomially related in size and easily obtained from each other. Fix *any* function r that implements the Cook-Karp-Levin reduction. That is, r is such that

1. for each N_i and x: $x \in L(N_i)$ if and only if $r(N_i, x) \in$ SAT.
2. there is a polynomial p such that $r(N_i, x)$ runs within time polynomial (in particular, with p being the polynomial) in $|N_i|$ and $|x|^i + i$.

It is very well known that such functions exist. Their existence—the Cook-Karp-Levin reduction—is proven in almost every textbook that covers NP-completeness (see, e.g., Hopcroft and Ullman [16]), and is the key moment that brings the theory of NP-completeness to life, by transferring the domain from machines to a concrete problem that itself can be used to show that other concrete NP problems are themselves NP-complete.

Note that the function r that we have thus fixed is *not* assumed to necessarily have an "inversion" function s, and is *not* assumed to necessarily avoid using literals involving the letter "z", and is *not* assumed to necessarily have the property that two applications of the r function are guaranteed to be variable-disjoint if they regard different machines (i.e., are not both about N_i for the same value i).

We now show how to use the above fixed function r as a building block to build our function $r_{\textit{Galil-Cook}}$, which will have all the properties just mentioned, yet will retain the time and reduction-to-SAT properties mentioned above regarding r. $r_{\textit{Galil-Cook}}$, when its first argument is N_i and its second argument is x, does the following. It simulates the run of r when its first argument is N_i and its second argument is x, and so computes the formula $r_{\textit{Galil-Cook}}(N_i, x)$, which we will henceforth denote by F for conciseness of notation. It then counts the number of variables occurring in that formula F; let us denote that number by γ. (So for example if the formula F is $z_1 \wedge \overline{z_1} \wedge \overline{w}$, then $\gamma = 2$, as there are two variables, z_1 and w.) Now, let F' denote F, except each variable a in F will be replaced in F' by the variable $x_{\langle N_i,q\rangle}$, where q is the location of a in lexicographic order within the variables of F. ($\langle\cdot,\cdot\rangle$ is any standard, nice, easily computable, easily invertible pairing function.) If we view w as coming lexicographically before z_1, then in our example, F' would be $x_{\langle N_i,2\rangle} \wedge \overline{x_{\langle N_i,2\rangle}} \wedge \overline{x_{\langle N_i,1\rangle}}$. Despite the pairings used, this increases the length of F by at most a multiplicative factor of the number of bits of N_i. (Since each of our uses of $r_{\textit{Galil-Cook}}$ in Section 2.1 only used the function on some two hypothetical, fixed machines, the time and length-of-output effect of this variable-renaming is at most a multiplicative constant (that depends on the two machines), and so is negligible in standard complexity-analysis terms). But although using the same trick to encode x into the output by pairing it too into the variable names would be valid, it would increase the formula size by a multiplicative factor of $|x|$, which is not negligible. So we take a different approach, which instead increases the formula size just by an *additive* factor of $\mathcal{O}(|x|)$. $r_{\textit{Galil-Cook}}(N_i, x)$ will output

$$(F') \wedge (c_0 \vee c_1 \vee c_{b_1} \vee \cdots \vee c_{b_{|x|}}),$$

where in the above b_i denotes the value of the ith bit of x and each c_0 above means to write $\overline{x_{\langle N_i,\gamma+1\rangle}}$ and each c_1 above means to write $x_{\langle N_i,\gamma+1\rangle}$.

So, in our running example, if the value of x was 101, $r_{\textit{Galil-Cook}}(N_i, x)$ would be $(x_{\langle N_i,2\rangle} \wedge \overline{x_{\langle N_i,2\rangle}} \wedge \overline{x_{\langle N_i,1\rangle}}) \wedge (\overline{x_{\langle N_i,3\rangle}} \vee x_{\langle N_i,3\rangle} \vee x_{\langle N_i,3\rangle} \vee \overline{x_{\langle N_i,3\rangle}} \vee x_{\langle N_i,3\rangle})$.

