Fast Reductions from RAMs to Delegatable Succinct Constraint Satisfaction Problems*

Eli Ben-Sasson†	Alessandro Chiesa†	Daniel Genkin†	Eran Tromer‡
eli@cs.technion.ac.il	alexch@csail.mit.edu	danielg3@cs.technion.ac.il	tromer@cs.tau.ac.il
Technion	MIT	Technion	Tel Aviv University

August 18, 2012

Abstract

Succinct arguments for NP are proof systems that allow a weak verifier to retroactively check computation done by a powerful prover. Constructions of such protocols prove membership in languages consisting of *very large yet succinctly-represented constraint satisfaction problems* that, alas, are unnatural in the sense that the problems that arise in practice are not in such form.

For general computation tasks, the most natural representation is typically as random-access machine (RAM) algorithms, because such a representation can be obtained very efficiently by applying a compiler to code written in a high-level programming language. Thus, understanding the efficiency of reductions from RAM computations to other NP-complete problem representations for which succinct arguments (or proofs) are known is a prerequisite to a more complete understanding of the applicability of these arguments.

Existing succinct argument constructions rely either on circuit satisfiability or (in PCP-based constructions) on algebraic constraint satisfaction problems. In this paper, we present new and more efficient reductions from RAM (and parallel RAM) computations to both problems that (a) preserve succinctness (i.e., do not "unroll" the computation of a machine), (b) preserve zero-knowledge and proof-of-knowledge properties, and (c) enjoy fast and highly-parallelizable algorithms for transforming a witness to the RAM computation into a witness for the corresponding problem. These additional properties are typically not considered in "classical" complexity theory but are often required or very desirable in the application of succinct arguments.

Fulfilling all these efficiency requirements poses significant technical challenges, and we develop a set of tools (both unconditional and leveraging computational assumptions) for generically and efficiently structuring and arithmetizing RAM computations for use in succinct arguments. More generally, our results can be applied to proof systems for NP relying on the aforementioned problem representations; these include various zero-knowledge proof constructions.

Keywords: delegation of computation; succinct arguments; random-access machines; probabilistically checkable proofs; zero-knowledge proofs

*We thank Ohad Barta and Arnon Yogev for reviewing prior versions of this technical report. We also thank Nir Bitansky, Oded Golredich, Omer Paneth, Ryan Williams, and Zeyuan Allen Zhu for useful comments.

†The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258.

‡This work was supported by the Check Point Institute for Information Security, by the Israeli Ministry of Science and Technology, and by the Israeli Centers of Research Excellence I-CORE program (center 4/11).

Contents

1 Introduction

Cryptographic protocols offering integrity and confidentiality guarantees for general computation are much needed, e.g., for attaining secure cloud computing and for reducing trust in execution platforms. Work in cryptography has focused, for instance, on the problems of *computing on encrypted data* [Gen09, GH11b, GH11a, BGV12, GSS12b, GSS12a], *computing on authenticated data* [GW12], *delegation of computation* [GKR08, GGP10, CKV10, AIK10, CRR11, KRR12], *delegation of memory and streams* [CTY10, CKLR11], *non-interactive zero-knowledge proofs* [GOS06b, GOS06a, AF07, Gro09, Gro10b, Gro10a], *succinct arguments* [Kil92, Mic00, BG08, BCCT11, DFH12, GLR11, BC12, BCCT12], *succinct arguments with preprocessing* [Gro10b, Lip12, GGPR12], and *proof-carrying data* [CT10, CT12].

Each such cryptographic protocol "supports" a specific problem representation. For example, constructions of fully-homomorphic encryption schemes and delegation protocols work with evaluation of circuits; constructions of non-interactive zero-knowledge proofs often work with satisfiability of circuits; PCP-based succinct argument constructions work with constraint-satisfaction problems having a strong combinatorial or algebraic flavor (e.g., coloring problems over certain hypergraphs or whether there are low-degree polynomials with a certain set of roots).

However, problem representations such as the above are unnatural for the envisioned applications. While the circuit model may be convenient for highly-structured computational tasks (e.g., computing Fourier transforms or statistics on numerical data), it is not convenient in general; this is even more true for problem representations with strong combinatorial or algebraic structure. In the real world, the problems that we are interested in arise in the form of *algorithms* written in high-level programming languages such as C or Java.

In principle, this gap is not a problem: circuit evaluation is a complete problem for P, circuit satisfiability is a complete problem for NP, and so on. We can thus rely on generic polynomial-time reductions for transforming algorithms into the problem representations we need. However, the envisioned applications often deal with *very large* computations, and thus *the efficiency of such reductions is paramount* — reductions with only "polynomial-time" efficiency are not efficient enough.

The following question is thus of fundamental interest:

> **Informal Question:** How can we *generically and efficiently* transform algorithms written in high-level programming languages into the problem representations required by the underlying cryptographic protocols?

Ultimately, the hope is to develop a toolset for efficiently reducing the correctness of computation of algorithms to a variety of popular problem representations. Achieving this hope would bestow great flexibility to protocols that benefit from these reductions, because our ability to easily program algorithms would translate into the ability to easily *and efficiently* "program" less natural problem representations such as circuits or polynomials.

1.1 Random-Access Machine Computations

Existing compilers do a splendid job at reducing algorithms expressed in a high-level programming language to a sequence of basic instructions for a variety of computer architectures. Correct execution of such instructions, given the semantic specification of the architecture, is trivially and tightly reducible to the correct execution of a *random-access machine* (RAM) [CR72, AV77]. Thus, RAM computations are a natural starting point to choose for reductions. More precisely, we consider the following constraint satisfaction problem: the **bounded-halting problem** on a two-tape random-access machine M, denoted $\mathrm{BH}_{\mathsf{RAM}}(M)$, is the language of all triples $(\mathtt{x}, T, S)$ such that there is a

witness $\mathtt{w}$ for which $M(\mathtt{x}, \mathtt{w})$ accepts within T time steps[1] and using at most S memory cells. (Without loss of generality, $S \leq T$ and $|\mathtt{x}| \leq T$.) The language $\mathrm{BH}_{\mathsf{RAM}}$ is the one induced by all quadruples $(M, \mathtt{x}, T, S)$ such that $(\mathtt{x}, \mathtt{w})$ is in $\mathrm{BH}_{\mathsf{RAM}}(M)$. Thus, thanks to compiler technology, focusing on $\mathrm{BH}_{\mathsf{RAM}}$ as a starting point for reductions comes at essentially no loss in efficiency.

We model the RAMs we use as realistic reduced-instruction-set computers (RISCs) as opposed to the traditional simpler, but less realistic, accumulator-based architectures with arbitrary-length memory cells used in complexity theory. Doing so will give us a more accurate understanding of the cost of reducing compiled programs into other problem representations. Concretely, a RAM is specified by a tuple (w, k, δ), potentially depending on the input size $|\mathtt{x}|$, where w is the *register width* (which also dictates the width of every memory cell), k is the number of local (i.e., non-memory) registers, and δ is the *transition function* (which is a boolean circuit that, when given random-access to memory, maps the k registers to their new values); note that $|\delta| \geq kw$ since δ takes as input the kw bits making up the registers.[2] See Section 6.4 for formal definitions.

1.2 Our Focus: Succinct Arguments for NP

While there are many cryptographic protocols for which efficient reductions from $\mathrm{BH}_{\mathsf{RAM}}$ are desirable, we choose to focus on the problem of constructing efficient reductions from $\mathrm{BH}_{\mathsf{RAM}}$ for *proof systems for* NP. Also, we seek reductions that preserve zero-knowledge and proof-of-knowledge properties, since both properties are often those that make a proof system interesting to begin with.

Furthermore, we seek reductions that work even for proof systems for NP that are *succinct* [Kil92, Mic00, BG08], i.e., in which the verifier can check membership in a given language L in a way that is *much* faster than by directly verifying the validity of a witness. Such proof systems cannot be statistically sound (under plausible complexity-theoretic assumptions [BHZ87, GH98, GVW02, Wee05]) so that one must settle for *argument systems* [BCC88] (i.e., proof systems with only *computational* soundness). Constructing efficient reductions for succinct argument systems for NP is particularly important because such proof systems enable many useful protocols for delegating computation and, moreover, are most likely to deal with extremely large computations — and thus here efficient reductions are particularly needed.

Reductions and efficiency. Suppose that we have a succinct argument system (P, V) for an NP-complete language L and a reduction from $\mathrm{BH}_{\mathsf{RAM}}$ to L. For concreteness, suppose that L is circuit satisfiability (CSAT). Let us elaborate on the impact that the efficiency of the reduction has on the efficiency of the resulting argument system (P', V') for $\mathrm{BH}_{\mathsf{RAM}}$.

The prover P' uses the reduction to transform a membership question of the form "is $(\mathtt{x}, T, S)$ in $\mathrm{BH}_{\mathsf{RAM}}(M)$?" to one of the form "is the circuit C satisfiable?" and then invokes P. The verifier V' *also* uses the reduction in order to then invoke V.

To specify the question "is $(\mathtt{x}, T, S)$ in $\mathrm{BH}_{\mathsf{RAM}}(M)$?" we only need $O(|M| + |\mathtt{x}| + \log(T))$ bits, but the constraint satisfaction problem it describes (i.e., the correct execution of $M(\mathtt{x}, \mathtt{w})$, for some witness $\mathtt{w}$, for at most T steps) is much larger: it has size that is $\Omega(T)$. In a succinct argument, the verifier V' is weak, so he is only allowed to run in time that is $\mathsf{poly}\big(|M| + |\mathtt{x}| + \log(T)\big)$ (and up to a polynomial in the security parameter). Thus, the reduction cannot "unroll" the computation of M and perform operations on it (e.g., by explicitly laying out a circuit of size $\Omega(T)$ encoding constraints, say, for each step of computation), since that would require $\Omega(T)$ time and make V' too slow, but must instead implicitly convert the *large yet succinctly-represented* constraint satisfaction problem

[1] We do not employ the logarithmic-cost criterion [CR72] but instead count the number of machine state transitions.

[2] In "natural" RAMs, both $|\delta|$ and kw are $O(\log T)$. However, they will generally be significantly larger than $1 \cdot \log T$ depending on how "rich" the architecture of the machine is (e.g., how many different instructions are there, which arithmetic instructions are available, and so on). This is why we want to explicitly keep track of these parameters.

described by $(M, \mathtt{x}, T, S)$ into a succinct representation of a question of the form "is C in satisfiable?", i.e., the reduction should in fact be from $\mathrm{BH}_{\mathsf{RAM}}$ to *succinct* CSAT (sCSAT). Thus, succinctness of the reduction is a property that we must ensure in order for it to work for succinct arguments.

But even if succinct, a reduction could still be very inefficient because it could blow up the size of the constraint satisfaction problem, e.g., by outputting a (succinct description of a) circuit of size $O(T^2)$; this would cause P' to invoke P on a much larger instance than the original computation. Thus, we seek reductions where the blowup in the size of the problem is as small as possible.

Additionally, the prover is also responsible for converting a witness $\mathtt{w}$ for the question "is $(\mathtt{x}, T, S)$ in $\mathrm{BH}_{\mathsf{RAM}}(M)$?" into a corresponding assignment of the circuit. The efficiency of this mapping is *also* critical. And because witnesses may themselves be as large as T, we shall seek reductions where this mapping is not only very fast but also *highly parallelizable*.

To preserve proof of knowledge, we shall also want that recovering a witness for $(M, \mathtt{x}, T, S)$ from a valid assignment of C can be done in polynomial-time, though we are not particularly interested in making this "inverse map" very efficient.[3] This requirement makes the reduction a *Levin* reduction.

Comparison with traditional reductions. Naturally, investigating the above question will benefit from insights from complexity theory, where the study of reductions between different computational problems is a basic tool. Nonetheless, the reductions we seek are different from the traditional ones in two ways. First, as already explained, we want reductions between *uniform and succinct* models of computation. Second, reductions need not have information-theoretic guarantees, but only *computational* ones. That is, computational reductions (when appropriately defined) will generally suffice. This relaxation may allow for simpler or more efficient reductions. (In this paper we will in fact see such an example.)

2 Overview of Results

In principle we may have to worry about reducing $\mathrm{BH}_{\mathsf{RAM}}$ to numerous and quite different problems, in order to cover as many argument system constructions as possible. Fortunately, it turns out that almost all succinct argument constructions rely either on *circuit satisfiability* or *algebraic constraint satisfaction problems*. Thus, in this paper, we concentrate on the two concrete goals of reducing $\mathrm{BH}_{\mathsf{RAM}}$ to both of these problems. We next describe each goal in more detail and our results for it in the next two subsections (Section 2.1 and Section 2.2 respectively). In both situations, preserving succinctness and minimizing the blowup in size of the constraint satisfaction problem will present significant challenges. Some of these will be overcome by *leveraging nondeterminism* as a resource.

2.1 Programming circuits: from $\mathrm{BH}_{\mathsf{RAM}}$ to circuit satisfaction

Circuit satisfiability is the language of all satisfiable boolean circuits. Its *succinct* version replaces each circuit by a (potentially much smaller) circuit descriptor. Here we consider the uniform formulation of this language. Informally:

Definition 1 (sCSAT)**.** *For a family of circuit descriptors $\Phi = \{\phi_{T,S}\}_{T,S\in\mathbb{N}}$,* $\mathrm{sCSAT}(\Phi)$ *is the language of $(\mathtt{x}, T, S)$ such that there exists $\mathtt{w}$ for which $C(\mathtt{x}, \mathtt{w})$ accepts, where C is the circuit described by $\phi_{T,S}$.*[4] *The language* sCSAT *is the set of all $(\Phi, \mathtt{x}, T, S)$ such that $(\mathtt{x}, T, S) \in \mathrm{sCSAT}(\Phi)$.*

[3] Concretely, the efficiency of the witness recovery map affects the efficiency of security reductions relying on proof of knowledge and thus, ultimately, affects the size of the security parameter used to invoke the proof system.

[4] A circuit descriptor is an algorithm that, given as input a gate number of C, outputs information about the type of the gate, which gates are its inputs, which gates are its outputs, and so on. See [Pap94] for a precise definition.

Reducing from $\mathrm{BH}_{\mathsf{RAM}}$ to sCSAT benefits many zero-knowledge argument systems, including [GOS06b, GOS06a, AF07, Gro09, Gro10b, Gro10a, Lip12, GGPR12, BCCT12].[5] Our first goal is:

> Goal #1: Efficiently program circuits. Namely, reduce $\mathrm{BH}_{\mathsf{RAM}}$ to sCSAT with a succinct Levin reduction that is as tight as possible (both in terms of circuit size and efficiency of the witness map).

Prior work toward Goal #1. Leveraging the fact that (multi-tape) Turing machines can sort in quasilinear time [Sch78], Gurevich and Shelah [GS89, Theorem 2] showed that nondeterministic random-access machine computations can be simulated by nondeterministic (multi-tape) Turing machine computations with only polylogarithmic overhead. Combining this with the result of Pippenger and Fischer [PF79] that any T-step multi-tape Turing machine computation can be performed by an oblivious two-tape Turing machine in $\Theta(T \log T)$ steps, and the fact that the computation of an oblivious Turing machine (say with two tapes) in T' steps can be reduced to a circuit of size $\Theta(T')$, we deduce that there is a quasilinear-time reduction from random-access machines to circuits.

This reduction path already tells us that "random-access machines have small boolean circuits" and, from an asymptotic standpoint, already allows to efficiently program circuits for the purpose of, e.g., running zero-knowledge proofs [GOS06b, GOS06a, AF07, Gro09, Gro10b, Gro10a]. This reduction path, however, does not explicitly address the requirement of succinctness (though plausibly could be made to) or studies the efficiency of converting the witness $\mathtt{w}$ into a satisfying assignment; further, it hides large constants because it goes through (two) Turing machine reductions. A more efficient reduction was given by Robson [Rob91], whose proof yields (when cast in our notation) a $\Theta(T \cdot (\log S + wk) \cdot \log S)$ reduction from random-access machines to boolean formulas. However, Robson also did not address succinctness or study the efficiency of converting the witness.

Our results toward Goal #1. Our first theorem addresses Goal #1:

Theorem 1. *There are functions $(\mathsf{p}, \mathsf{w}_1, \mathsf{w}_2)$ such that, for every random-access machine M (with register width w, number of registers k, and transition function δ) and instance $(\mathtt{x}, T, S)$,*

Syntax: $\Phi = \{\phi_{T,S}\}_{T,S \in \mathbb{N}} = \mathsf{p}(M)$ *is a uniform family of circuit descriptors.*

Soundness: $(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ *if and only if* $(\mathtt{x}, T, S) \in \mathrm{sCSAT}(\Phi)$.

Witness reductions:

- *if* $\mathtt{w}$ *is a witness to* $(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ *then* $\mathsf{w}_1(M, \mathtt{x}, T, S, \mathtt{w})$ *is a witness to* $(\mathtt{x}, T, S) \in \mathrm{sCSAT}(\Phi)$*;*
- *if* $\mathtt{w}'$ *is a witness to* $(\mathtt{x}, T, S) \in \mathrm{sCSAT}(\Phi)$ *then* $\mathsf{w}_2(M, \mathtt{x}, T, S, \mathtt{w}')$ *is a witness to* $(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$.

Efficiency:

- $\mathsf{p}(M)$ *runs in linear time.*
- $\phi_{T,S}$ *can be generated in* $O(|\delta| + \log T)$ *time and describes a circuit of size* $O(|\delta| + (\log S + w) \cdot \log S) \cdot T$ *with* $O(kw + (\log S + w) \cdot \log S) \cdot T$ *variables.*
- $\mathsf{w}_1(M, \mathtt{x}, T, S, \mathtt{w})$ *runs in time* $O(kw + (\log S + w) \log S) \cdot T$ *and space* $O(kw + (\log S + w) \log S) \cdot S$*, or in parallel time* $O((\log S)^2)$ *when given the transcript of computation of* M *on* $(\mathtt{x}, \mathtt{w})$.
- $\mathsf{w}_2(M, \mathtt{x}, T, S, \mathtt{w}')$ *runs in linear time.*

Note that p is run by both the prover and (succinct) verifier, while the witness map w_1 is run only by the prover. (And the inverse witness map w_2 ensures proof of knowledge is preserved, and is usually only invoked in security reductions.)

[5] Not all these arguments are succinct, and thus not all benefit from the succinctness of the reduction; but, of course, all do benefit from the efficiency of the reduction (i.e., the blow up in circuit size and how fast an assignment for the circuit can be computed).

In terms of blowup in the size of the problem, our Theorem 1 provides a modest improvement over the one of Robson [Rob91]. More importantly, however, unlike Robson's result, our Theorem 1 provides a *succinct* reduction with a witness map w_1 that is very efficient (in both time *and* space) and is highly parallelizable. (Also see Section 2.3 for how computational assumptions can improve further the space complexity of w_1.) Our main technique for proving Theorem 1 is the use of routing networks together with nondeterminism, and our proof strategy builds on and simplifies Robson's.

For example, the succinctness property and the efficiency of the witness map w_1 from our Theorem 1 have been used in an essential way by Bitansky and Chiesa [BC12]. Concretely, they defined a *complexity-preserving* multiprover interactive proof (MIP) [BOGKW88] to be one where, to check that a random-access machine M non-deterministically accepts an input $\mathtt{x}$ within T time steps and using at most S memory cells, each MIP prover runs in time $(|M| + T) \cdot \mathsf{polylog}(T)$ and space $(|M| + |\mathtt{x}| + S) \cdot \mathsf{polylog}(T)$ and the MIP verifier runs in time $(|M| + |\mathtt{x}|) \cdot \mathsf{polylog}(T)$ — these complexities are essentially optimal. Bitansky and Chiesa showed how to use Theorem 1 to construct a complexity-preserving one-round MIP of knowledge, and used it to construct "complexity-preserving" succinct arguments from standard assumptions.