It is not hard to see that the $r_{\textit{Galil-Cook}}$ we have constructed has all the promised properties. It has the correct running time, it validly reduces from whether N_i accepts x to the issue of whether $r_{\textit{Galil-Cook}}(N_i, x)$ is in SAT, it never outputs any literal involving the letter z, all its literals in fact are tagged by the N_i in use and so two applications created with regard to different machines (e.g., N_4 and N_7) are guaranteed to have variable-disjoint outputs, and it even is such that the desired s function exists. Our s function will take an input, parse it to get $(A) \wedge (B)$, will decode N_i from the variable names in A and will decode x from the fact that it is basically written out by the bits encoded by all but the first two disjuncts of B, and then will output the pair (N_i, x). If anything goes wrong in that process, as to unexpected syntax or so on, then what we were given is not an actual output of some run of $r_{\textit{Galil-Cook}}$ on a legal input, and we can output any junk pair that we like, without violating our promise as to the behavior of s. (Some inputs that are not valid outputs of $r_{\textit{Galil-Cook}}$ will not trigger the above "if anything goes wrong," since we did not here take the (N_i, x) we are about to output and compute $r_{\textit{Galil-Cook}}(N_i, x)$ to see whether the output of that matches our input. But we do not need to. The needed behavior here is that all valid inputs have the right output, and we have achieved that.)

References

[1] M. Bellare and S. Goldwasser. The complexity of decision versus search. *SIAM Journal on Computing*, 23(1):97–119, 1994.

[2] A. Borodin and A. Demers. Some comments on functional self-reducibility and the NP hierarchy. Technical Report TR 76-284, Department of Computer Science, Cornell University, Ithaca, NY, July 1976.

[3] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158. ACM Press, May 1971.

[4] B. Dilkina, C. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoors to satisfiability: dynamic sub-solvers and learning during search. *Annals of Mathematics and Artificial Intelligence*, 70(4):399–431, 2014.

[5] G. Erdélyi, L. Hemaspaandra, J. Rothe, and H. Spakowski. Generalized juntas and NP-hard sets. *Theoretical Computer Science*, 410(38–40):3995–4000, 2009.

[6] S. Fenner, L. Fortnow, A. Naik, and J. Rogers. Inverting onto functions. *Information and Computation*, 186(1):90–103, 2003.

[7] Z. Galil. On some direct encodings of nondeterministic Turing machines operating in polynomial time into P-complete problems. *SIGACT News*, 6(1):19–24, 1974.

[8] C. Gomes, A. Sabharwal, and B. Selman. Model counting. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, pages 633–654. IOS Press, 2009.

[9] J. Hartmanis and L. Hemachandra. Complexity classes without machines: On complete languages for UP. *Theoretical Computer Science*, 58(1–3):129–142, 1988.

[10] E. Hemaspaandra, L. Hemaspaandra, and C. Menton. Search versus decision for election manipulation problems. *ACM Transactions on Computation Theory*, 12(Article 3):1–42, 2020.

[11] L. Hemaspaandra. Computational social choice and computational complexity: BFFs? In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pages 7971–7977. AAAI Press, February 2018.

[12] L. Hemaspaandra and D. Narváez. The opacity of backbones. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 3900–3906. AAAI Press, February 2017.

[13] L. Hemaspaandra and D. Narváez. Existence versus exploitation: The opacity of backdoors and backbones. *Progress in Artificial Intelligence*, 2021. https://doi.org/10.1007/s13748-021-00234-6.

[14] L. Hemaspaandra, J. Rothe, and G. Wechsung. Easy sets and hard certificate schemes. *Acta Informatica*, 34(11):859–879, 1997.

[15] L. Hemaspaandra and R. Williams. An atypical survey of typical-case heuristic algorithms. *SIGACT News*, 43(4):71–89, 2012.

[16] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[17] R. Karp. Reducibilities among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, 1972.

[18] R. Karp, E. Upfal, and A. Wigderson. The complexity of parallel search. *Journal of Computer and System Sciences*, 36(1):225–253, 1988.

[19] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 1368–1373. AAAI Press, July 2005.

[20] L. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1975. March 1975 translation into English of Russian article originally published in 1973.

[21] A. Meyer and M. Paterson. With what frequency are apparently intractable problems difficult? Technical Report MIT/LCS/TM-126, Laboratory for Computer Science, MIT, Cambridge, MA, 1979.

[22] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.

[23] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. In *Informal Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, pages 96–103, May 2004.

[24] A. Procaccia and J. Rosenschein. Junta distributions and the average-case complexity of manipulating elections. *Journal of Artificial Intelligence Research*, 28:157–181, 2007.

[25] J. Rothe. Complexity of certificates, heuristics, and counting types, with applications to cryptography and circuit theory. Habilitation thesis, Friedrich-Schiller-Universität Jena, Institut für Informatik, Jena, Germany, June 1999.

[26] U. Schöning. Complete sets and closeness to complexity classes. *Mathematical Systems Theory*, 19(1):29–42, 1986.

[27] L. Valiant. The relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20–23, 1976.

[28] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[29] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1173–1178. Morgan Kaufmann, August 2003.