2.2 Programming polynomials: from $\mathbf{BH}_{\mathsf{RAM}}$ to algebraic constraint satisfaction

Probabilistically-checkable proofs (PCPs) [BFLS91] are perhaps the starkest examples of powerful proof systems that rely on unnatural problem representations: typically, they are constructed for constraint satisfaction problems having a strong combinatorial or algebraic flavor. All known PCP-based succinct argument constructions [Kil92, Mic00, BG08, DCL08, CT10, BCCT11, DFH12, GLR11] inherit these problem representations. Unlike studying reductions to (succinct) circuit satisfiability, though, studying reductions to problems used in PCP constructions is not as well-defined a problem, because these problems are different from each other. Which one should we pick as a target?

Because not all PCP constructions are equally efficient (e.g., some do not have a succinct verifier, some have long proofs, and so on), we focus on problems for which there are PCPs where the prover and verifier running times are sate of the art. Following a line of work on PCPs with short proofs [BFLS91, PS94, HS00, GS06, BSSVW03, BSGH+04, BSS08, BSGH+05, Mei12], Ben-Sasson et al. [BSCGT12] constructed PCPs with essentially-optimal prover and verifier running time (and both highly parallelizable); their PCPs are constructed using algebraic (rather than combinatorial) techniques, and involve testing proximity and checking properties of univariate (rather than multivariate) polynomials. Concretely, they construct PCPs for a *succinct algebraic constraint satisfaction problem* (sACSP) that is informally defined as follows (see Section 6.6 for details):

Definition 2 (sACSP)**.** *For a tuple of families* $\mathsf{par} = (\mathbf{F}, \mathbf{H}, \mathbf{N}, \mathbf{P}, \mathbf{I})$ *where*

- $\mathbf{F} = \{\mathbb{F}_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $\mathbb{F}_{T,S}$ *is a finite field of characteristic* 2*,*
- $\mathbf{H} = \{H_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $H_{T,S}$ *is a subspace of* $\mathbb{F}_{T,S}$*,*
- $\mathbf{N} = \{\vec{N}_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $\vec{N}_{T,S} = (N_{T,S,1}, \ldots, N_{T,S,|\vec{N}_{T,S}|})$ *is a vector of* neighbor polynomials*,*
- $\mathbf{P} = \{P_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $P_{T,S}$ *is a multivariate* constraint polynomial*,*
- $\mathbf{I} = \{\vec{I}_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $\vec{I}_{T,S} = (I_{T,S,1}, \ldots, I_{T,S,T})$ *is a vector of subspaces each contained in* $H_{T,S}$*,*
- $|\mathbb{F}_{T,S}| \geq 4 \cdot \deg\left(P_{T,S}(x, x^{|H_{T,S}|\cdot\deg(N_{T,S,1})}, \ldots, x^{|H_{T,S}|\cdot\deg(N_{T,S,|\vec{N}_T|})})\right)$*,*

the language $\text{sACSP}(\mathsf{par})$ *is defined as follows:*

$$\text{sACSP}(\mathsf{par}) = \left\{ (\mathtt{x}, T, S) : \begin{array}{c} \exists \text{ a univariate polynomial } A \text{ with } \deg(A) \leq |H_{T,S}| \text{ over } \mathbb{F}_{T,S} \text{ s.t.} \\ \text{(i) for all } \alpha \in H_{T,S},\ P_{T,S}(\alpha, A(N_{T,S,1}(\alpha)), \ldots, A(N_{T,S,|\vec{N}_T|}(\alpha))) = 0 \\ \text{(ii) } \mathtt{x} = A|_{I_{T,S,|\mathtt{x}|}} \end{array} \right\}.$$

The language sACSP *is then defined as the set of all* $(\mathsf{par}, \mathtt{x}, T, S)$ *such that* $(\mathtt{x}, T, S) \in \text{sACSP}(\mathsf{par})$.[6]

As mentioned, reducing BH_{RAM} to sACSP benefits all existing PCP-based succinct argument constructions. Thus, our second goal is;

> Goal #2: Efficiently program PCP-friendly polynomials. Namely, reduce BH_{RAM} to sACSP with a succinct Levin reduction that is as tight as possible (both in terms of circuit size and efficiency of the witness map).

Prior work toward Goal #2. Reductions from "non-algebraic" satisfaction problems to algebraic satisfaction problems are common in the PCP literature, where this process is known as *arithmetization*. Typically, the measure of how "large" is an algebraic constraint satisfaction problem is the field size (i.e., this metric is analogous to circuit size). Most arithmetizations in the PCP literature do *not* preserve succinctness and, moreover, are also quite expensive in terms of field size; for example, [Har04, Chapter 5] shows how to arithmetize circuits, but with a *cubic* blow up and with a reduction that does not preserve succinctness.

To the best of the authors' knowledge, the only arithmetizations that have taken succinctness into account are those of [BFLS91] and [BSGH+05]. Babai et al. [BFLS91] study a (succinct) reduction from *pointer machines* (concretely, those of Schönhage [Sch80], which in turn generalize Kolmogorov–Uspenskiĭ machines [Kol53, KU58]); however, even if pointer machines can simulate random-access machines with only polylogarithmic overhead [GS89], the reduction of Babai et al. is not of quasilinear efficiency and, more importantly, generates an algebraic constraint satisfaction problem for multivariate polynomials for which, unlike sACSP as defined above, PCPs as efficient are those of Ben-Sasson et al. [BSCGT12] are not known. Ben-Sasson et al. [BSGH+05] study a (succinct) reduction from *Turing machines* to a problem that is very similar to (and inspired the definition of) sACSP. Once again, because Turing machines can simulate random-access machines with only polylogarithmic overhead [GS89, Theorem 2], the result of [BSGH+05] could yield a reduction from BH_{RAM} to sACSP with quasilinear field size. However, this reduction path goes through two Turing machine reductions (the first to simulate the random-access machine and the second in their proof) and the field size depends *exponentially* on the (constant but large) number of states of the (second) Turing machine. A simple application of the techniques in [BSGH+05] to circumvent the use of Turing machines only yields a reduction with *quadratic* field size. Additionally, the efficiency of converting the witness $\mathtt{w}$ into a suitable polynomial was not studied in [BSGH+05].

Is there a "direct" reduction from BH_{RAM} to sACSP that, e.g., does not run into the inefficiencies of Turing machines and allows for a fast transformation of the witness?

Our results toward Goal #2. For a given random-access machine M, we introduce an additional complexity measure: we say that M has degree d if the (total) degree of its transition function δ, when viewed as an arithmetic circuit over $\mathbb{F}_2$, is at most d. Note that d is incomparable to both the state size kw and the size $|\delta|$ of the transition function as a boolean circuit; typically, $d = O(\log T)$.

Our second theorem addresses Goal #2:

Theorem 2. *There are functions* $(\mathsf{p}, \mathsf{w}_1, \mathsf{w}_2)$ *such that, for every random-access machine* M *(with register width* w*, number of registers* k*, and transition function* δ *with degree* d*) and instance* $(\mathtt{x}, T, S)$*,*

Syntax: $\mathsf{par} = (\mathbf{F}, \mathbf{H}, \mathbf{N}, \mathbf{P}, \mathbf{I}) = \mathsf{p}(M)$ *is a parameter choice for* sACSP*.*

Soundness: $(\mathtt{x}, T, S) \in \text{BH}_{\mathsf{RAM}}(M)$ *if and only if* $(\mathtt{x}, T, S) \in \text{sACSP}(\mathsf{par})$*.*

[6] To ensure succinctness, the field is represented via an irreducible polynomial of the appropriate degree over $\mathbb{F}_2$, subspaces via a basis and an offset, and polynomials via arithmetic circuits.

WITNESS REDUCTIONS:

- *if* $\mathtt{w}$ *is a witness to* $(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ *then* $\mathsf{w}_1(M, \mathtt{x}, T, S, \mathtt{w})$ *is one to* $(\mathtt{x}, T, S) \in \mathrm{sACSP}(\mathsf{par})$*;*
- *if* A *is a witness for* $(\mathtt{x}, T, S) \in \mathrm{sACSP}(\mathsf{par})$*" then* $\mathsf{w}_2(M, \mathtt{x}, T, S, A)$ *is one to "*$(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$.

EFFICIENCY:

- $\mathsf{p}(M)$ *runs in linear time.*
- $|\mathbb{F}_{T,S}| = O((\log T + kw) \cdot \log T \cdot d) \cdot T$.
- *a basis-and-offset representation for* $H_{T,S}$ *can be generated in* $O(\log |\mathbb{F}_{T,S}|)$ *field operations.*
- $\vec{N}_{T,S}$ *has* $O(\log S + kw)$ *affine functions, and each can be generated in* $O(\log |\mathbb{F}_{T,S}|)$ *field operations.*
- $P_{T,S}$ *has size* $O(|\delta| + \log |\mathbb{F}_{T,S}|)$ *and can be generated in* $O(|\delta| + (\log |\mathbb{F}_{T,S}|)^2 \log\log |\mathbb{F}_{T,S}|)$ *field operations.*
- *a basis and offset for each affine subspace in* $\vec{I}_{T,S}$ *can be generated in* $O(\log |\mathbb{F}_{T,S}|)$ *field operations.*
- $\mathsf{w}_1(M, \mathtt{x}, T, S, \mathtt{w})$ *runs in time* $O(|\mathbb{F}_{T,S}|(\log |\mathbb{F}_{T,S}|)^2)$*, or in parallel time* $O((\log |\mathbb{F}_{T,S}|)^2)$ *when given the transcript of computation of* M *on* $(\mathtt{x}, \mathtt{w})$.
- $\mathsf{w}_2(M, \mathtt{x}, T, S, A)$ *runs in time* $O(|\mathbb{F}_{T,S}|(\log |\mathbb{F}_{T,S}|)^2)$*, or in parallel time* $O((\log |\mathbb{F}_{T,S}|)^2)$.

(As mentioned, all polynomials are represented via arithmetic circuits.)

As before, p is run by both the prover and (succinct) verifier, while w_1 is run only by the prover. (And w_2 ensures proof of knowledge is preserved, and is typically invoked only in security reductions.)

Our Theorem 2 thus provides a succinct reduction from $\mathrm{BH}_{\mathsf{RAM}}$ to sACSP with small overhead and with a witness map w_1 that is very efficient in time (and is highly parallelizable). Unlike in Theorem 1, however, the witness map in Theorem 2 requires space $\Omega(|\mathbb{F}_{T,S}|) = \Omega(T)$ regardless of the space complexity of the machine M. While the space complexity of the witness map in Theorem 1 is dominated by the space needed to solve a routing problem of size $O(S)$, the space complexity of the witness reduction in Theorem 2 is dominated by the space needed to solve a univariate interpolation problem on a $\Omega(|\mathbb{F}_{T,S}|)$-size domain that, when solved with an FFT, requires $\Omega(|\mathbb{F}_{T,S}|)$ space. This inefficiency in terms of space complexity appears inherent.

The main techniques for proving Theorem 2 are the use of routing networks together with nondeterminism (as in Theorem 1), the use of novel arithmetization techniques (including further development of the computational properties of *linearized polynomials* [LN97, Section 2.5] used in [BSGH+05]), and additive FFTs [Mat08].

Our reduction from Theorem 2 has been used by Ben-Sasson et al. [BSCGT12] to construct a PCP system for random-access machine computations where, to check that a random-access machine M non-deterministically accepts an input $\mathtt{x}$ within T time steps, the PCP prover runs in sequential time $(|M| + T) \cdot \mathsf{polylog}(T)$ (or parallel time $O((\log T)^2)$ when given a transcript of computation) and the PCP verifier runs in sequential time $(|M| + |\mathtt{x}|) \cdot \mathsf{polylog}(T)$ (or parallel time $O((\log T)^2)$); Ben-Sasson et al. then used these to construct a variant of universal arguments [BG08] with similar efficiency. In both cases, both the prover and verifier are highly parallelizable but the space complexity of the prover remains $\Omega(T)$. See [BSCGT12] for more details and open problems in this direction.

2.3 Extensions

Parallel RAMs. Our results can be used in a simple and black-box way to also handle *parallel RAMs* (PRAMs) [Fic93] and obtain completely analogous results for these. Essentially, a PRAM can be reduced to a corresponding (sequential) RAM in the trivial fashion (by round-robin instruction-by-instruction simulation on a single processor, swapping the registers to memory at each switch). This simple reduction preserves parallelism in the following sense: the transcript of the sequential RAM can still be generated in parallel time essentially equal to that of the original PRAM. One can then invoke our results for sequential RAMs, which themselves have low parallel time complexity.

Non-determinism and computational assumptions. Theorem 1 and Theorem 2 provide unconditional guarantees. A natural question is whether by using computational assumptions one can obtain more efficient reductions. In this direction we identify several convenient transformations that leverage the existence of *universal one-way hash functions* (UOWHFs) [NY89], which can be obtained from the existence of one-way functions [Rom90].

Even if Theorem 1 ensures that the witness map can be computed in time and space that are close to those of the original machine M, it may still be not so efficient to run the witness map, e.g., when the space used by M is large only for a short period of time. While one can conceive of more sophisticated proofs to Theorem 1 that take into account additional information about the memory usage of M, a simpler approach can be based on UOWHFs. The lemma is the following:

Lemma 1. *Let $\mathcal{F} = \{f_\rho\}_\rho$ be a UOWHF family. There exist functions $(\mathsf{p}, \mathsf{w}_1, \mathsf{w}_2, \mathsf{t}, \mathsf{s})$ such that for every random-access machine M:*

SYNTAX: *For every seed $\rho \in \{0,1\}^*$, $\mathsf{p}(\rho, M)$ is a random-access machine.*

WITNESS REDUCTIONS:

- *For every seed $\rho \in \{0,1\}^*$, instance $(\mathtt{x}, T, S)$, and $\mathtt{w}$, if $\mathtt{w}$ is a witness to $(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ then $\mathsf{w}_1(\rho, M, \mathtt{x}, T, S, \mathtt{w})$ is a witness to $(\mathtt{x}, \mathsf{t}(\rho, M, T, S), \mathsf{s}(\rho, M, T, S)) \in \mathrm{BH}_{\mathsf{RAM}}(\mathsf{p}(\rho, M))$.*
- *With high probability over ρ, if a witness $\mathtt{w}'$ for $(\mathtt{x}, \mathsf{t}(\rho, M, T, S), \mathsf{s}(\rho, M, T, S)) \in \mathrm{BH}_{\mathsf{RAM}}(\mathsf{p}(\rho, M))$ can be efficiently generated, then a witness $\mathtt{w}$ for $(\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ can be efficiently extracted from it (so that, in particular, such a witness exists). Formally, for every polynomial-size circuit family $\{C_\kappa\}_{\kappa \in \mathbb{N}}$ and sufficiently large $\kappa \in \mathbb{N}$,*

$$\Pr_{\rho \leftarrow \{0,1\}^\kappa}\left[\begin{array}{c} \mathsf{w}_2(\rho, M, \mathtt{x}, T, S, \mathtt{w}') \text{ is not a witness to} \\ (\mathtt{x}, T, S) \in \mathrm{BH}_{\mathsf{RAM}}(M) \text{ and } \mathtt{w}' \text{ is witness to} \\ (\mathtt{x}, \mathsf{t}(\rho, M, T, S), \mathsf{s}(\rho, M, T, S)) \in \mathrm{BH}_{\mathsf{RAM}}(\mathsf{p}(\rho, M)) \end{array} \middle| (\mathtt{x}, 1^T, S, \mathtt{w}') \leftarrow C_\kappa(\rho) \right] < \mathsf{negl}(\kappa) \ .$$

EFFICIENCY:

- $\mathsf{p}(\rho, M)$ *runs in linear time.*
- $\mathsf{w}_1(\rho, M, \mathtt{x}, T, S, \mathtt{w})$ *runs in time $T \cdot \mathsf{poly}(|\rho| + \log S)$ and space that "mirrors" that used by $M(\mathtt{x}, \mathtt{w})$ (i.e., the number of non-zero memory cells in the i-th block of $\mathsf{poly}(|\rho| + \log S)$ steps of w_1 is equal to that at step i of $M(\mathtt{x}, \mathtt{w})$ times $\mathsf{poly}(|\rho| + \log S)$).*
- $\mathsf{w}_2(\rho, M, \mathtt{x}, T, S, \mathtt{w}')$ *runs in linear time.*
- $\mathsf{t}(\rho, M, T, S) = T \cdot \log S \cdot \mathsf{poly}(|\rho|)$.
- $\mathsf{s}(\rho, M, T, S) = \mathsf{poly}(|\rho|)$.

The important efficiency features of Lemma 1 are that the space complexity of $\mathsf{p}(\rho, M)$ is a fixed polynomial in $|\rho|$ and the witness map w_1 has space complexity that "mirrors" that of the computation of M on $(\mathtt{x}, \mathtt{w})$; in particular, even if M has large space complexity for a short period of time, w_1 will not have space complexity that is large for the entire computation (as was the case in Theorem 1).

Lemma 1 is in fact an immediate consequence of *online memory checking* [BEG+91] via Merkle trees [Mer89]: $\mathsf{p}(\rho, M)$ keeps memory in "untrusted storage" by dynamically maintaining a Merkle tree over it; essentially, every load instruction of M is replaced by a *secure load*, consisting of a sequence of instructions verifying an alleged value, and every store instruction of M is replaced by a corresponding *secure store*, which appropriately updates the root of the Merkle tree, stored in a register.

The new machine $\mathsf{p}(\rho, M)$ from Lemma 1 can then be plugged back into Theorem 1 to obtain a reduction to sCSAT where the space complexity of the witness map mirrors that of M.[7] As

[7] In fact, because the space complexity of $\mathsf{p}(\rho, M)$ is so low, one can at this point also use the trivial reduction from RAMs to circuits. In a sense, UOWHFs can thus be interpreted as an alternative to routing techniques as far as memory consistency is concerned.

another application, Lemma 1 was used by Bitansky et al. [BCCT12] as an essential component to the construction of *complexity-preserving* non-interactive succinct arguments of knowledge.[8]

In fact, UOWHFs have additional efficiency benefits. For example, if $\mathtt{x}$ is very long and it is inconvenient for, e.g., the verifier to keep $\mathtt{x}$ around, then the verifier can store $\mathtt{x}$ somewhere untrusted (say, give $\mathtt{x}$ to the prover) and only keep a short digest $\sigma_{\mathtt{x}}$ of of $\mathtt{x}$. Later on, the verifier can reduce any given machine M to another machine M' having $\sigma_{\mathtt{x}}$ hardcoded in it that first verifies that the first part of the witness contains $\mathtt{x}$ and then runs M on $\mathtt{x}$ and the second part of the witness. A similar observation holds when M is itself too large (e.g., when it encodes some nonuniform problem).

Note that all of the above applications of UOWHFs crucially leverage nondeterminism as a resource and require the proof system that is eventually used to have a proof-of-knowledge property. We find it an interesting open question to further investigate the power of computational reductions to simplify or obtain results that cannot be achieved with only unconditional reductions.

Reductions from other models. We have already noted that, because many problems arising in practice are algorithmic in nature and are written in high-level programming languages, it is convenient to use $\mathrm{BH}_{\mathsf{RAM}}$ as a starting point for reductions. For those occasions in which one is interested to use another model of computation as a starting point, one can always, e.g., write in C an interpreter for that model of computation and then compile it — after all, random-access machines are a powerful model of computation and can simulate other classical models of computation with little overhead [HPV77, Pau78, DT83, Rob86, KvLP88, Rob92].

3 Open Problems

Leveraging the expressive power of combinatorial and algebraic properties of graphs and polynomials, in this paper we have developed tools for efficiently embedding generic computation of programs into the satisfiability of circuits and into algebraic constraint satisfaction problems. We believe that a deeper understanding of the efficiency (and inefficiency) of reductions from $\mathrm{BH}_{\mathsf{RAM}}$ to these problems, as well as other problems used in other protocols, is an important and exciting research direction. We list here several intriguing questions that our work leaves unanswered.

Arithmetizing RAMs with less overhead. Our Theorem 1 gives a reduction from $\mathrm{BH}_{\mathsf{RAM}}$ to sCSAT with a circuit of size $O(|\delta|+(\log S+w)\cdot\log S)\cdot T$; in contrast, our Theorem 2 gives a reduction from $\mathrm{BH}_{\mathsf{RAM}}$ to sACSP with a field of size $O((\log T+kw)\cdot\log T\cdot d)\cdot T$. When $S=T$, the latter result is more expensive by "two $\log T$ factors": the degree d of the transition function δ, typically $\Omega(\log T)$, appears as a multiplicative factor in the field size; and each field element requires $\log|\mathbb{F}_{T,S}|\geq\log T$ bits to represent. Furthermore, when $S<T$, the $\log S$ factors in the former result are replaced by $\log T$ factors. While we conjecture that Theorem 1 is essentially tight, determining whether the additional $\log T$ factors to reduce from $\mathrm{BH}_{\mathsf{RAM}}$ to sACSP are inherent remains a challenging problem.

Programming deterministic models. The reductions presented in this paper are inherently *nondeterministic*: they leverage non-determinism as a resource in order to gain efficiency by "enlarging" the witness. Indeed, many techniques used in this paper can be distilled into the paradigm of carefully choosing what additional information we should "put in the witness" because it is much faster to verify its goodness rather than to generate such information from scratch. Thus, even when M does not read any witness, our, e.g., Theorem 1 reduces $\mathrm{BH}_{\mathsf{RAM}}(M)$ to a circuit *satisfaction* problem (that has a witness) and not a circuit *evaluation* one (that has no witness).

[8] Indeed, it seems that the recursive composition and bootstrapping techniques of Bitansky et al. [BCCT12] cannot be performed on the circuit obtained in the proof of Theorem 1. The result of [BCCT12] is an example of an application that we know how to achieve only via a reduction with only computational guarantees.

Our reliance on non-determinism for efficiency appears necessary: nondeterministic models of computation are quite robust under quasilinear-time reductions [PR79, Sch80, GS89, LL92, Jon93, NRS94] but reductions with such efficiency are not believed to exist for deterministic ones. Injecting non-determinism means that, unfortunately, our reductions cannot be used to benefit protocols that do not support nondeterminism; such protocols include all existing schemes for delegating polynomial-time functions, such as [GKR08, GGP10, CKV10, AIK10, CRR11, KRR12]. Many of these protocols require a circuit or Turing machine representation, and the best (deterministic) reductions from random-access machine computations to (deterministic) circuit or Turing machine computations have a cost of $\Omega(T^2)$; such a cost is large for long computations, which are precisely the ones that we are likely to delegate. Thus, the applicability of proof systems that do *not* support nondeterminism is limited in the sense that they cannot benefit from the great efficiency benefits that nondeterminism brings to reductions.

While for sufficiently structured problems one can carry out careful circuit design with great efficiency gains [CMT12], it is an interesting question whether reducing arbitrary random-access machines to "weaker" models of computation without the use of nondeterminism is bound to be expensive. Answering this question may be hard as it is related to circuit lower bounds.

Programming circuits for FHE. Existing constructions of fully-homomorphic encryption [Gen09] require a function to be represented as a circuit in order for it to be homomorphically evaluated on an encrypted input. The fact that our reductions leverage nondeterminism seems to imply that they cannot be used to efficiently program circuits for homomorphic evaluation either. Indeed, given only a random-access machine M and an encryption c of an input $\mathtt{x}$, it is not clear how to obtain an encryption of an assignment to the corresponding (succinctly-described) circuit generated by our reduction, without spending time that is at least quadratic in the running time of M on $\mathtt{x}$;[9] with such a slow transformation, one might as well have used the trivial quadratic (but deterministic) reduction from random-access machines to circuits. Thus, we do not know of any approach that is able to homomorphically evaluate a random-access machine in sub-quadratic time.[10]

While the aforementioned problem can be alleviated by exploiting the fact that the algorithms arising in practice are not "worst-case" [FvDD12], we believe it is an important open problem to either find a solution that can homomorphically evaluate an arbitrary random-access machine in, say, quasilinear time, or show that such a solution is unlikely to exist. Until then, the applicability of fully-homomorphic encryption may be limited to sufficiently structured problems (i.e., those that are naturally represented as circuits) or algorithms with special properties (e.g., use little memory).

Other starting points and other targets? While we have argued that $\mathrm{BH}_{\mathsf{RAM}}$ is a natural starting point due to the ease with which high-level programming languages can be reduced to it, there may be other powerful models of computation that are perhaps better suited to run certain kinds of algorithms. We have already mentioned one such example, parallel RAMs, that in fact we can already support using the results of this paper. What other models of computation (not easily supported by RAMs or parallel RAMs) provide useful starting points? Conversely, we have focused on reducing $\mathrm{BH}_{\mathsf{RAM}}$ to sCSAT and sACSP; which other underlying problem representations urgently need efficient reductions?

[9] Indeed, it is not clear how one would obtain, in less than quadratic time, an encryption of the transcript of the computation of M on $\mathtt{x}$, which is required in order to (homomorphically) solve the routing problem.

[10] A natural attempt would be to convert the given RAM into an *oblivious RAM* [GO96] in order to hide the information leaked by (logical) memory accesses, and then evaluate the transition function of the resulting "processor" under fully-homomorphic encryption in order to hide the registers and the oblivious RAM's secrets. However, in order to efficiently satisfy the processor's (oblivious) memory accesses, the evaluator needs to know the requested addresses, and these are produced in encrypted form. Thus, one needs an encryption scheme that lets the evaluator decrypt (correctly-computed) addresses and nothing else; no technique is known for this implausible-sounding goal.

4 Proof Strategy

We prove Theorem 2 in two steps and the first step of our proof will yield a proof to Theorem 1. Concretely, we identify a convenient "stepping-stone" problem (similar to those used in [VL88, BSS08, BSGH$^+$05]) that is a class of *succinct graph coloring problems* (sGCP). Roughly, an instance of sGCP consists of a coloring problem over a known (yet succinctly-represented) graph topology: each vertex in the graph has a corresponding coloring constraint (that can easily be deduced from the identity of the vertex) and this coloring constraint only involves the colors of the vertex itself and its neighbors. For comparison, sACSP can in fact be viewed as sGCP with the addition of certain algebraic constraints. Informally (see Section 6.5 for details):

Definition 3 (sGCP)**.** *For a tuple of families* $\mathsf{par} = (\mathbf{V}, \mathbf{\Gamma}, \mathbf{K}, \mathbf{W})$ *where*

- $\mathbf{V} = \{V_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $V_{T,S}$ *is a vertex set,*
- $\mathbf{\Gamma} = \{\vec{\Gamma}_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $\vec{\Gamma}_{T,S} = (\Gamma_{T,S,1}, \ldots, \Gamma_{T,S,|\vec{\Gamma}_{T,S}|})$ *is a vector of neighbor functions,*
- $\mathbf{K} = \{K_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $K_{T,S}$ *is a color constraint function,*
- $\mathbf{W} = \{\vec{W}_{T,S}\}_{T,S\in\mathbb{N}}$ *and each* $\vec{W}_{T,S} = (W_{T,S,1}, \ldots, W_{T,S,T})$ *is a vector of vertex sets each contained in* $V_{T,S}$*,*

the language sGCP(par) *is defined as follows:*

$$\text{sGCP}(\mathsf{par}) = \left\{ (\mathtt{x}, T, S) : \begin{array}{c} \exists \text{ a coloring } C \text{ of the vertices } V_{T,S} \text{ s.t.} \\ \text{(i) for all } v \in V_{T,S},\ K_{T,S}(v, C(\Gamma_{T,S,1}(v)), \ldots, C(\Gamma_{T,S,|\vec{\Gamma}_{T,S}|}(v))) = 0 \\ \text{(ii) } \mathtt{x} = C|_{W_{T,S,|\mathtt{x}|}} \end{array} \right\} .$$

The language sGCP *is then defined as the set of all* $(\mathsf{par}, \mathtt{x}, T, S)$ *such that* $(\mathtt{x}, T, S) \in \text{sGCP}(\mathsf{par})$.[11]

Our proof of Theorem 2 thus consists of the following two steps:

Step 1: from $\mathbf{BH_{RAM}}$ to sGCP by "localizing & structuring" constraints. In this step we translate the constraints that determine correctness of computation of a given random-access machine M to a graph coloring problem over a graph with special topology (i.e., to some choice $(\mathbf{V}, \mathbf{\Gamma}, \mathbf{K}, \mathbf{W})$ of parameters for sGCP). This is where we tame the global and unstructured nature of the RAM's memory access operations, which can reach arbitrarily across memory and need to be consistent with values stored arbitrarily long ago, by showing how to efficiently capture these by coloring constraints. The choice of topology and the fact that coloring constraints are local provides key properties that are used in the next step, during the process of arithmetization.

Step 2: from sGCP to sACSP by arithmetizing constraints. In this step we translate the choices of parameters for sGCP obtained in Step 1 to corresponding algebraic constraints in a finite field (i.e., to some choices of parameters $(\mathbf{F}, \mathbf{H}, \mathbf{N}, \mathbf{P}, \mathbf{I})$ for sACSP).

As mentioned, any sGCP problem easily induces a corresponding sCSAT problem: consider the boolean circuit that verifies (in parallel) all of the coloring constraints, and then takes the conjunction of all of these verification results; the succinctness of sGCP ensures that this boolean circuit can be represented succinctly. Thus, the reduction from sGCP to sCSAT is trivial and comes with no overhead. In the next two subsections (Section 4.1 and Section 4.2), we respectively discuss our proof strategy for Step 1, which will imply a proof to Theorem 1, and then Step 2, which together with Step 1 will imply Theorem 2.

[11] To ensure succinctness, vertex sets are represented via circuits that can output their elements and functions are represented via boolean circuits that compute them.

4.1 Step 1: From $\mathrm{BH}_{\mathsf{RAM}}$ To sGCP By Localizing & Structuring Constraints

The constraints that determine correctness of computation of a given random-access machine M can be divided into *code-consistency* constraints (i.e., did the machine correctly execute the correct instruction at every time step?) and *memory-consistency* constraints (i.e., did the machine load a value equal to the last stored value at the same address, every time it read from memory?). Our goal is to express these via a choice $(\mathbf{V}, \mathbf{\Gamma}, \mathbf{K}, \mathbf{W})$ of parameters for sGCP (see Definition 3) such that the underlying graph topology has certain special properties (to be discussed later). Let us fix a time bound T and let us temporarily fix the space bound S to be equal to T.

Given a time-ordered transcript of computation (i.e., a list of states of M) it is easy to locally check the code-consistency constraints: one can do so one at a time, considering pairs of adjacent states, by using the transition function δ of M. Furthermore, given a memory-ordered transcript of computation (i.e., a transcript of computation ordered by address, breaking ties using the timestamps) it is also easy to locally check the memory-consistency constraints: one can do so again one at a time, considering pairs of adjacent states, by ensuring that when a load follows a store at the same address the same value that was stored is loaded. The main difficulty, faced in every simulation result for machines with powerful memory models [BFLS91, GS89, Rob91], is how to efficiently ensure that the time-ordered transcript and the memory-ordered transcript are encoding the same computation; the one technique that has been used to do so in all these results is various forms of *nondeterministic sorting*. Essentially, one relies on some mechanism that can verify that one list is either a sorting or a permutation of the other list [Ofm65, Sch78, SH86].

In our case, we have several additional requirements: the mechanism must preserve succinctness, allow for a fast witness map, and have the appropriate structure to allow for an efficient arithmetization. (Let us temporarily ignore this last requirement of efficient arithmetization — we will address it in the next step in Section 4.2.) As the mechanism for nondeterministic sorting, we rely on a *routing network* with good algorithmic properties: *Beneš networks*. A Beneš network of size T is a graph with $O(\log T)$ columns, each with T vertices, that can route T elements from the first column to the last column, according to any given permutation, with no congestion. Here "route" simply means that intermediate columns receive a packet and either forward it to the "up neighbor" or "down neighbor" of the next column. Crucially, these routing decisions can be efficiently computed (and then treated as colors) in time linear in the size of the graph [Ben65, Wak68, OTW71, Lei92] or in parallel time $O((\log T)^2)$ [NS82].

We can then define $\mathbf{V}$ and $\mathbf{\Gamma}$ to encode a family of Beneš networks of the appropriate size and define $\mathbf{K}$ to be a family of coloring constraints that verifies code-consistency, memory-consistency, or routing constraints depending on whether the colors to be verified are on the first column, last column, or middle columns of the appropriate Beneš network. Additional care is required to appropriately define a family of vertex sets $\mathbf{W}$ that takes care of consistency of the transcript with the input `x`. (Indeed, not only do we need to ensure via constraints that some valid computation occurred, but we must also establish that this computation had something to do with the specific input at hand.)

When the space bound S is not equal to T, however, the above simplistic approach is not efficient enough because the Beneš network has $O(T \log T)$ vertices and we can only afford graphs of size $O(T \log S)$. To tackle this problem, Robson [Rob91] devises a system of checkpoints to ensure that he never has to sort a list much larger than S; his construction of a CNF formula, however, is further complicated by the fact that he is simulating a machine where the memory cells can store arbitrarily large values and thus must resort to an amortization scheme on top of his system of checkpoints.

We use a simpler approach. Concretely, we first transform M into a new machine M' that runs M with the twist that, every S steps of computation of M are preceded and followed by reading all S memory cells in order, *twice* in a row. (Note that the running time of M' is at most three

times that of M.) The memory accesses of M', due to their special pattern, can be routed with a permutation that exhibits strong *locality*: essentially, no state needs to be routed more than $2S$ far from its position in the time ordered transcript, since every memory address is accessed at least once in every $2S$ steps. This observation raises the hope of leveraging "shallower" routing networks that can route only sufficiently local permutations. Such routing networks do exist [Kan05]. In fact, in our case, the permutations that we need to route enjoy additional properties besides their locality and it suffices to consider the subnetwork consisting of the first $O(\log S)$ columns of the Beneš network — essentially, this subnetwork consists of T/S Beneš networks of size $O(S)$, and thus we still benefit from the algorithmic results for Beneš networks.

Overall, we reduce $\mathrm{BH}_{\mathsf{RAM}}$ to a sGCP over a certain family of "shallow" Beneš networks. Formalizing the above intuition involves many technical details, including some imposed by the requirement that all of the above must be able to be expressed succinctly and imposing other coloring constraints not discussed above. As stated above, doing so yields a proof to Theorem 1. We work out the details of this step for the case $S = T$ in Section 7.

4.2 Step 2: From sGCP To sACSP By Arithmetizing Constraints

Our second step is to efficiently arithmetize the sGCP problem on the graphs obtained in the first step. That is, we need to map the choice of parameters $(\mathbf{V}, \mathbf{\Gamma}, \mathbf{K}, \mathbf{W})$, where $(\mathbf{V}, \mathbf{\Gamma})$ encodes a family of shallow Beneš networks, to a choice of parameters $(\mathbf{F}, \mathbf{H}, \mathbf{N}, \mathbf{P}, \mathbf{I})$ for sACSP (see Definition 2) with as small a field size as possible while at the same time preserving succinctness. This process is quite delicate because one runs the risk of obtaining high-degree polynomials that are either not sparse or do not have small arithmetic circuits, or, worse, have a degree that is *too* high thereby forcing us to make the field size too large. Building on the initial work of Ben-Sasson et al. [BSGH$^+$05], we develop a set of algebraic tools that leverages the additive structure of subspaces of finite fields to tackle various problems that arise during arithmetization; many of these insights leverage the computational properties of *linearized polynomials* [LN97, Section 2.5].

At high level, we have two subtasks:

1. **Graph Arithmetization.** While the definition of sGCP does not impose any constraints on the vertex set family $\mathbf{V}$ and neighbor function family $\mathbf{\Gamma}$, the definition of sACSP imposes algebraic constraints on the vertex set family $\mathbf{H}$ and the neighbor function family $\mathbf{N}$. Namely, each vertex set $H_{T,S} \in \mathbf{H}$ must be a *subspace* of the field $\mathbb{F}_{T,S}$ and each vector of neighbor functions $\vec{N}_{T,S} \in \mathbf{N}$ must be a vector of (univariate) *polynomials* over $\mathbb{F}_{T,S}$. Thus, we need a way to efficiently arithmetize a shallow Beneš network: find $H_{T,S}$ and $\vec{N}_{T,S}$ such that the the graph induced by $(H_{T,S}, \vec{N}_{T,S})$ contains as a subgraph a shallow Beneš network of size T and $O(\log S)$ columns, where $H_{T,S}$ is as small as possible and $\vec{N}_{T,S}$ contains as few low-degree polynomials as possible.

   For this subtask we recall a routing network whose algebraic properties have been studied before by Polishchuk and Spielman [PS94], Ben-Sasson and Sudan [BSS08], and Ben-Sasson et al. [BSGH$^+$04, BSGH$^+$05]: *De Bruijn graphs*. When cast in our language, Ben-Sasson et al. [BSS08] showed how to arithmetize a De Bruijn graph of size T (having $O(T \log T)$ vertices) with an affine subspace $H_{T,S}$ of size $O(T \log T)$ and a constant number of degree-1 polynomials in $\vec{N}_{T,S}$.

   Unfortunately, we do not know if an analogous result holds for the shallow Beneš networks constructed in our first step. Nonetheless, these can still be embedded in De Bruijn graphs of size T, so that Beneš networks benefit from the aforementioned result of Ben-Sasson and Sudan as well. Crucially, the algorithmic properties of Beneš networks carry over to De Bruijn graphs.

2. **Constraints Arithmetization.** Once again, while the definition of sGCP does not impose any constraints on the color constraint function family $\mathbf{K}$, the definition of sACSP requires that the

constraint function family $\mathbf{P}$ be a family of (multivariate) polynomials where, essentially, for each $P_{T,S} \in \mathbf{P}$ all variables but the first have low degree. How to map $\mathbf{K}$ to $\mathbf{P}$? Doing so is the most delicate part of our reduction from sGCP to sACSP, so let us briefly discuss it next.

Storing coloring information. A witness to the sACSP problem is an *assignment polynomial* $A\colon \mathbb{F}_{T,S} \to \mathbb{F}_{T,S}$ of degree at most $|H_{T,S}|$. Such a polynomial A can be interpreted as assigning a "color" to each field element in $\mathbb{F}_{T,S}$, and thus A can be thought of as a "low-degree coloring function". A first difficulty is about how we should store coloring information A in a way that it can be efficiently retrieved by the constraint polynomial $P_{T,S} \in \mathbf{P}$. Concretely, suppose that we have a coloring function $C\colon V_{T,S} \to \{0,1\}^c$, assigning a color of c bits to each vertex in the De Bruijn graph, that is a witness to the sGCP problem (and, for simplicity, assume that $c \leq \log|\mathbb{F}_{T,S}|$). A natural approach would be to ensure that the bit string $C(v)$ is stored in the bit representation of $A(\Phi(v))$, where Φ is the function that embeds the vertex set $V_{T,S}$ of the De Bruijn graph into the field (guaranteed by the graph arithmetization discussed above). Such an economical "packing", however, has the problem that retrieving any given bit from $A(\Phi(v))$ requires a *projection polynomial* that has degree $\Omega(T)$; when the projection polynomial is composed with A, which also has degree $\Omega(T)$, we obtain a variable with degree $\Omega(T^2)$ — far too expensive.

To avoid this problem, we in fact use not just *one* De Bruijn graph but instead create c *copies* of the De Bruijn graph in $\mathbb{F}_{T,S}$. That is, we do not set $H_{T,S} = \Phi(V_{T,S})$ but instead consider an "extended vertex set" $\hat{V}_{T,S} := \bigcup_{j=1}^{c}(\Phi(V_{T,S}) + \theta_j)$ where $(\theta_j)_{j=1}^{c}$ is a vector of c (fixed) affine shifts of $\Phi(V_{T,S})$ chosen so that, $\forall j,k \in [c]$, $(\Phi(V_{T,S}) + \theta_j) \cap (\Phi(V_{T,S}) + \theta_k) = \emptyset$ and $\hat{V}_{T,S}$ is contained in a subspace of size $O(c \cdot |\Phi(V_{T,S})|)$. We can then store the i-th bit that C associates with the vertex $v \in V_{T,S}$ on the field element $\Phi(v) + \theta_i$, i.e., we ensure that $A(\Phi(v) + \theta_i) = C(v)_i$. Now no projection polynomials are needed to retrieve any given bit of a color because field elements store single bits. Overall, $H_{T,S}$ can (roughly) be set to be equal to $\hat{V}_{T,S}$ and $\vec{N}_{T,S}$ can (roughly) be set to equal to the $c \cdot O(1)$ degree-1 polynomials obtained by considering all the c shifts of the $O(1)$ degree-1 polynomials induced by the embedding Φ.

Having ensured that color information can be retrieved efficiently let us briefly discuss the construction of the constraint polynomial $P_{T,S} \in \mathbf{P}$, which will constrain the values of a candidate-witness low-degree polynomials, starting from the boolean circuit $K_{T,S} \in \mathbf{K}$.

Building the constraint polynomial. The boolean circuit $K_{T,S}$ takes as input the current vertex $v \in V_{T,S}$ together with the colors assigned to the vertices in its "neighborhood", namely, to $\Gamma_{T,S,1}(v), \ldots, \Gamma_{T,S,|\vec{\Gamma}_{T,S}|}(v)$. The constraint polynomial $P_{T,S}$ takes as input $\Phi(v)$ together with the colors that A assigns to the field elements in its neighborhood, namely, the colors that A assigns to $N_{T,S,1}(\Phi(v)), \ldots, N_{T,S,|\vec{N}_T|}(\Phi(v))$.

Intuitively, $P_{T,S}$ has to evaluate $K_{T,S}$ as part of its computation. Yet, $P_{T,S}$ only has access to $\Phi(v)$, but not to v, so that $P_{T,S}$ must compute the bits of v himself in order to pass them on to $K_{T,S}$. We show that, *given* the bit representation of $\Phi(v)$, computing v amounts to only evaluating a collection of (easy-to-compute) multilinear polynomials. Furtermore, in this case it is not a problem to use (high-degree) projection polynomials to deduce the bits of $\Phi(v)$ starting from $\Phi(v)$ because (i) these are not composed with A, and (ii) despite being high-degree they can be shown to be sparse (since they are *linearized polynomials* [LN97, Section 2.5]).

Another issue is that the embedding of the De Bruijn graph mentioned earlier is not perfect: while a De Bruijn graph has out-degree 2 the embedding Φ requires $O(1)$ edges inside the field. We do not know a priori which of the neighbors of $\Phi(v)$ are "true" (i.e., are images of neighbors of v in the De Bruijn graph), and thus $P_{T,S}$ needs to figure out which are the colors in the neighborhood of $\Phi(v)$ that should be forwarded to $K_{T,S}$ as inputs. We observe that, given the bit representation of

$\Phi(v)$ (which $P_{T,S}$ already computes for the reasons explained in the previous paragraph) it is possible to use constant-degree *multiplexer polynomials* to select which are the true neighbors of $\Phi(v)$.

There are additional issues, requiring careful arithmetization, needed for the final construction of $P_{T,S}$. For example, we need a small arithmetic circuit for an *alternator polynomial*: namely, a polynomial that, given two subspaces $V, W \subseteq \mathbb{F}_{T,S}$ with $V \subseteq W$, is equal to 1 on V but vanishes everywhere on $W - V$. Such a polynomial is useful because it allows to "turn on" a given constraint on the subspace V but to specifically turn off the same constraint everywhere on $V - W$ (where another constraint can be turned on). An alternator polynomial has degree $|W|$, which is very large; nonetheless, we show how to make a small arithmetic circuit for it, even if the polynomial itself is not sparse.

In summary. Once again, formalizing the above intuition involves many technical details, mostly having to do with keeping track of how expensive it is to convert, represent, and access the very large objects involved, as well as enforcing consistency with the input $\mathtt{x}$ (by appropriately defining the family of vector of subspaces $\mathbf{I}$). Overall, we obtain a choice of parameters $(\mathbf{F}, \mathbf{H}, \mathbf{N}, \mathbf{P}, \mathbf{I})$ for sACSP where $|H_{T,S}| = O(cT \log T)$, $\vec{N}_{T,S}$ contains $O(c)$ degree-1 polynomials, and $P_{T,S}$ has degree $|H_{T,S}|$ and total degree $\deg(K_{T,S})$ in the other variables. These settings allow us to choose a field $\mathbb{F}_{T,S}$ of size $O(cT \log T \deg(K_{T,S}))$. In Step 1, $c = \log T + kw$ and $\deg(K_{T,S}) = d$ (i.e., the degree of the transition function of M), as claimed in Theorem 2. The efficiency properties of converting the witness follow from routing algorithms on De Bruijn graphs (discussed in Step 1) and additive-FFT techniques [Mat08]. See Section 8 for more details.

5 Roadmap of Technical Sections

The rest of this paper is organized as follows. In Section 6 we introduce the basic definitions that we will use throughout this paper, including the notions of reduction (Section 6.1), routing networks (Section 6.2), finite fields (Section 6.3), random-access machines (Section 6.4), sGCP (Section 6.5), and sACSP (Section 4). Next, Section 7 provides details for Step 1 (which we already discussed informally in Section 4.1) and then Section 8 provides details for Step 2 (which we already discussed informally in Section 4.2); together, these two sections provide all the details for the proof of Theorem 2. Recall from the discussion in Section 4 that Theorem 1 follows immediately from Step 1. The appendices provide additional details that are used throughout the paper: on routing networks (Section A), circuit diagrams (Section B), and finite fields and efficient computation (Section C).

6 Definitions

We cover basic definitions used throughout the paper. Other definitions are given when appropriate.

Definition 6.1. *A function $c\colon \mathbb{N} \to \mathbb{N}$ is a* **proper (complexity) function** *if it can be computed in time $O(|x| + c(|x|))$ and space $O(c(|x|))$ by some Turing machine. (See [Pap94, Definition 7.1].)*

Functions computed by circuits. If f is a boolean function (a function over finite binary strings), we denote by $[f]^{\mathsf{B}}$ a boolean circuit (of AND, OR, NOT, and constant gates) for computing f. If f is a field function (a function over a vector of field elements), we denote by $[f]^{\mathsf{A}}$ an arithmetic circuit for computing f.

6.1 Levin Reductions

A *Levin reduction* is a Karp reduction together with efficient functions to "convert witnesses from both directions". Traditionally, it is defined as follows:

Definition 6.2. *Let L and L' be languages with respective polynomial-time-computable polynomially-balanced binary relations R_L and $R_{L'}$. A* **Levin reduction** *from L to L' is a triple of polynomial-time-computable functions $(f, \mathsf{w}_1, \mathsf{w}_2)$ over finite binary strings such that the following conditions hold for every $x \in \{0,1\}^*$:*

Soundness: *$x \in L$ if and only if $f(x) \in L'$.*

Witness Reductions:

- *for every $w \in \{0,1\}^*$, if $(x, w) \in R_L$ then $(f(x), \mathsf{w}_1(x, w)) \in R_{L'}$; and*
- *for every $w' \in \{0,1\}^*$, if $(f(x), w') \in R_{L'}$ then $(x, \mathsf{w}_2(x, w')) \in R_L$.*

We consider Levin reductions that are in certain respects more general and in other respects more specific compared to those induced by the traditional Definition 6.2. More precisely:

- We consider languages L whose instances have the form $(\mathtt{x}, 1^t, 1^s)$, where $\mathtt{x}$ is a binary string and $t, s \in \mathbb{N}$ (with $|\mathtt{x}|$ and 2^s bot at most 2^t). Loosely, the interpretation is that $(\mathtt{x}, 1^t, 1^s)$ is in L if there is some 2^t-size witness $\mathtt{w}$ for which $\big((\mathtt{x}, 1^t, 1^s), \mathtt{w}\big) \in R_L$. We put *no restriction* on the magnitude of t relative $|\mathtt{x}|$ (other than the aforementioned $|\mathtt{x}| \leq 2^t$), and thus the relations of the languages that we consider will *not necessarily be polynomially-balanced*. Intuitively, that is because we want to capture long computations on small inputs as well.
- We only consider Levin reductions via the "identity mapping"; that is, we will only consider reductions where the "instance map" of the reduction (i.e., f) is the identity. This restriction will be mostly out of simplicity, rather than out of necessity (as the reductions will already contain enough complexity).

Furthermore, we consider Levin reductions between *classes* of languages: a reduction from a class of languages $\mathcal{L}$ to another class of languages $\mathcal{L}'$ includes an efficient "parameter" function p that takes as input (a description of) a language in $\mathcal{L}$ and generates (the description of) a language in $\mathcal{L}'$. Our motivation is to highlight the "succinct and uniform" nature of our reductions, across a number of languages with similar properties.

We concretize the foregoing discussion as a template definition for the notion of Levin reduction that we use, and we will instantiate it later for the specific cases that we consider. (See Definition 7.1 and Definition 8.1.)

Definition 6.3. *Let $\mathcal{L}$ and $\mathcal{L}'$ be classes of languages with polynomial-time-computable binary relations. A* **Levin reduction** *from $\mathcal{L}$ to $\mathcal{L}'$ is a triple of polynomial-time-computable boolean functions $(\mathsf{p}, \mathsf{w}_1, \mathsf{w}_2)$ such that the following conditions hold for every $L \in \mathcal{L}$ and $(\mathtt{x}, 1^t, 1^s) \in \{0,1\}^*$:*

SYNTAX: $\mathsf{p}(L) \in \mathcal{L}'$.

SOUNDNESS: $(\mathtt{x}, 1^t, 1^s) \in L$ *if and only if* $(\mathtt{x}, 1^t, 1^s) \in \mathsf{p}(L)$.

WITNESS REDUCTIONS:

- *if* $\mathtt{w} \in \{0,1\}^*$ *is a witness to* $(\mathtt{x}, 1^t, 1^s) \in L$ *then* $\mathsf{w}_1(L, \mathtt{x}, 1^{2^t}, 1^s, \mathtt{w})$ *is a witness to* $(\mathtt{x}, 1^t, 1^s) \in \mathsf{p}(L)$.
- *if* $\mathtt{w}' \in \{0,1\}^*$ *is a witness to* $(\mathtt{x}, 1^t, 1^s) \in \mathsf{p}(L)$ *then* $\mathsf{w}_2(L, \mathtt{x}, 1^{2^t}, 1^s, \mathtt{w}')$ *is a witness to* $(\mathtt{x}, 1^t, 1^s) \in L$.

As mentioned in Section 1.2, when using a Levin reduction in order to invoke a proof system, both the prover and verifier need to run p; the prover needs to run w_1 to transform the witness, while w_2 ensures that proof of knowledge is preserved. We will be mostly concerned in ensuring that p does not "blow up" the constraint satisfaction problem induced by L and that w_1 is as fast to compute as possible (and, ideally, also highly parallelizable).

6.2 Routing Networks

We make use of special classes of *routing networks*. For us, a routing network is a graph with a distinguished set of *sources* and a distinguished set of *sinks*, both of the same size, such that for every permutation of sources to sinks there exists a set of node-disjoint paths that connects each source to the corresponding sink — a property known as *rearrangeability*.[12]

Rearrangeability of Beneš networks. Beneš networks [Ben65] are a class of routing networks. Roughly, a κ-dimensional Beneš network is a directed graph with $O(\kappa)$ columns each containing 2^κ vertices where each vertex is connected to two vertices in the following column. A Beneš network is rearrangeable; the sources are the vertices in the first column and the sinks are the vertices in the last column. Given a permutation π over 2^κ elements, a set of routing paths can be computed in sequential time $O(\kappa \cdot 2^\kappa)$ [Wak68, OTW71, Lei92] and in parallel time $O(\kappa^2)$ [NS82]. When the given permutation π is promised to be ℓ-*local* (i.e., $|\pi(i) - i| \leq \ell$ for all i) then one can modify the Beneš network in a way that it has only $O(\log \ell)$ columns (instead of $O(\kappa)$) and is still be able to route the permutation [Kan05]. A special case of ℓ-local permutations are permutations that "factor" into several permutations over ℓ elements at a time; such permutations can of course be routed via several (small) Beneš networks.

Beneš networks are closely connected to (extended) De Bruijn graphs, discussed next.

Rearrangeability of extended De Bruijn graphs. We employ (a more general version of) the definition of *extended* (also, *wrapped*) De Bruijn graph given in [BSS08, Definition 5.4]:[13]

Definition 6.4. *Let κ and L be two positive integers. The (κ, L)* **extended De Brujn graph**, *denoted $\mathrm{DB}(\kappa, L)$, is a directed 2-regular graph with L layers numbered $0, \ldots, L-1$, each containing 2^κ vertices identified by κ-bit strings. A vertex in layer $i \in \{0, \ldots, L-1\}$ with identifier $w \in \{0,1\}^\kappa$*

[12] Routing networks are also known as *connection networks*. These are a special case of *generalized connection networks*, which can implement every mapping from sources to sinks (and not just permutations). Even though generalized connection networks are not much harder to construct than connection networks [Ofm65, Pip73, Pip77, Tho78], we shall indeed only need connection networks.

[13] Equivalent definitions have appeared in the same context. For example, the graph defined in [BSGH$^+$05, Definition 4.1] is homomorphic via the mapping $(w, i) \mapsto (\mathrm{sr}^i(w^\mathrm{r}), i)$ to ours; also, the graph defined in [Spi95, Definition 4.3.2] is homomorphic via the mapping $(w, i) \mapsto (w^\mathrm{r}, i)$ to ours.

has two neighbors in layer $(i+1 \bmod L)$ *with identifier* $\mathrm{sr}(w)$ *and* $\mathrm{sr}(w) \oplus e_1$*. In other words, the edge set is induced by the following two neighbor functions:*

$$\Gamma_1\big((i,w)\big) = \Big((i+1 \bmod L), \mathrm{sr}(w)\Big) \ , \ \textit{and}$$
$$\Gamma_2\big((i,w)\big) = \Big((i+1 \bmod L), \mathrm{sr}(w) \oplus e_1\Big) \ ,$$

where sr *denotes the cyclic "shift right" bit operation.*

As stated in Section 4.1, we are interested in extended De Bruijn graphs due to their convenient algebraic properties [PS94, BSS08, BSGH+04, BSGH+05]. Essentially, an extended De Bruijn graph can be efficiently embedded into an *affine graph* (i.e., a graph whose neighbor functions are degree-1 polynomials) over an affine subspace of a finite field.

Extended De Bruijn graphs are closely related to Beneš networks in that algorithmic results for the latter can typically be translated into algorithmic results for the former. For example, this holds for the property of rearrangeability: namely, extended De Bruijn graphs, when "sufficiently wide", are rearrangeable, that is, they can route any given permutation of the leftmost vertices.

Claim 6.5. *Let* κ *be a positive integer and* $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ *a permutation. There exists a set* S_π *of* 2^κ *node-disjoint paths such that each vertex* $(0,w)$ *in* $\mathrm{DB}(\kappa, 4\kappa-1)$ *is connected to* $(0,\pi(w))$*. Moreover,* S_π *can be found in time and space* $O(\kappa \cdot 2^\kappa)$ *or parallel time* $O(\kappa^2)$*.*

Details about the claim can be found in Appendix A; for now we simply explain how the claim follows from the material there.[14]

Proof. Because $\mathrm{DB}(\kappa, 4\kappa-1)$ "contains three and a half" κ-dimensional De Bruijn graphs connected in tandem (except that the first and last column are identified, or "wrapped"), we can simply follow the optimizations discussed after Claim A.11 to obtain the desired set S_π. □

We do not know if a result for ℓ-local permutations analogous to the one of Kannan [Kan05] (or even to the one for permutations that factor into smaller ones) holds for extended De Bruijn graphs.

Why not oblivious routing techniques? The aforementioned routing results are of the *non-oblivious* kind, which means that the set of routing paths is computed upfront with knowledge of the permutation π. There are (deterministic or randomized) routing approaches that are *oblivious*, but these are less efficient (from an algorithmic or algebraic standpoint); thus, because in our setting we can tolerate non-oblivious solutions (which, additionally, are highly parallelizable), we are content to rely on the aforementioned non-oblivious routing results.

For example, a *sorting network* is a routing network where packets are routed via local comparisons. While there exist sorting networks whose size is asymptotically as good as that of Beneš networks, such constructions hide large constants and, more importantly, it is not clear how they can be arithmetized efficiently. (For instance, Ajtai et al. [AKS83] construct a sorting network of size $O(\kappa \cdot 2^\kappa)$ by relying on expander graphs, which we do not know how to arithmetize as efficiently as De Bruijn graphs.)

[14] While the routing properties of extended De Bruijn graphs are folklore, we have not been able to find explicit algorithms in the literature for routing; we deduce an explicit routing algorithm over *three and a half* De Bruijn graphs connected in tandem.

6.3 Finite Fields

We represent elements of a finite field as polynomials modulo an irreducible polynomial of the appropriate degree. Specifically, to represent elements of $\mathbb{F}_q$, where $q = p^f$ and p is the characteristic of $\mathbb{F}_q$: we consider any *irreducible* polynomial I over $\mathbb{F}_p$ of degree f; I has a root x in $\mathbb{F}_q$ and thus $\mathbb{F}_q = \mathbb{F}_p(\mathsf{x})$, so that every element of $\mathbb{F}_q$ can be uniquely expressed as a polynomial $\alpha(\mathsf{x})$ in x over $\mathbb{F}_p$ of degree less than f. See [LN97, Section 2.5] for more details. In particular, this representation will allow us to perform field operations in time that is polylogarithmic in the field size and space that is logarithmic in the field size. See Appendix C for more details and additional notation.

6.4 Random-Access Machines

Continuing the discussion from Section 1.1, in this section we formally introduce the notion of random-access machines that we use in this paper. We begin with an informal discussion and and then proceed to formal definitions.

6.4.1 Informal Discussion

At the highest level, we consider random-access machines modeling a simple *reduced-instruction-set computer* with a simple *load/store architecture*. Informally, a random-access machine M works as follows:

- *Code:* M has hardcoded in it a vector of n *instructions* $\mathbb{C} = (I_0, \ldots, I_{n-1})$, called M's *code*.
- *Registers:* M maintains a vector $r = (r_0, \ldots, r_{k-1})$ of k "local" *registers*, where each r_i is a string of w bits.
- *Memory:* M has random-access to *memory*, a vector $\mathbb{M}$ of 2^w memory cells of w bits each.
- *Tapes:* M has two read-only tapes, called tape A and B; we think of tape A as the *input tape* and B as the *witness tape*; without loss of generality, both tapes are unidirectional; M also has a single write-only unidirectional *output tape*.
- *Program counter:* M maintains a pointer to the next instruction to be executed; this pointer is known as the *program counter*, denoted by pc and initially set to 0^w, and is, like other registers, a w-bit string. (In particular, the number of instructions n is at most 2^w.)

At every time step the instruction I_{pc} is executed; the instruction may modify registers, the program counter, or memory as dictated by the *transition function* of M (which knows M's code). If the program counter is not modified by the instruction, it is set to $\mathsf{pc} + 1$, so that in the next time step instruction $I_{\mathsf{pc}+1}$ is executed. An instruction may also choose to read the next w bits from either tape. For technical reasons, we also assume that every time the machine executes a load instruction it executes a special "post-load" (pload) instruction in the next step with the same operands. A special instruction denotes the end of the computation, and the machine outputs either accept or reject. See Figure 1 for a schematic diagram.

A important matter is what kind of instructions a machine M is allowed to execute. We consider the following instruction set:

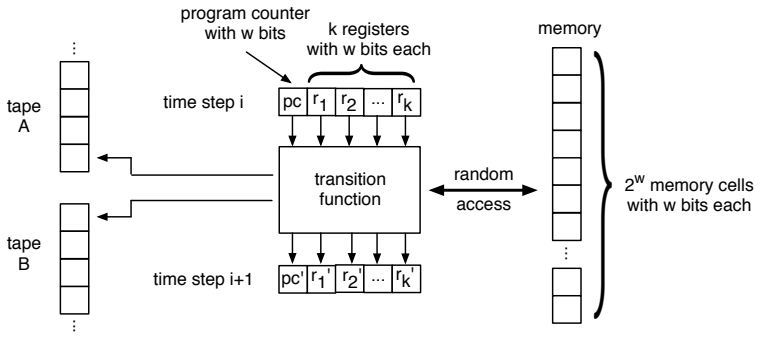


Figure 1: A diagram depicting the kind of random-access machines we use.

instruction	**arg. 1**	**arg. 2**	**arg. 3**	**semantics**
nop				no operation (does nothing)
read_A	i			read the next w-bit string (or #) from tape A and put it into r_i
read_B	i			read the next w-bit string (or #) from tape B and put it into r_i
load	i	j		$r_j \leftarrow \mathbb{M}[r_i]$
pload	i	j		no operation (does nothing)
store	j	x		$\mathbb{M}[r_j] \leftarrow x$
iseof	i	j		if $r_j = \#$, then $r_i \leftarrow 1$, else $r_i \leftarrow 0$
out	s			print s and halt
$a \in \mathbb{A}$	i	x	y	$r_i \leftarrow a(x, y)$ (see below)

where $\mathsf{s} \in \{\mathsf{accept}, \mathsf{reject}\}$ and each of x and y is a register, the symbol "pc", or a w-bit constant. As mentioned above, we assume that reading from a tape is in one direction only: every time the machine reads the next w-bit string from (say) tape A, the pointer on tape A moves to the next w-bit string and never moves back; when the end of the input on the tape has been reached, reading from the tape returns the symbol #; similarly for tape B. Finally, $\mathbb{A}$ is any set of (Turing-complete) binary *arithmetic* operations. For example, the set $\mathbb{A}$ may consist of the following operations:

instruction	**arg. 1**	**arg. 2**	**arg. 3**	**semantics**
add	i	x	y	$r_i \leftarrow x + y$
sub	i	x	y	$r_i \leftarrow x - y$
mul	i	x	y	$r_i \leftarrow x * y$
and	i	x	y	$r_i \leftarrow x \,\&\, y$
or	i	x	y	$r_i \leftarrow x \mid y$
not	i	x		$r_i \leftarrow !\, x$
cmp	i	x	y	if $x < y$ then $r_i \leftarrow 1$ else if $x = y$ then $r_i \leftarrow 0$ else if $x > y$ then $r_i \leftarrow -1$
jl	i	c		if $r_i \leq 0$ then $\mathsf{pc} \leftarrow c$
jle	i	c		if $r_i < 0$ then $\mathsf{pc} \leftarrow c$
je	i	c		if $r_i = 0$ then $\mathsf{pc} \leftarrow c$
jne	i	c		if $r_i \neq 0$ then $\mathsf{pc} \leftarrow c$
jmp	c			$\mathsf{pc} \leftarrow c$

where in the above table $c \in \{0, \ldots, n-1\}$ is an instruction number. Of course, the aforementioned example is a very rich one and much fewer instructions suffice for Turing completeness; for instance, with $\mathbb{A}$ having only the add, mul, and je, we can run any computation.

Finally, we are thinking of the *register width* w as implicitly being some function of the input size (and thus so is the size of each memory cell, the number of addressable memory cells, etc.) but we do not make this dependence explicit for the sake of simplicity of notation.

Measuring complexity. We do not employ the logarithmic-cost criterion but instead measure time in terms of *time steps* (i.e., state transitions of the machine). As for space complexity, we count the *maximum number of memory cells used during the computation.* Note that restricting memory only to addresses between 0 and $2^w - 1$ comes without loss of generality because code that uses very large addresses can be turned into code that uses smaller addresses. (See, e.g., [Rob91].)

Other architectures. As mentioned above, our notion of a random-access machine can be viewed as a simple reduced-instruction-set computer with a simple load/store architecture. Of course, there are other "architectures" that do not exactly fit our notion.

For example, in a *one instruction set computer* [Jon88, GL03], the single instruction typically has multiple simultaneous memory accesses, while our definition only envisages a single memory access per load or store instruction. (Such minimalist architectures are still Turing complete, and programs are written using *self-modifying code.*) The techniques that we develop for handling our notion of random-access machine can be extended to also handle architectures, e.g., having multiple memory accesses per instruction.

Alternatively, (in a "morally equivalent way") we can perform a "software reduction" of other architectures to our notion of random access machine by, e.g., striping multiple memory accesses across successive instructions. For example, if the single instruction of the computer is "subtract and branch if less than or equal to zero" (subleq), there are three simultaneous memory accesses, two loads and a write. More precisely, the instruction $\mathsf{subleq}\ a\ b\ c$ means $\mathbb{M}[b] := \mathbb{M}[b] - \mathbb{M}[a]$ and if $\mathbb{M}[b] \leq 0$ go to c. We can then map the instruction $\mathsf{subleq}\ a\ b\ c$ to the three "microcode" instructions $\mathsf{subleq}_1\ a$, $\mathsf{subleq}_2\ b$, and $\mathsf{subleq}_3\ c$ which all together perform what is needed by $\mathsf{subleq}\ a\ b\ c$.

6.4.2 Formal Definitions

We now turn to formalizing the above discussion. The goal of this section is to define the *bounded-halting problem for random-access machines.* We begin with the definition of a random-access machine itself.

Definition 6.6. *A* **random-access machine** *(RAM) is a tuple $M = \langle w, k, \mathbb{A}, \mathbb{C} \rangle$, where:*

- $w \in \mathbb{N}$ *is the* register size*;*
- $k \in \mathbb{N}$ *is the* number of registers*;*
- $\mathbb{A}$*, a set of functions $a \colon \{0,1\}^w \times \{0,1\}^w \to \{0,1\}^w$, is the* arithmetic unit*; and*
- $\mathbb{C} = (I_0, \ldots, I_{n-1})$*, where $n \in \{1, \ldots, 2^w\}$ and each I_i is an instruction, is the* code*.*

The "local" state of a random-access machine is given by a configuration, which contains the current program counter and the values of all the registers.

Definition 6.7. *Let $M = \langle w, k, \mathbb{A}, \mathbb{C} \rangle$ be a random-access machine. A* **configuration** *of M is a tuple*

$$S = \left[\mathsf{pc}, r_0, \ldots, r_{k-1}\right] \ ,$$

where I_{pc} is the next instruction to be executed in the code $\mathbb{C}$ and $r_0, \ldots, r_{k-1}$ are the current w-bit values of the k registers. (Note that the size of a configuration is $|S| = (1+k)w$ bits.)

Certain configurations are special in that they mark the beginning of a computation, or its ending (and, if so, whether the computation was accepting or not).

Definition 6.8. *Let $M = \langle w, k, \mathbb{A}, \mathbb{C} \rangle$ be a random-access machine and $S = [\mathsf{pc}, r_0, \ldots, r_{k-1}]$ a configuration of M.*

- *We say that S is* **initial** *for M if $\mathsf{pc} = 0^w$, and $r_0 = \cdots = r_{k-1} = 0^w$.*
  (I.e., the program counter points to the first instruction and all the k w-bit registers are zero.)
- *We say that S is* **final** *for M if $I_{\mathsf{pc}} = (\mathsf{out}, \mathsf{s})$ for some s.*
  (I.e., the program counter points to an out *instruction.)*
- *We say that a final configuration S is* **accepting** *for M if $I_{\mathsf{pc}} = (\mathsf{out}, \mathsf{accept})$.*
  (I.e., the program counter points to an out *instruction whose argument is* accept.*)*
- *We say that a final configuration S is* **rejecting** *for M if $I_{\mathsf{pc}} = (\mathsf{out}, \mathsf{reject})$.*
  (I.e., the program counter points to an out *instruction whose argument is* reject.*)*

A sequence of configurations starting in an initial configuration and ending in an accepting configuration is accepting.

Definition 6.9. *Let M be a random-access machine, and $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations of M. We say that $\vec{S}$ is* **accepting** *for M if S_0 is initial for M and S_{T-1} is accepting for M.*

If a random-access machine has inputs on either tape, we must specify what it means for a sequence of configurations to be consistent with the two inputs. Intuitively, this simply means that, when the machine reads from the tape, the actual values of the inputs appear in the configurations.

Definition 6.10. *Let M be a random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations of M, and $\mathtt{x}$ and $\mathtt{w}$ two input strings. We say that $\vec{S}$ is* **consistent** *with $(\mathtt{x}, \mathtt{w})$ if:*

(i) $\rho_0 \cdots \rho_{\ell-1}$ is a prefix of $\mathtt{x}$, where $\rho_0 \cdots \rho_{\ell-1}$ is the (time-ordered) concatenation of all the w-bit strings ρ_i (not equal to #) read from tape A according to $\vec{S}$, and

(ii) $\sigma_0 \cdots \sigma_{\ell-1}$ is a prefix of $\mathtt{w}$, where $\sigma_0 \cdots \sigma_{\ell-1}$ is the (time-ordered) concatenation of all the w-bit strings σ_i (not equal to #) read from tape B according to $\vec{S}$.

Given a configuration S and a memory vector $\mathbb{M}$, the configuration obtained by executing one step of computation is "implied" by S.

Definition 6.11. *Let M be a random-access machine, and let S and S' be two configurations of M.*

- *For any memory vector $\mathbb{M}$ for M, we say that S* **implies** *S' via $\mathbb{M}$, denoted $S \overset{\mathbb{M}}{\rightsquigarrow} S'$, if, by executing the instruction I_{pc}, M goes from configuration S and memory $\mathbb{M}$ to configuration S' (and possibly some other memory vector $\mathbb{M}'$), for* some *settings of the two tapes.*
- *We say that S* **implies** *S' (without specifying a memory vector $\mathbb{M}$), denoted $S \rightsquigarrow S'$, if there exists a memory vector $\mathbb{M}$ for M such that $S \overset{\mathbb{M}}{\rightsquigarrow} S'$.*

Having defined implication from a given configuration to another one, we are now ready to define the transition function of a random-access machine. Intuitively, the transition function verifies "code consistency".

Definition 6.12. *Let M be a random-access machine. The* **transition function** *of M, denoted δ_M, is the boolean function over pairs of configurations of M such that:*

$$\delta_M(S, S') \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } S \rightsquigarrow S' \\ 0 & \text{if } S \text{ is final for } M \text{ and } S' \text{ is initial for } M \\ 1 & \text{otherwise} \end{cases} .$$

(In Appendix B we provide and discuss a circuit for the transition function.)

We now specify under what conditions sequences of configurations are considered valid.

Definition 6.13. *Let M be a random-access machine, and $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations of M. We say that $\vec{S}$ is* **valid** *with M if $\delta_M(S_i, S_{i+1 \bmod T}) = 0$ for $i = 0, \ldots, T-1$ and, in addition, there exists a sequence $\mathbb{M}_0, \ldots, \mathbb{M}_{T-2}$ of memory vectors for M such that*

- $\mathbb{M}_0[j] = 0^w$ *for all* $j \in \{0,1\}^w$ *and* $S_0 \overset{\mathbb{M}_0}{\rightsquigarrow} S_1 \overset{\mathbb{M}_1}{\rightsquigarrow} \cdots \overset{\mathbb{M}_{T-2}}{\rightsquigarrow} S_{T-1}$, *and*
- *for all* $0 \leq i \leq T-1$ *it holds that* $\mathbb{M}_i = \mathbb{M}_{i+1}$ *if* S_{i+1} *does not contain a* store *instruction and* $\mathbb{M}_i = \mathbb{M}_{i+1}$ *except the value of the memory cell accessed by the* store *instruction whose value is required to be the value of the second operand of the* store *instruction.*

We are finally ready to introduce bounded-halting problems for random-access machines. Intuitively, for a given machine M, $\mathrm{BH}_{\mathsf{RAM}}(M)$ is the set of all $(\mathtt{x}, 1^t, 1^s)$ such that, for some $\mathtt{w}$, $(\mathtt{x}, \mathtt{w})$ is accepted by some computation of M within time at most 2^t and space at most 2^s. In fact, for convenience, we shall require the length of the computation to be a power of 2; of course, this is without loss of generality (and incurs in at most a multiplicative factor of 2 in the length of the computation) as we can always pad the computation with enough nop's.

Definition 6.14. *Let M be a random-access machine. The language $\mathrm{BH}_{\mathsf{RAM}}(M)$ consists of instances $(\mathtt{x}, 1^t, 1^s)$, where $\mathtt{x}$ is a binary string and t, s are positive integers (with $|\mathtt{x}| \leq 2^t$), such that there exists a binary string $\mathtt{w}$ of length at most 2^t for which the following condition holds: there exists an accepting sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ of M that (i) is valid for M, (ii) is consistent with $(\mathtt{x}, \mathtt{w})$, and (iii) accesses at most 2^s distinct addresses. Furthermore, we denote by $\mathrm{BH}_{\mathsf{RAM}}$ the language of all quadruples $(M, \mathtt{x}, 1^t, 1^s)$ such that $(\mathtt{x}, 1^t, 1^s) \in \mathrm{BH}_{\mathsf{RAM}}(M)$.*

6.5 sGCP: A Generic Succinct Graph Coloring Problem

We now formalize the class of problems sGCP, which we informally introduced in Section 3. At the highest level, sGCP is a class of *graph coloring problems*; each vertex in the graph has a induces a "local constraint" over colors in the *neighborhood* of the vertex, and membership of an instance in the language is determined by whether there exists an assignment of colors for every vertex in the graph, satisfying every local constraint, that "contains" the given instance. It is for this class of problems that we construct reductions from $\mathrm{BH}_{\mathsf{RAM}}$, and it is from this class we reduce to sACSP (described in Section 6.6). Formally:

Definition 6.15 (Succinct Graph Coloring)**.** Consider the following parameters:

1. two proper functions associated with the family $\mathbf{V}$ in Parameter 2:
   (a) a cardinality function $\mathsf{c}_{\mathbf{V}} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and
   (b) a time function $\mathsf{t}_{\mathbf{V}} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

2. a family $\mathbf{V} = \{V_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $V_{t,s}$ is a vertex set of cardinality $2^{\mathsf{c}_\mathbf{V}(t,s)}$, and
   (b) there exists a $\mathsf{t}_\mathbf{V}$-time algorithm $\textsc{Find}\mathbf{V}$ such that $v_i = \textsc{Find}\mathbf{V}(1^t, 1^s, i)$ is the i-th vertex of $V_{t,s}$ for every $t, s \in \mathbb{N}$ and $i \in \{1, \ldots, 2^{\mathsf{c}_\mathbf{V}(t,s)}\}$;

3. two proper functions associated with the family $\mathbf{\Gamma}$ in Parameter 4
   (a) a regularity function $\alpha_\mathbf{\Gamma} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and
   (b) a time function $\mathsf{t}_\mathbf{\Gamma} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

4. a family $\mathbf{\Gamma} = \{\vec{\Gamma}_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $\vec{\Gamma}_{t,s} = (\Gamma_{t,s,i} \colon V_{t,s} \to V_{t,s})_{i=1}^{\alpha_\mathbf{\Gamma}(t,s)}$ is a vector of $\alpha_\mathbf{\Gamma}(t,s)$ neighbor functions, and
   (b) there exists a $\mathsf{t}_\mathbf{\Gamma}$-time algorithm $\textsc{Find}\mathbf{\Gamma}$ such that $\textsc{Find}\mathbf{\Gamma}(1^t, 1^s, i, \cdot)$ computes $\Gamma_{t,s,i}(\cdot)$ for every $t, s \in \mathbb{N}$ and $i \in \{1, \ldots, \alpha_\mathbf{\Gamma}(t,s)\}$;

5. a proper function associated with the family $\mathbf{C}$ in Parameter 6:
   (a) a cardinality function $\mathsf{c}_\mathbf{C} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

6. a family $\mathbf{C} = \{C_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $C_{t,s} = \{0,1\}^{\mathsf{c}_\mathbf{C}(t,s)}$ is a finite color set;

7. two proper functions associated with the family $\mathbf{K}$ in Parameter 8:
   (a) a size function $\mathsf{s}_\mathbf{K} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and
   (b) a time function $\mathsf{t}_\mathbf{K} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

8. a family $\mathbf{K} = \{K_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $K_{t,s} \colon V_{t,s} \times C_{t,s}^{\alpha_\mathbf{\Gamma}(t)} \to \{0,1\}$ is a coloring constraint, and
   (b) there exists a $\mathsf{t}_\mathbf{K}$-time algorithm $\textsc{Find}\mathbf{K}$ such that, for every $t, s \in \mathbb{N}$, $[K_t]^\mathsf{B} = \textsc{Find}\mathbf{K}(1^t, 1^s)$ is a $\mathsf{s}_\mathbf{K}(t,s)$-size circuit computing $K_{t,s}$;

9. one proper function associated with the family $\mathbf{W}$ in Parameter 10:
   (a) a time function $\mathsf{t}_\mathbf{W} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

10. a family $\mathbf{W} = \{W_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
    (a) $W_{t,s}$ is a subset of $V_{t,s}$, and
    (b) there exists a $\mathsf{t}_\mathbf{W}$-time algorithm $\textsc{Find}\mathbf{W}$ such that $v_i := \textsc{Find}\mathbf{W}(1^t, 1^s, i)$ is i-th vertex in $W_{t,s}$ (under some canonical ordering of $W_{t,s}$) for every $t, s \in \mathbb{N}$ and $i \in \{1, \ldots, |W_{t,s}|\}$;

11. one proper function associated with the family $\mathbf{F}$ in Parameter 12:
    (a) a time function $\mathsf{t}_\mathbf{F} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

12. a family $\mathbf{F} = \{F_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
    (a) $F_{t,s} \colon \{0,1\}^* \to \{0,1\}$ is a function, and
    (b) there exists a $\mathsf{t}_\mathbf{F}$-time algorithm $\textsc{Comp}\mathbf{F}$ such that $\textsc{Comp}\mathbf{F}(1^t, 1^s, \cdot)$ computes $F_{t,s}(\cdot)$ for every $t, s \in \mathbb{N}$.

The language sGCP, with respect to a choice

$$\mathsf{par}_{\text{sGCP}} = \Big((\mathsf{c}_{\mathbf{V}}, \mathsf{t}_{\mathbf{V}}, \mathbf{V}), (\alpha_{\mathbf{\Gamma}}, \mathsf{t}_{\mathbf{\Gamma}}, \mathbf{\Gamma}), (\mathsf{c}_{\mathbf{C}}, \mathbf{C}), (\mathsf{s}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathbf{K}), (\mathsf{t}_{\mathbf{W}}, \mathbf{W}), (\mathsf{t}_{\mathbf{F}}, \mathbf{F})\Big)$$

of the above parameters, consists of instances $(\mathtt{x}, 1^t, 1^s)$, where $\mathtt{x}$ is a binary input string and t, s are positive integers (with $|\mathtt{x}|, 2^s \leq 2^t$), such that there exists a coloring $C \colon V_{t,s} \to C_{t,s}$ for which the following two conditions hold:

(i) *Satisfiability of constraints.* For every vertex $v \in V_{t,s}$,

$$K_{t,s}\Big(v, \big(C \circ \Gamma_{t,s,1}\big)(v), \ldots, \big(C \circ \Gamma_{t,s,\alpha_{\mathbf{\Gamma}}(t)}\big)(v)\Big) = 0 \ . \tag{1}$$

If so, we say that the coloring C *satisfies the coloring constraints* induced by $K_{t,s}$.

(ii) *Consistency with the input.* For every $i \in \{1, \ldots, |\mathtt{x}|\}$, letting v_i be the i-th vertex in $W_{t,s}$,

$$F_{t,s}\Big(\mathtt{x}, \big(C(v_i)\big)_{i=1}^{|\mathtt{x}|}\Big) = 0 \ .$$

If so, we say that the coloring C *is consistent* with the input $\mathtt{x}$.

The above definition for succinct graph coloring problems is quite a mouthful; this is because the language itself encodes requirements ensuring that "large objects" (such as the graph topology) can be computed using very few resources by using succinct and functional representations of such objects (for example, efficiently constructible boolean circuits). Ultimately, these requirements will directly affect similar requirements in the corresponding sACSP.

Remark 6.16. How is a choice of parameters $\mathsf{par}_{\text{sGCP}}$ for sGCP (concisely) specified? All the "complexity functions" from Definition 6.15 (i.e., those specifying running times, sizes of arithmetic circuits, and so on) were chosen to be proper (see Definition 6.1), and thus each has an efficient algorithm that computes it (which, for simplicity, we denote with the same name as the function); moreover, every infinite family comes with an algorithm that computes the information about the family we are interested in. Thus, a choice of parameters $\mathsf{par}_{\text{sGCP}}$ can be specified as follows:

$$\mathsf{par}_{\text{sGCP}} = \begin{pmatrix} (\mathsf{c}_{\mathbf{V}}, \mathsf{t}_{\mathbf{V}}, \textsc{Find}\mathbf{V}), \\ (\alpha_{\mathbf{\Gamma}}, \mathsf{t}_{\mathbf{\Gamma}}, \textsc{Find}\mathbf{\Gamma}), \\ \mathsf{c}_{\mathbf{C}}, \\ (\mathsf{s}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \textsc{Find}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \textsc{Find}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \textsc{Comp}\mathbf{F}) \end{pmatrix} .$$

Intuitive discussion. At high level, a choice of parameters $\mathsf{par}_{\text{sGCP}}$ for sGCP identifies a collection of infinite families of objects (one object for each $t, s \in \mathbb{N}$). When a specific instance $(\mathtt{x}, 1^t, 1^s)$ is considered for membership in $\text{sGCP}(\mathsf{par}_{\text{sGCP}})$, the (t, s)-th element from each of these families in the collection is used to determine membership of the instance; candidate witnesses are colorings for the graph $G_{t,s}$ induced by the vertex set $V_{t,s}$ and vector of neighbor functions $\vec{\Gamma}_{t,s}$. Despite the long definition, these objects interact in natural ways, so we now go over the parameters from Definition 6.15 in less formal terms, explaining some of the intuition behind the design of the definition.

- **Parameter 1 and Parameter 2.** The parameter $\mathbf{V} = \{V_{t,s}\}_{t,s \in \mathbb{N}}$ is a family of vertex sets, the (t, s)-th one with $2^{\mathsf{c}_{\mathbf{V}}(t,s)}$ vertices, that are to be colored by a coloring function (allegedly encoding information about a computation). Each vertex set $V_{t,s}$ is succinctly represented in the sense that there is an algorithm $\textsc{Find}\mathbf{V}$ that computes the "name" of each vertex in $V_{t,s}$.

- **Parameter 3 and Parameter 4.** The parameter $\mathbf{\Gamma} = \{\vec{\Gamma}_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of neighbor functions, where each $\vec{\Gamma}_{t,s}$ contains $\alpha_{\mathbf{\Gamma}}(t,s)$ neighbor functions that provide a topology to the vertex set $V_{t,s}$. There is an algorithm FIND$\mathbf{\Gamma}$ that can compute every neighbor function.

- **Parameter 5 and Parameter 6.** The parameter $\mathbf{C} = \{C_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of finite color sets, the (t,s)-th one of cardinality $2^{\mathsf{c}_{\mathbf{C}}(t,s)}$. We assume without loss of generality that the colors in C_t are simply binary strings of length $\mathsf{c}_{\mathbf{C}}(t)$.

- **Parameter 7 and Parameter 8.** The parameter $\mathbf{K} = \{K_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of constraints, where the (t,s)-th constraint is a *single* condition that is required to hold for every vertex in the graph; the constraint takes as input the vertex the colors of the vertices in the neighborhood of the vertex. Intuitively, depending on the vertex, the constraint will enforce different properties among the colors of the neighbors. With a careful use of vertex information, even if we only consider a single constraint enforced across the whole graph, we can express and enforce different color constraints on the graph. To make sure that each of the constraints can be found (and then evaluated) efficiently, the definition prescribes the existence of an algorithm FIND$\mathbf{K}$ that for each $t, s \in \mathbb{N}$ is able to output a small boolean circuit that computes $K_{t,s}$.

At high level, the parameters described until now could, say, encode a variety of constraints to ensure "correct computation" (say, of Turing machine, or a random-access machine), but so far did not encode anything about whether this correct computation had anything to do with the input $\mathtt{x}$ under consideration. The remaining parameters relate the computation to the input at hand.

- **Parameter 9 and Parameter 10.** The parameter $\mathbf{W} = \{W_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of subsets, where the (t,s)-th subset $W_{t,s}$ is a subset of the vertex set $V_{t,s}$. Each subset $W_{t,s}$ identifies on which part of $V_{t,s}$ a (candidate-witness) coloring must contain information related to the input $\mathtt{x}$, which can be verified by the next parameter. As usual, we need an algorithm FIND$\mathbf{W}$ to ensure that accessing elements of $W_{t,s}$ can be done appropriately efficiently.

- **Parameter 11 and Parameter 12.** The parameter $\mathbf{F} = \{F_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of "instance-consistency" functions. Intuitively, given the color of enough elements in $W_{t,s}$ assigned by a (candidate-witness) coloring for the graph, the function $F_{t,s}$, which can be computed by the algorithm COMP$\mathbf{F}$, verifies that the received colors are consistent with $\mathtt{x}$.

In summary, the "satisfiability of constraints" condition of the definition should be interpreted as saying that some computation (the particulars of which are captured by the specific choices of parameters) was performed correctly, while the other condition, "consistency with the input", should be interpreted as saying that this computation was performed on the instance at hand.

Remark 6.17. Our Definition 6.15 is inspired by related definitions that have appeared in [BSS08] and [BSGH$^+$05]. We briefly discuss here how our definition compares to the previous ones.

The definition of Ben-Sasson and Sudan [BSS08, Definition 5.6] also considers coloring problems over graphs; however, their definition is specialized to De Bruijn graphs and (as is clear for its compactness!) does *not* require succinctness because their paper does not attempt to construct PCP verifiers that are succinct. (Also note that, in the non-succinct case, the "consistency with the instance" requirement disappears.) The later paper of Ben-Sasson et al. [BSGH$^+$05], which studies PCPs with succinct verifier, does indeed give a definition (cd. [BSGH$^+$05, Definition 4.3]) for a succinct graph coloring problem (similar to the previous one of Ben-Sasson and Sudan [BSS08, Definition 5.6]), but again is specific to De Bruijn graphs and the parameters obtained when reducing from bounded halting problems on Turing machines. Additionally, both of the previous definitions

do not account for the additional "space bound" index s. The definition of Ben-Sasson et al. is related to the generalized coloring problem of Venkatesan and Levin [VL88].

We consider a generalization of [BSGH$^+$05, Definition 4.3] that leaves unspecified degrees of freedom that are crucial for "supporting" our efficient reductions.

6.6 sACSP: A Generic Succinct Algebraic Constraint Satisfaction Problem

We discuss and define sACSP, already (informally) introduced in Definition 2 in Section 2.2. As described by Ben-Sasson et al. [BSCGT12], sACSP is a class of *succinct algebraic constraint satisfaction problems*, each specified by a list of parameters, for univariate polynomials over finite field extensions of GF(2); we can in fact interpret sACSP as a class of succinct graph-coloring problems (cf. sGCP in Section 6.5) that must respect certain algebraic constraints.

Each particular sACSP problem simultaneously allows for the construction of PCPs with prover and verifier running times that are essentially optimal (as shown in [BSCGT12]) and is flexible enough to support efficient reductions from random-access machine computations, as we show in this paper. In fact, to give ourselves more flexibility, we extend the definition presented in [BSCGT12] so to index members of a family not only via a "time bound" T but also a "space bound" S.

Informally, a choice par of parameters of sACSP consists of the following:

- A field size function $f\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, inducing a finite field family $\{\mathbb{F}_{T,S}\}_{T,S\in\mathbb{N}} := \{\mathrm{GF}(2^{f(T,S)})\}_{T,S\in\mathbb{N}}$.
- A family $\{H_{T,S}\}_{T,S\in\mathbb{N}}$, where each $H_{T,S}$ is an affine subspace of $\mathbb{F}_{T,S}$.
- A family $\{\vec{N}_{T,S}\}_{T,S\in\mathbb{N}}$, where each $\vec{N}_{T,S}$ is a vector of neighbor polynomials.
- A family $\{P_{T,S}\}_{T,S\in\mathbb{N}}$, where each $P_{T,S}$ is a constraint polynomial.
- A family $\{\vec{I}_{T,S}\}_{T,S\in\mathbb{N}}$, where each $\vec{I}_{T,S}$ is a vector of affine subspaces each contained in $H_{T,S}$.

A certain algebraic relation constrains which parameter choices are valid. A reduction to sACSP will concretely instantiate a choice of the above parameters.

A triple $(\mathtt{x}, T, S)$ is a member of the language sACSP(par) if it fulfills the following: there exists a low-degree assignment polynomial $A\colon \mathbb{F}_{T,S} \to \mathbb{F}_{T,S}$ that "colors" elements of the field $\mathbb{F}_{T,S}$ such that (1) for every element α of the subspace $H_{T,S}$, the constraint polynomial $P_{T,S}$, when given as input the colors in the "$\vec{N}_{T,S}$-induced neighborhood" of α, is satisfied; and (2) the colors of elements in the $(\log|\mathtt{x}|)$-th affine subspace in $\vec{I}_{T,S}$ are consistent with $\mathtt{x}$.

Crucially, for each of the families in par, the (T,S)-th object must be able to be "understood" in time, say, $\mathsf{polylog}(T)$ (e.g., generating an element in $H_{T,S}$, generating an arithmetic circuit for each polynomial in $\vec{N}_{T,S}$, etc.); this is the requirement of *succinctness* of the problem. Additional properties that are essential for us to construct efficient reductions include, for example, the fact that $H_{T,S}$ is an affine subspace (so that one may leverage the computational properties of *linearized polynomials* [LN97, Section 2.5]).

From the perspective of this paper, what is important is that we have a lot of freedom in choosing what is (and how to use) the subspace $H_{T,S}$, what exactly are the affine functions in $\vec{N}_{T,S}$ (so that we may induce a variety of topologies for a corresponding "affine graph" over the field), what are the properties that the constraint polynomial $P_{T,S}$ verifies, and so on. Quantitatively, our goal is to ensure that f grows as slow as possible.

In the formal discussions, it will in fact be more convenient to index the above families by t and s, where the (t,s)-th elements will correspond to problems of "time" $T \approx 2^t$ and "space" $S \approx 2^s$.

Definition 6.18 (Succinct Algebraic Constraint Satisfaction)**.** Consider the following parameters:

1. a field size function $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, inducing a family of finite fields $\{\mathbb{F}_{t,s}\}_{t,s\in\mathbb{N}}$ where $\mathbb{F}_{t,s} = \mathbb{F}_2(\mathsf{x})$ and x is the root of $I_{t,s}$, which is the irreducible polynomial of degree $f(t,s)$ over $\mathbb{F}_2$ output by FindIrrPoly$(1^{f(t,s)})$;

2. two proper functions associated with the family $\mathbf{H}$ in Parameter 3:
   (a) a dimension function $\mathsf{m}_{\mathbf{H}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and
   (b) a time function $\mathsf{t}_{\mathbf{H}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

3. a family $\mathbf{H} = \{H_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $H_{t,s}$ is an $\mathsf{m}_{\mathbf{H}}(t,s)$-dimensional affine subspace of $\mathbb{F}_{t,s}$ specified by a basis $\mathcal{B}_{H_{t,s}}$ and an offset $\mathcal{O}_{H_{t,s}}$ for all $t, s \in \mathbb{N}$, and
   (b) there exists a $\mathsf{t}_{\mathbf{H}}$-time algorithm Find$\mathbf{H}$ such that $(\mathcal{B}_{H_{t,s}}, \mathcal{O}_{H_{t,s}}) = \text{Find}\mathbf{H}(1^t, 1^s)$ for all $t, s \in \mathbb{N}$;

4. three proper functions associated with the family $\mathbf{N}$ in Parameter 5:
   (a) a neighborhood size function $\mathsf{c}_{\mathbf{N}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$,
   (b) a time function $\mathsf{t}_{\mathbf{N}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and
   (c) a size function $\mathsf{s}_{\mathbf{N}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

5. a family $\mathbf{N} = \{\vec{N}_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $\vec{N}_{t,s} = (N_{t,s,i}\colon \mathbb{F}_{t,s} \to \mathbb{F}_{t,s})_{i=1}^{\mathsf{c}_{\mathbf{N}}(t,s)}$ is a vector of $\mathsf{c}_{\mathbf{N}}(t,s)$ neighbor polynomials over $\mathbb{F}_{t,s}$, and
   (b) there exists a $\mathsf{t}_{\mathbf{N}}$-time algorithm Find$\mathbf{N}$ such that $[N_{t,s,i}]^{\mathsf{A}} = \text{Find}\mathbf{N}(1^t, 1^s, i)$ is an $\mathsf{s}_{\mathbf{N}}$-size $\mathbb{F}_{t,s}$-arithmetic circuit computing $N_{t,s,i}$ for all $t, s \in \mathbb{N}$ and $i \in \{1, \ldots, \mathsf{c}_{\mathbf{N}}(t,s)\}$;

6. two proper functions associated with the family $\mathbf{P}$ in Parameter 7:
   (a) a time function $\mathsf{t}_{\mathbf{P}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, and
   (b) a size function $\mathsf{s}_{\mathbf{P}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$;

7. a family $\mathbf{P} = \{P_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $P_{t,s}\colon \mathbb{F}_{t,s}^{1+\mathsf{c}_{\mathbf{N}}(t,s)} \to \mathbb{F}_{t,s}$ is a constraint polynomial, and
   (b) there exists a $\mathsf{t}_{\mathbf{P}}$-time algorithm Find$\mathbf{P}$ such that $[P_{t,s}]^{\mathsf{A}} = \text{Find}\mathbf{P}(1^t, 1^s)$ is a $\mathsf{s}_{\mathbf{P}}$-size $\mathbb{F}_{t,s}$-arithmetic circuit computing $P_{t,s}$ for all $t, s \in \mathbb{N}$;

8. a proper function associated with the family $\mathbf{I}$ in Parameter 9:
   (a) a time function $\mathsf{t}_{\mathbf{I}}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

9. a family $\mathbf{I} = \{\vec{I}_{t,s}\}_{t,s\in\mathbb{N}}$ such that:
   (a) $\vec{I}_{t,s} = (I_{t,s,m})_{m=1}^{t}$ is a vector of affine subspaces each contained in $H_{t,s}$ specified by a basis $\mathcal{B}_{I_{t,s,m}}$ of m elements and an offset $\mathcal{O}_{I_{t,s,m}}$ for all $t, s \in \mathbb{N}$, and
   (b) there exists a $\mathsf{t}_{\mathbf{I}}$-time algorithm Find$\mathbf{I}$ such that $(\mathcal{B}_{I_{t,s,m}}, \mathcal{O}_{I_{t,s,m}}) = \text{Find}\mathbf{I}(1^t, 1^s, 1^m)$ for all $t, s \in \mathbb{N}$ and $m \in \{1, \ldots, t\}$.

The following **algebraic constraint** must hold:

$$\forall\, t, s \in \mathbb{N}\,,\ \deg\left(P_{t,s}\left(x, x^{(2^{\mathsf{m}_{\mathbf{H}}(t,s)}-1)\cdot\deg(N_{t,s,1})}, \ldots, x^{(2^{\mathsf{m}_{\mathbf{H}}(t,s)}-1)\cdot\deg(N_{t,s,\mathsf{c}_{\mathbf{N}}(t,s)})}\right)\right) \leq 2^{f(t,s)-2}\ . \quad (2)$$

The language sACSP, with respect to a choice $\mathsf{par}_{\text{sACSP}}$

$$\mathsf{par}_{\text{sACSP}} = \Big(f, (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \mathbf{H}), (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \mathbf{N}), (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \mathbf{P}), (\mathsf{t}_{\mathbf{I}}, \mathbf{I})\Big)$$

of the above parameters, consists of instances $(\mathtt{x}, 1^t, 1^s)$, where $\mathtt{x}$ is a binary input string and $t, s \in \mathbb{N}$ with $|\mathtt{x}| \in \{2, \ldots, 2^t\}$, such that there exists an assignment polynomial $A \colon \mathbb{F}_{t,s} \to \mathbb{F}_{t,s}$ of degree less than $2^{\mathsf{m}_{\mathbf{H}}(t,s)}$ for which the following two conditions hold:

(i) *Satisfiability of constraints.* For every element $\alpha(\mathsf{x}) \in H_{t,s}$,

$$P_{t,s}\Big(\alpha(\mathsf{x}), \big(A \circ N_{t,s,1}\big)(\alpha(\mathsf{x})), \ldots, \big(A \circ N_{t,s,\mathsf{c}_{\mathbf{N}}(t,s)}\big)(\alpha(\mathsf{x}))\Big) = 0_{\mathbb{F}_{t,s}} \ .$$

If so, we say that the assignment polynomial A *satisfies the constraint polynomial* $P_{t,s}$.

(ii) *Consistency with the input.* Letting $\alpha_i(\mathsf{x})$ be the i-th element in $I_{t,s,\log|\mathtt{x}|}$ for every index $i \in \{1, \ldots, |\mathtt{x}|\}$,

$$\mathtt{x} = \mathrm{bit}\big(A(\alpha_1(\mathsf{x}))\big) \cdots \mathrm{bit}\big(A(\alpha_{|\mathtt{x}|}(\mathsf{x}))\big) \ ,$$

where $\mathrm{bit} \colon \mathbb{F}_{t,s} \to \{0, 1, \perp\}$ maps $0_{\mathbb{F}_{t,s}}$ to 0, $1_{\mathbb{F}_{t,s}}$ to 1, and anything else to $\perp$. If so, we say that the assignment polynomial A *is consistent* with the input $\mathtt{x}$.

Similarly to the definition of sGCP (cf. Definition 6.15), the above definition for (univariate) succinct algebraic constraint satisfaction problems is quite a mouthful; again, this is because the language itself encodes requirements ensuring that "large objects" (in this case large subspaces of a large finite field, high-degree polynomials, and so on) can be computed using very few resources by using succinct and functional representations of such objects (for example, efficiently computable bases and offsets, efficiently constructible small arithmetic circuits, and so on). Ultimately, these requirements will enable the existence of a PCP system for sACSP with prover and verifier running times that are essentially optimal. (See [BSCGT12] for details.)

Remark 6.19. How is a choice of parameters $\mathsf{par}_{\text{sACSP}}$ for sACSP (concisely) specified? All the "complexity functions" from Definition 6.18 (i.e., those specifying running times, sizes of arithmetic circuits, and so on) were chosen to be proper (see Definition 6.1), and thus each has an efficient algorithm that computes it (which, for simplicity, we denote with the same name as the function); moreover, every infinite family comes with an algorithm that computes the information about the family we are interested in. Thus, a choice of parameters $\mathsf{par}_{\text{sACSP}}$ can be specified as follows:

$$\mathsf{par}_{\text{sACSP}} = \begin{pmatrix} f, \\ (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \textsc{Find}\mathbf{H}), \\ (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \textsc{Find}\mathbf{N}), \\ (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \textsc{Find}\mathbf{P}), \\ (\mathsf{t}_{\mathbf{I}}, \textsc{Find}\mathbf{I}) \end{pmatrix} \ .$$

An intuitive "meaning" for each parameter. At high level, a choice of parameters $\mathsf{par}_{\text{sACSP}}$ for sACSP identifies a collection of infinite families of objects (one object for each $t, s \in \mathbb{N}$). When an instance $(\mathtt{x}, 1^t, 1^s)$ is considered for membership in $\text{sACSP}(\mathsf{par}_{\text{sACSP}})$, the (t, s)-th element from each of these families in the collection is used to determine membership of the instance; candidate witnesses are low-degree polynomials over the field $\mathbb{F}_{t,s}$. Despite the long definition, these objects interact in natural ways, so we now go over the parameters from Definition 6.18 in less formal terms, explaining some of the intuition behind the design of the definition.

- **Parameter 1.** The parameter f governs the "growth rate" of the size of the finite fields in the family $\{\mathbb{F}_{t,s}\}_{t,s \in \mathbb{N}}$; reductions from different languages to sACSP may yield different choices of f, and, roughly, the slower-growing the function f is the more efficient is the reduction.

- **Parameter 2 and Parameter 3.** The parameter $\mathbf{H} = \{H_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of affine subspaces, where each $H_{t,s}$ is contained in the corresponding field $\mathbb{F}_{t,s}$. A "succinct" representation of this family is provided by the algorithm FIND$\mathbf{H}$, which, on input $(1^t, 1^s)$, generates a basis and offset for $H_{t,s}$. (Moreover, FIND$\mathbf{H}$ runs within the specified time complexity.) Intuitively, the subset $H_{t,s}$ is where "interesting things will happen", and the only reason for $H_{t,s}$ not being equal to the whole field $\mathbb{F}_{t,s}$ is that we need some "room" for some technical but essential conditions to go through (cf. Equation 2); for example, such conditions allow unique decoding of Reed-Solomon codes for certain distance ranges (see [BSCGT12] for more details).

- **Parameter 4 and Parameter 5.** The parameter $\mathbf{N} = \{\vec{N}_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of vectors of polynomials, where each vector $\vec{N}_{t,s}$ induces an "low-degree neighborhood" of size $\mathsf{c}_{\mathbf{N}}(t,s)$ for each element in the finite field $\mathbb{F}_{t,s}$. Later, a certain condition will impose a single "local" constraint on the values of (candidate-witness) low-degree polynomials at every such affine neighborhood that can be found in $H_{t,s}$. Again, the (description of) affine functions that define each affine low-degree neighborhoods can be computed via an algorithm FIND$\mathbf{N}$ (which runs with the prescribed time complexity). The restriction to affine neighborhoods is again technical but essential (cf. Equation 2).

- **Parameter 6 and Parameter 7.** The parameter $\mathbf{P} = \{P_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of constraint polynomials, where each polynomial determines which values of low-degree polynomials are valid for a given neighborhood (specifically, those values that make it zero). Each polynomial $P_{t,s}$ itself may not have low-degree (and usually it will not), nor may be sparse, though it will be ensured that it can be computed via a small arithmetic circuit that can be generated efficiently, via the algorithm FIND$\mathbf{P}$.

At high level, the parameters described until now could, say, encode a variety of constraints that ensure "correct computation" (say, of a Turing machine, or a random-access machine), but so far did not encode anything about whether this correct computation had anything to do with the input $\mathtt{x}$ under consideration. The remaining parameters relate the computation to the input at hand.

- **Parameter 8 and Parameter 9.** The parameter $\mathbf{I} = \{\vec{I}_{t,s}\}_{t,s\in\mathbb{N}}$ is a family of vectors of affine subspaces where each subspace in $\vec{I}_{t,s}$ is contained in the corresponding affine subspace $H_{t,s}$. The appropriate subspace in $\vec{I}_{t,s}$ identifies on which "part" of $H_{t,s}$ a (candidate-witness) low-degree polynomial must contain information related to the input $\mathtt{x}$; specifically, when such information is concatenated, one should obtain $\mathtt{x}$. As usual, we need an algorithm FIND$\mathbf{I}$ to ensure that generating succinct representations of subspaces in $\vec{I}_{t,s}$ can be done appropriately efficiently, and this is the case as the definition requires that a basis and offset for it is easy to find.

In summary, the "satisfiability of constraints" condition of the definition should be interpreted as saying that some computation (the particulars of which are captured by the specific choices of parameters) was performed correctly, while the other condition, "consistency with the input", should be interpreted as saying that this computation was performed on the input at hand.

7 From $\mathrm{BH}_{\mathsf{RAM}}$ To sGCP

In Section 4 we discussed a high-level strategy for proving Theorem 2 in two steps (respectively discussed in Section 4.1 and Section 4.2). The goal of this section is to prove in detail the first step, in the special case $s = t$. In this special case, we can directly use De Bruijn graphs instead of relying on Beneš networks.

Given the "template" Definition 6.3, we give here the following specialized definition:

Definition 7.1. *Fix a class of random-access machines $\mathcal{P}_{\mathrm{RAM}}$. We say that a pair of polynomial-time-computable functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a* **Levin reduction** *from $\mathrm{BH}_{\mathsf{RAM}}$ to sGCP with respect to $\mathcal{P}_{\mathrm{RAM}}$ if the following three conditions are satisfied for every choice of random-access machine $M \in \mathcal{P}_{\mathrm{RAM}}$:*

1. $\mathsf{par}_{\mathrm{sGCP}} := F_{\mathsf{p}}(M)$ *is a choice of parameters for* sGCP.
2. *For every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$, $(\mathtt{x}, 1^t) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ if and only if $(\mathtt{x}, 1^t) \in \mathrm{sGCP}(\mathsf{par}_{\mathrm{sGCP}})$.*
3. *For every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$, if $(\mathtt{w}, \vec{S})$ is a witness to "$(\mathtt{x}, 1^t) \in \mathrm{BH}_{\mathsf{RAM}}(M)$" then the coloring $C := F_{\mathsf{w}}(M, 1^{2^t}, (\mathtt{w}, \vec{S}))$ is a witness to "$(\mathtt{x}, 1^t) \in \mathrm{sGCP}(\mathsf{par}_{\mathrm{sGCP}})$".*

The two functions F_{p} and F_{w} are respectively called as the "parameter reduction" and the "witness reduction".

High-level idea. Our goal is to construct a Levin reduction from bounded-halting problems on random-access machines to succinct GCPs. Our strategy is to single out the constraints that need to be met in order for a computation of a random-access machine to be valid, accepting, and consistent with the given input; these constraints fall into two categories: constraints to ensure code consistency (i.e., the transition function of the machine correctly went from the current state to the next one) and constraints to ensure memory consistency (i.e., each memory load returns the last value that was written there). We express these constraints as edge constraints in a graph, which we then "structure" by using routing techniques. The resulting coloring problem can be then formalized as a succinct GCP problem.

Our plan, step by step. We construct the Levin reduction in four steps:

- **Step 1 (Section 7.1).** For a given random-access machine M, sequence of configurations $\vec{S}$, and input strings $(\mathtt{x}, \mathtt{w})$, we show that the problem of whether $\vec{S}$ is accepting and valid for M and is consistent with $(\mathtt{x}, \mathtt{w})$ (recall Definition 6.9, Definition 6.13, and Definition 6.10) can be equivalently stated as whether a corresponding "computation graph" $G_M^{\vec{S}}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$ exists or not.

  Informally, the vertices of a computation graph $G_M^{\vec{S}}$ are the (time-stamped) configurations in $\vec{S}$; the validity of $G_M^{\vec{S}}$ can be determined by verifying, at every edge of the graph, a *local* constraint (which can be generated easily from M) that only takes as input the configuration of each of the two vertices of the edge.

  However, different sequences of configurations for the same M determine different sets of edges allowable in a valid computation graph (if one exists).

- **Step 2 (Section 7.2).** For a given random-access machine M, sequence of configurations $\vec{S}$, and input strings $(\mathtt{x}, \mathtt{w})$, we show that the problem of determining whether a computation graph $G_M^{\vec{S}}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$ exists or not can be equivalently stated as whether a corresponding "computation routing" $R_M^{\vec{S}}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$ exists or not.

Informally, the validity of $R_M^{\vec{S}}$ can be determined by verifying, at every vertex of a *fixed* routing network, a local constraint (which can be generated easily from M) that only takes as input information available at the vertex and its neighbors.

Intuitively, we do so by leveraging the rearrangeability of De Bruijn graphs in order to "route" vertices of a computation graph $G_M^{\vec{S}}$.

- **Step 3 (Section 7.3).** For a given random-access machine M and input string $\mathtt{w}$, we show how the problem of whether there exists a second input string $\mathtt{w}$, a sequence of configurations $\vec{S}$, and a computation routing $R_M^{\vec{S}}$ (for M with respect to $\vec{S}$) that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$ exists or not can be formalized as a problem of deciding whether $(\mathtt{x}, 1^t)$ is in $\mathrm{sGCP}(\mathsf{par}_{\mathrm{sGCP}})$, for an appropriate choice of $\mathsf{par}_{\mathrm{sGCP}}$ depending on M.

  Informally, we do so by noting that figuring out which constraint to apply at a given vertex is a simple operation that can easily be made to depend on a properly abstracted "vertex type"; we can then "bunch up" all the requisite constraints into a universal constraint that needs to be enforced at every vertex of the graph.

- **Step 4 (Section 7.4).** We conclude by writing down explicitly the two functions F_{p} and F_{w} of the Levin reduction and explaining why they are correct.

We now proceed to describe each of the four steps in detail. Throughout, it will be useful to have the definitions from Section 6.4 in mind.

7.1 Step 1: From RAMs to computation graphs

We begin by defining what it means for a timestamp-configuration pair (τ, S) to "precede in memory" another timestamp-configuration pair (τ', S'). Intuitively, this will be so when both configurations correspond to memory operations, and the second configuration has a memory dependency on the first one. (The definition is in fact slightly more complicated, because we shall eventually consider memory accesses that are somewhat canonical; see Definition 7.7.)

Definition 7.2. *Let M be a random-access machine, $S = [\mathsf{pc}, r_0, \ldots, r_{k-1}]$ and $S' = [\mathsf{pc}', r'_0, \ldots, r'_{k-1}]$ two configurations of M, and τ and τ' non-negative integers. We say that (τ, S)* **precedes** *(τ', S')* **in memory**, *denoted $(\tau, S) \overset{\mathrm{m}}{\prec} (\tau', S')$, if pc and pc' each point to a store or pload instruction and at least one of the following three conditions is satisfied: letting r_j and $r'_{j'}$ be the two memory addresses respectively accessed and i and i' the two registers respectively read/written,*

(i) S and S' correspond to accesses to the same memory address (i.e., $\tau < \tau'$ and $r_j = r'_{j'}$) and the accesses are consistent (i.e., if both S' and S' are pload instructions, then the value loaded by both S and S' is the same; if S is a store instruction and S' is a pload instruction, then the value loaded in S' is the same as the value stored in S).

(ii) S and S' correspond to accesses to different memory addresses (i.e., $r_j < r'_{j'}$); moreover, if pc' points to a pload instruction then the loaded value is 0 (i.e., $r_{i'} = 0$).

(iii) S and S' correspond to accesses to memory where the second configuration is the first in the computation and thus it must write to the first memory cell (i.e., $\tau' = 0$, $r'_{j'} = 0$ and the instruction pointed by pc' is a store instruction).

Next, for a random-access machine M and sequence of configurations $\vec{S}$, we define its "computation graph" $G_M^{\vec{S}}$. Intuitively, it is a graph where the edges consist of all the configurations, and

the edges consist of "time edges" (connecting successive configurations in time) and "memory edges" (allegedly, connecting configurations that are successive "in memory").

Definition 7.3. *Let M be a random-access machine and $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M. A* **computation graph** *for M with respect to $\vec{S}$ is a triple $G_M^{\vec{S}} = (V, E_{\mathsf{t}}, E_{\mathsf{m}})$ such that $(V, E_{\mathsf{t}} \cup E_{\mathsf{m}})$ is a graph satisfying the following two conditions:*

(i) the set of vertices V is equal to $\{(\tau, S_\tau)\}_{\tau \in \{0,\ldots,T-1\}}$; and

(ii) the set of edges E_{t} is equal to $\{((\tau, S_\tau), (\tau + 1 \bmod T, S_{\tau+1 \bmod T}))\}_{\tau \in \{0,\ldots,T-1\}}$.

Intuitively, a computation graph $G_M^{\vec{S}}$ is valid if its time edges respect the transition function and its memory edges are successive "in memory"; it is accepting if $\vec{S}$ is also, and it is consistent with the input strings if $\vec{S}$ is also.

Definition 7.4. *Let M be a random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, $(\mathtt{x}, \mathtt{w})$ two input strings, and $G_M^{\vec{S}}$ a computation graph for M with respect to $\vec{S}$.*

- *We say that $G_M^{\vec{S}}$ is* **accepting** *if $\vec{S}$ is accepting for M. (Recall Definition 6.9.)*
- *We say that $G_M^{\vec{S}}$ is* **consistent** *with $(\mathtt{x}, \mathtt{w})$ if $\vec{S}$ is consistent with $(\mathtt{x}, \mathtt{w})$. (Recall Definition 6.10.)*
- *We say that $G_M^{\vec{S}}$ is* **valid** *if it satisfies the following conditions:*
  - *(i) For every edge $((\tau, S_\tau), (\tau + 1 \bmod T, S_{\tau+1 \bmod T})) \in E_{\mathsf{t}}$, $S_\tau \rightsquigarrow S_{\tau+1 \bmod T}$. (Recall Definition 6.11.)*
  - *(ii) For every edge $((\tau, S), (\tau', S')) \in E_{\mathsf{m}}$, $(\tau', S') \overset{\mathrm{m}}{\prec} (\tau, S)$. (Recall Definition 7.2.)*
  - *(iii) For every vertex (τ, S_τ) such that the instruction in S_τ is a* pload *or a* store *instruction, the in-degree and out-degree of (τ, S_τ) in E_{m} is equal to 1.*

We show that in a valid computation graph $G_M^{\vec{S}}$ every vertex whose configuration points to a store instruction or a pload instruction can be reached and can reach the "initial" vertex $(0, S_0)$.

Lemma 7.5. *Let M be a random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, and $G_M^{\vec{S}}$ a computation graph for M with respect to $\vec{S}$. Suppose that $G_M^{\vec{S}}$ is valid. Then, for any $(\tau, S) \in V$ such that* pc *in S points to a* store *instruction or a* pload*,*

(i) there exists a path in E_{m} from (τ, S) to $(0, S_0)$ and

(ii) there exists a path in E_{m} from $(0, S_0)$ to (τ, S).

Proof. We only prove (i); a similar argument can be made to prove (ii).

Let (τ, S) and (τ', S') be vertices in V such that both S and S' point to a pload or a store instruction. Let $S = [\mathsf{pc}, r_0, \ldots, r_{k-1}]$ and $S' = [\mathsf{pc}', r'_0, \ldots, r'_{k-1}]$, and let r_j and $r'_{j'}$ be the two memory addresses respectively accessed and i and i' the two registers respectively read/written by the two instructions pointed to by pc and pc'.

For the purpose of this proof, we write $S' \prec S$ if either "$\tau' < \tau$ and $r'_{j'} \leq r_j$" or "$r'_{j'} < r_j$". Let X be the set of those S_τ in $\vec{S}$ pointing to a pload or a store instruction.

Note that the relation $\prec$ is well-founded with respect to X.[15] Assume by way of contradiction that this is not the case, i.e., that there exists $(k, \widetilde{S}) \in V$ such that $\widetilde{S}$ contains a pload or a store

[15] Recall that a binary relation is *well-founded* with respect to a set X if every non-empty subset of X has a minimal element with respect to the relation.

instruction and there is no path from $(k, \widetilde{S})$ to $(0, S_0)$ using only E_{m} edges. Let P be the set of all vertices $(k', \widetilde{S}')$ for which there exists a path from $(k, \widetilde{S})$ to $(k', \widetilde{S}')$ using only edges in E_{m}. Let $(k', \widetilde{S}')$ be a minimal element of P. Since $(0, S_0) \notin P$, we have that $(k', \widetilde{S}') \neq (0, S_0)$. By Item (iii) of Definition 7.4 there exists $(k'', \widetilde{S}'')$ such that $((k', \widetilde{S}'), (k'', \widetilde{S}'')) \in E_{\mathsf{m}}$. By Item (ii) of Definition 7.4 we obtain that $(k'', \widetilde{S}'') \overset{\mathrm{m}}{\prec} (k', \widetilde{S}')$ and therefore, since $(k', \widetilde{S}') \neq (0, S_0)$, we have that $\widetilde{S}'' \prec \widetilde{S}$ as defined above.

To complete the proof, we distinguish two cases:

- *Case 1:* $(k'', \widetilde{S}'') \in P$. Since $\widetilde{S}'' \prec \widetilde{S}'$, we have reached a contradiction to the fact that $(k', \widetilde{S}')$ is a minimal element of P.
- *Case 2:* $(k'', \widetilde{S}'') \notin P$. Since $((k', \widetilde{S}'), (k'', \widetilde{S}'')) \in E_{\mathsf{m}}$ and since $(k', \widetilde{S}') \in P$, we conclude that there exists a path from $(k, \widetilde{S})$ to $(k'', \widetilde{S}'')$, which is a contradiction to $(k'', \widetilde{S}'')$ not being in P.

□

We can now use the previous lemma to show that in a valid computation graph $G_M^{\vec{S}}$ there is a *single* cycle passing through all the vertices whose configurations point to a store instruction or a pload instruction.

Lemma 7.6. *Let M be a random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, and $G_M^{\vec{S}}$ a computation graph for M with respect to $\vec{S}$. Suppose that $G_M^{\vec{S}}$ is valid. Then there exists a cycle K in $G_M^{\vec{S}}$, using only edges in E_{m}, that goes through $(0, S_0)$ and contains all those (τ, S) for which the program counter in S points to a* pload *or a* store *instruction.*

Proof. By Lemma 7.5, since $G_M^{\vec{S}}$ is valid, we know that every (τ, S) for which pc in S points to a pload or a store instruction appears on some cycle K_τ that goes through $(0, S_0)$ and (τ, S). By definition of a computation graph, $(0, S_0)$ has only in degree and out degree equal to 1 in E_{m}. Thus, only one E_{m} cycle can go through $(0, S_0)$ and therefore there must exist a single cycle K such that $K = K_\tau$ for all τ. □

We now focus only on those random-access machines that satisfy the very minor requirement of writing to the first memory cell in the first computation step and have a pload instruction after every load instruction. (which we can certainly assume without loss of generality).

Definition 7.7. *Let M be a random-access machine. We say that M is* **memory-well-behaved** *if M always writes to first memory cell in the first step and after every* load *instruction M performs a* pload *instruction with the same operands. In addition, we require that the two operands of a* load *instruction will be always different (i.e. we forbid* load r_i, r_i *commands for all possible i).*

We finally have the tools to prove that a computation graph $G_M^{\vec{S}}$ is valid (as well as accepting, and consistent with input strings x and w) if and only if $\vec{S}$ is valid for M (as well as accepting, and consistent with input strings x and w), as long as M is memory-well behaved.

Claim 7.8. *Let M be a memory-well-behaved random-access machine, $\vec{S} = (S_0, \ldots, S_{T-1})$ a sequence of configurations for M, and $(\mathtt{x}, \mathtt{w})$ two input strings. Then $\vec{S}$ is valid and accepting for M and consistent with $(\mathtt{x}, \mathtt{w})$ if and only if there exists a computation graph $G_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting and consistent with $(\mathtt{x}, \mathtt{w})$.*

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Assume that $\vec{S}$ is valid and accepting for M and consistent with $(\mathtt{x}, \mathtt{w})$. Let $G_M^{\vec{S}} = (V, E_\mathsf{t}, E_\mathsf{m})$ be the computation graph where we construct E_m as follows. For every node $(\tau, S) \in V$ such that the program counter in S points to a store instruction or a pload instruction:

- If S is the first configuration accessing a memory cell a_1 such that $\tau \neq 0$, let (τ', S') be the last configuration accessing a memory cell $a_2 < a_1$ such that all the cells between a_2 and a_1 are not accessed in $\vec{S}$, and add the edge $((\tau, S), (\tau', S'))$ to E_m.
- If $\tau = 0$, let (τ', S') be the last configuration in $\vec{S}$ containing a store or a pload instruction, and add the edge $((\tau, S), (\tau', S'))$ to E_m.
- If S is not the first configuration accessing a memory cell a, let (τ', S') with $\tau' < \tau$ be another configuration accessing a such that all configurations between S' and S in $\vec{S}$ do not access a, and add the edge $((\tau, S), (\tau', S'))$ to E_m.

We now argue that $G_M^{\vec{S}}$ is valid, accepting and consistent with $(\mathtt{x}, \mathtt{w})$. Indeed, since $G_M^{\vec{S}}$ is a computation graph for M with respect to $\vec{S}$ by the first two items of Definition 7.4 we obtain that $G_M^{\vec{S}}$ is accepting and consistent with $(\mathtt{x}, \mathtt{w})$. Regarding the valid requirement, first, notice that since $G_M^{\vec{S}}$ is a computation graph, it immediately satisfies the first requirement of the valid section of Definition 7.4. Next, if $((\tau, S), (\tau', S')) \in E_\mathsf{m}$ then (τ', S') precedes (τ, S) in memory. In addition, since the cases above do not overlap and each configuration S that contains a store or a pload instruction has a case corresponding to it, we obtain that the degree in E_m of each node (τ, S) such that S contains a store instruction or a pload instruction is 1. This fulfills the last two requirements of the valid section of Definition 7.4.

(2) Conversely, assume that $G_M^{\vec{S}}$ is valid, accepting and consistent with $(\mathtt{x}, \mathtt{w})$. By the first two sections of Definition 7.4 we obtain that $\vec{S}$ is accepting and consistent with $(\mathtt{x}, \mathtt{w})$. By definition of E_t, we have that, for every $0 \leq \tau \leq T$, $S_\tau \rightsquigarrow S_{\tau+1 \bmod T}$. Thus, the only thing that is left to prove is that the memory of the random-access machine behaves as defined in the code of the machine (i.e., that every load instruction indeed receives the value that was stored in that memory cell at the last store to it).

Assume by way of contradiction that there exists a configuration containing a load instruction in $\vec{S}$ such that the value received by this load in not the last value stored in that cell by the previous store or not 0 (if this is the first load that accesses that cell). Let S_τ be the first such configuration. By Lemma 7.6 there exists a cycle K through $(0, S_0)$ in $G_M^{\vec{S}}$ such that K contains only edges from E_m and every pload or store instruction in $\vec{S}$ is present of this cycle.

Furthermore, K is the only E_m cycle in $G_M^{\vec{S}}$. Let (r, S_r) be the vertex following $(\tau + 1, S_{\tau+1})$ in K, since the out degree of (τ, S_τ) in E_m is 1, (r, S_r) is the only vertex that has an E_m edge from (τ, S_τ). Finally, we distinguish between several cases, according to (r, S_r):

- If (r, S_r) contains a store instruction for the same cell that $(\tau+1, S_{\tau+1})$ accesses, since the value loaded in (τ, S_τ) is not the value stored in (r, S_r), we obtain that the edge $((\tau+1, S_{\tau+1}), (r, S_r))$ is not memory ordered which is a contradiction to the first part of Definition 7.2.
- If $(r-1, S_{r-1})$ contains a load instruction for the same cell that (τ, S_τ) accesses and the value loaded in (τ, S_τ) is not the value loaded in $(r-1, S_{r-1})$, then the edge $((\tau+1, S_{\tau+1}), (r, S_r))$ is not memory ordered which is a contradiction to the first part of Definition 7.2.

- If (r, S_r) contains a store or pload instruction that accesses a different cell than (τ, S_τ) accesses, since the value loaded in (τ, S_τ) is not 0 we obtain that the the edge $((\tau+1, S_{\tau+1}), (r, S_r))$ is not memory ordered. This is a contradiction to the second or third part of Definition 7.2 (based on whether $r = T$ or not).

□

Witness reduction. Note that not only does Claim 7.8 tell us an equivalence between the two decision problems, but its proof tells us how deduce a "witness" too. Specifically, given M, $\vec{S}$, and $(\mathtt{x}, \mathtt{w})$, we can construct the graph $G_M^{\vec{S}}$ by deciding which edges should go in E_{m} by following the three bullet points in part **(1)** of the proof. (As for the reverse direction, part **(2)** of the proof gives us that, but we are not interested in "going back".)

7.2 Step 2: From computation graphs to (double) De Bruijn graphs

Recall that extended De Bruijn graphs (see Definition 6.4), when "sufficiently wide", are rearrangeable (see Claim 6.5). At high level, we wish to leverage the rearrangeability property De Bruijn graphs to ensure that the edges of a computation graph do not depend on the particular sequence of configurations under consideration.

Looking back at Definition 7.4, we note that the two sets of edges E_{t} and E_{m} of a computation graph are "treated differently" in the sense that exactly which edges are allowed in each of those sets in a valid computation graph is different. More precisely, the edges in E_{t} (i.e., the "time edges") induce a permutation on the vertices consisting of a configuration being mapped to the successive configuration in time; the edges in E_{m} (i.e., the "memory edges") induce a permutation on the vertices consisting of a configuration being mapped to the successive configuration in memory (according to Definition 7.2).

We shall route each of the two permutations with an extended De Bruijn graph.

We thus begin by defining a "double" extended De Bruijn graph, which is simply a graph with two extended De Bruijn graphs side by side, connected at the 0-th column.[16]

Definition 7.9. *Let κ and L be two positive integers. The (κ, L)* **double extended De Bruijn graph***, denoted $\mathrm{DDB}(\kappa, L)$, is a 3-regular directed graph consisting of the Cartesian graph product of the directed two-cycle and $\mathrm{DB}(\kappa, L)$. In other words, the vertex set V of $\mathrm{DDB}(\kappa, L)$ consists of vertices $v = (b, i, w)$, where $b \in \{0, 1\}$, $i \in \{0, \ldots, L-1\}$, and $w \in \{0,1\}^\kappa$, and the edge set E is induced by the following three neighbor functions:*

- $\Gamma_1\big((b, i, w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w))$, *i.e., one kind of De Bruijn edges;*
- $\Gamma_2\big((b, i, w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w) \oplus e_1)$, *i.e., the other kind of De Bruijn edges; and*
- $\Gamma_3\big((b, i, w)\big) = (b \oplus 1, i, w)$, *i.e., edges between corresponding vertices of the two De Bruijn graphs.*

[16] We note here that one of the two permutations to be routed is always "nice": namely the permutation induced by the "time edges" E_{t} is always the "$+1 \bmod T$" permutation that, because of its strong locality, does not need an entire permutation network to be routed. (In fact, we could even avoid routing it by adding "fixed" $+1 \bmod T$ edges in the first column of the "memory edge" De Bruijn graph, without bringing in another De Bruijn graph.) However, we still route both permutations, each on a De Bruijn graph, because later on, during arithmetization, we will need a "well-structured" graph.

Remark 7.10. A (κ, L) double extended De Bruijn graph is simply two (κ, L) extended De Bruijn graphs "side by side", connected with bi-directional edges at corresponding vertices. (Compare with Definition 6.4, where single De Bruijn graphs are defined.) While the only edges between the two graphs that we will need are the edges between the two 0-th columns, for convenience we still define edges between the two graphs at every node, in order to ensure that the graph is 3-regular.

Given $t \in \mathbb{N}$, let G_t be the graph $\mathrm{DDB}(\kappa, L)$ where $\kappa = t$ and $L = 4t - 1$, i.e., the $(t, 4t-1)$ double extended De Bruijn graph. By Claim 6.5 we can route on G_t any *two* permutations of 2^t elements.

We now define the analogue of a computation graph (Definition 7.3) for double extended De Bruijn graphs, which we call a "computation routing"; intuitively, it is a coloring $R_M^{\vec{S}}$ of the graph G_t such that the 0-th columns of both extended De Bruijn graphs contain the sequence of configurations $\vec{S}$ (whose length we now take without loss of generality to be a power of 2) and other columns may contain any configuration.

Definition 7.11. *Let M be a random-access machine and $\vec{S} = (S_0, \ldots, S_{2^t-1})$ a sequence of configurations for M. We say that a coloring $R_M^{\vec{S}}$ of G_t is a* **computation routing** *for M with respect to $\vec{S}$ if, for every $b \in \{0,1\}$, $i \in \{0, \cdots, L-1\}$, and $w \in \{0,1\}^\kappa$ it holds that*

$$(s, \tau, S) = R_M^{\vec{S}}\big((b, i, w)\big) \in \{0,1\} \times \{0,1\}^\kappa \times \mathcal{S}$$

where $\mathcal{S}$ is the set of all possible configurations of M.

Intuitively, the bit s denotes whether at the given node the routing of the packet, consisting of a "timestamp" τ and configuration S, is done by forwarding the packet "straight ahead" or "diagonally".

Analogously to the notion of a valid computation graph (Definition 7.4), we now define a valid computation routing; intuitively, we need to ensure that, in the columns other than the 0-th one, routing constraints are respected, and that between the last and 0-th columns the appropriate code and memory consistency checks are performed (as well as ensuring that the two 0-th columns are consistent with each other).

Definition 7.12. *Let M be a random-access machine, $\vec{S} = (S_0, \ldots, S_{2^t-1})$ a sequence of configurations for M, $(\mathtt{x}, \mathtt{w})$ two input strings, and $R_M^{\vec{S}}$ a computation routing for M with respect to $\vec{S}$.*

- *We say that $R_M^{\vec{S}}$ is* **accepting** *if $\vec{S}$ is accepting. (Recall Definition 6.9.)*
- *We say that $R_M^{\vec{S}}$ is* **consistent** *with $(\mathtt{x}, \mathtt{w})$ if $\vec{S}$ is consistent with $(\mathtt{x}, \mathtt{w})$. (Recall Definition 6.10.)*
- *We say that $R_M^{\vec{S}}$ is* **valid** *if it satisfies the following conditions:*

  (i) The routing "packets" are initialized correctly. *For every $b \in \{0,1\}$ and $w \in \{0,1\}^\kappa$, it holds that $(s, w, S_w) = R_M^{\vec{S}}((b, 0, w))$ for some s and one of the following conditions holds:*
  - $(R_M^{\vec{S}} \circ \Gamma_1)((b,0,w)) = (1, w, S_w)$ *and* $\Big((R_M^{\vec{S}} \circ \Gamma_2)((b,0,w))\Big)_0 = 1$ *or,*
  - $(R_M^{\vec{S}} \circ \Gamma_2)((b,0,w)) = (0, w, S_w)$ *and* $\Big((R_M^{\vec{S}} \circ \Gamma_1)((b,0,w))\Big)_0 = 0$.

  (ii) The routing constraints are respected. *For every $b \in \{0,1\}$, $i \in \{1, \ldots, L-2\}$, and $w \in \{0,1\}^\kappa$, letting $(s', \tau', S) = R_M^{\vec{S}}((b, i, w))$, one of the following conditions holds:*
  - $(R_M^{\vec{S}} \circ \Gamma_1)((b,i,w)) = (1, \tau', S)$ *and* $\Big((R_M^{\vec{S}} \circ \Gamma_2)((b,i,w))\Big)_0 = 1$ *or,*

– $(R_M^{\vec{S}} \circ \Gamma_2)((b,i,w)) = (0,\tau',S)$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((b,i,w))\right)_0 = 0$.

(iii) Code consistency is maintained. *For every* $w \in \{0,1\}^\kappa$*, letting* $(s',\tau',S) = R_M^{\vec{S}}((0,L-1,w))$*, one of the following holds:*

– $(R_M^{\vec{S}} \circ \Gamma_1)((0,L-1,w)) = (1,\tau'+1 \bmod 2^t,S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_2)((0,L-1,w))\right)_0 = 1$ *and* $S \rightsquigarrow S'$ *or,*

– $(R_M^{\vec{S}} \circ \Gamma_2)((0,L-1,w)) = (0,\tau'+1 \bmod 2^t,S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((0,L-1,w))\right)_0 = 0$ *and* $S \rightsquigarrow S'$.

(iv) Memory consistency is maintained. *For every* $w \in \{0,1\}^\kappa$*, letting* $(s',\tau',S) = R_M^{\vec{S}}((1,L-1,w))$*, one of the following holds:*

– $(R_M^{\vec{S}} \circ \Gamma_1)((1,L-1,w)) = (1,\tau'',S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_2)((1,L-1,w))\right)_0 = 1$ *and if both* S *and* S' *contain* pload *or* store *instructions we require that* $(\tau',S) \overset{\text{m}}{\prec} (\tau'',S')$ *and otherwise we require that* $(\tau'' = \tau' \wedge S' = S)$*, or*

– $(R_M^{\vec{S}} \circ \Gamma_2)((1,L-1,w)) = (0,\tau'',S')$ *and* $\left((R_M^{\vec{S}} \circ \Gamma_1)((0,L-1,w))\right)_0 = 0$ *and if both* S *and* S' *contain* pload *or* store *instructions we require that* $(\tau',S) \overset{\text{m}}{\prec} (\tau'',S')$ *and otherwise we require that* $(\tau'' = \tau' \wedge S' = S)$.

(v) Column-0 consistency is maintained. *For any* $w \in \{0,1\}^\kappa$ *it holds that*

$$(R_M^{\vec{S}} \circ \Gamma_3)((1,0,w)) = (s,w,S_w) \text{ and } (R_M^{\vec{S}} \circ \Gamma_3)((0,0,w)) = (s',w,S_w)$$

for some $s,s' \in \{0,1\}$.

It is easy to see that any two routing on G_t of the same packets of the form $\{(w,S_w)\}_{w\in\{0,1\}^\kappa}$ can be viewed as a computation routing $R_M^{\vec{S}}$ satisfying Item (i), Item (ii), and Item (v) of Definition 7.12.

We now formally establish the connection between computation graphs and computation routings.

Claim 7.13. *Let* M *be a random-access machine and let* $\vec{S} = (S_0,\ldots,S_{2^t-1})$ *a sequence of configurations for* M*, and* $(\mathtt{x},\mathtt{w})$ *two input strings. Then there exists a computation graph* $G_M^{\vec{S}}$ *for* M *with respect to* $\vec{S}$ *that is valid, accepting, and consistent with* $(\mathtt{x},\mathtt{w})$ *if and only if there exists a computation routing* $R_M^{\vec{S}}$ *for* M *with respect to* $\vec{S}$ *that is valid, accepting, and consistent with* $(\mathtt{x},\mathtt{w})$.

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Assume that $G_M^{\vec{S}}$ is a computation graph for M with respect to $\vec{S}$ that is valid, accepting and consistent with $(\mathtt{x},\mathtt{w})$. We construct a valid computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ as follows:

1. First, $R_M^{\vec{S}}$ colors the zero layers G_t such that $R_M^{\vec{S}}((1,0,w)) = R_M^{\vec{S}}((0,0,w)) = (0,w,S_w)$.
2. For any edge $((\tau,S),(\tau',S')) \in E_{\mathsf{m}}$, the coloring $R_M^{\vec{S}}$ of G_t is updated so it expresses the routing of the node $(1,0,\tau')$ to the node $(1,0,\tau)$.
3. For any node $(1,0,\tau)$ not routed in the previous case, the coloring $R_M^{\vec{S}}$ of G_t is updated so it expresses the routing of the node $(1,0,\tau)$ to itself.
4. For any edge $((\tau,S),(\tau+1 \bmod 2^t,S')) \in E_{\mathsf{t}}$, the coloring $R_M^{\vec{S}}$ of G_t is updated so it expresses the routing of the node $(0,0,\tau)$ to the node $(0,0,\tau+1 \bmod 2^t)$.

We now argue that $R_M^{\vec{S}}$ is a valid, accepting and consistent with $(\mathtt{x}, \mathtt{w})$ computation routing.

By the first two parts of Definition 7.12 it follows that $R_M^{\vec{S}}$ is accepting and consistent with $(\mathtt{x}, \mathtt{w})$.

Regarding the validity requirement, by Lemma 7.6 the edges in E_{m} form a cycle such that every configuration S that contains a pload instruction is on the cycle. In addition, we notice that a cycle induces a permutation on its nodes. Also, we notice that Item 3 above also defines a permutation but on the nodes "skipped" by part Item 2. Therefore Item 2 and Item 3 define a permutation on the nodes of the form $(1, 0, \tau)$ for some $\tau \in \{0, \ldots, 2^t - 1\}$. Next we look on the edges used in Item 4 of the construction above. We notice that this requirement also induces a permutation on the nodes of the form $(0, 0, \tau)$ for some $\tau \in \{0, \ldots, 2^t - 1\}$. Because G_t can route any permutation from layer 0 to layer 0 by setting the nodes in the other layers of the graph to be a "straight ahead" or a "diagonal", the two permutations defined by Item 2, Item 3, and Item 4 above can be routed. This fulfills the requirement posed in Item (i) and Item (ii) of Definition 7.12.

Next, by Item 4 of the construction above we have that $(0, 0, \tau)$ is routed to $(0, 0, \tau + 1 \bmod 2^t)$ and, by definition of E_{t}, we have that, for all τ, $S_\tau \rightsquigarrow S_{\tau+1 \bmod 2^t}$. Thus, the requirement of Item (iii) of Definition 7.12 is fulfilled.

We turn our attention to Item 2 and Item 3 of the construction above. Since $G_M^{\vec{S}}$ is a valid computation graph we have that if $(1, 0, \tau')$ is routed to $(1, 0, \tau)$ then $(\tau', S_{\tau'}) \overset{\mathrm{m}}{\prec} (\tau, S_\tau)$ if both S_τ and $S_{\tau'}$ contain pload or store instructions and $S_\tau = S_{\tau'}$ otherwise. Therefore Item (iv) of Definition 7.12 is also fulfilled. Regarding the requirement posed in Item (v) of Definition 7.12, we notice that Item 1 in the construction of $R_M^{\vec{S}}$ colors all the nodes in layer zero in a way that respects the requirement posed in Item (v) of Definition 7.12. In addition, since all the other items in the construction do not update the configuration part of the colors of layer 0 we obtain that $R_M^{\vec{S}}$ fulfills the requirement of Item (v) of Definition 7.12.

Thus we get that $R_M^{\vec{S}}$ is a valid computation routing.

(2) Conversely, assume that $R_M^{\vec{S}}$ is a valid computation routing. Let $G_M^{\vec{S}} = (V, E_{\mathsf{t}}, E_{\mathsf{m}})$ be the graph that is defined as follows

- $V = \{(\tau, S_\tau) : R_M^{\vec{S}}((0, 0, \tau)) = (0, \tau, S_\tau) \text{ and } \tau \in \{0, \ldots, 2^t - 1\}\}$,
- $E_{\mathsf{t}} = \{((\tau, S_\tau), (\tau', S_{\tau'})) : (0, 0, \tau) \text{ is routed to } (0, 0, \tau')\}$, and
- $E_{\mathsf{m}} = \{((\tau, S_\tau), (\tau', S_{\tau'})) : (1, 0, \tau') \text{ is routed to } (1, 0, \tau) \text{ and } S_\tau, S_{\tau'} \text{ both contain pload or store instruction}$

We now argue that $G_M^{\vec{S}}$ is a computation graph that is valid, accepting and consistent with $(\mathtt{x}, \mathtt{w})$.

We begin by proving that $G_M^{\vec{S}}$ is a computation graph. The first item of Definition 7.3 immediately follows from the definition of V. As for the second item, we notice that by the first two items in Definition 7.12 we obtain that the configuration and timestamp part of the node's color remains unchanged throughout the different layers. These constraints, combined with the constraints of Item (iii) of Definition 7.12, imply that that every $(0, 0, \tau)$ is routed to $(0, 0, \tau + 1 \bmod 2^t)$ and therefore every (τ, S_τ) is routed to $(\tau + 1 \bmod 2^t, S_{\tau+1 \bmod 2^t})$. Thus we have obtained that $G_M^{\vec{S}}$ is indeed a computation graph.

Next, we show that $G_M^{\vec{S}}$ is valid, accepting and consistent with $(\mathtt{x}, \mathtt{w})$.

Notice that if $R_M^{\vec{S}}$ is accepting and consistent with $(\mathtt{x}, \mathtt{w})$ then so is $\vec{S}$. Therefore $G_M^{\vec{S}}$ is accepting and consistent with $(\mathtt{x}, \mathtt{w})$ if $R_M^{\vec{S}}$ is. This fulfills the acceptance and consistency requirements as stated in the first two items of Definition 7.4. Next we turn our attention to the validity requirement of Definition 7.4. Indeed, by the first two items in Definition 7.12 we obtain that each node on layer i is routed to exactly one node in layer $i + 1 \bmod L$ and that the configuration and time stamp part

of the node's color remains unchanged throughout the layers. Moreover, those items also enforce that the routing respects the edge relation of the underlaying graph. Next, Item (iii) and Item (v) of Definition 7.12 requires that every $(0,0,\tau)$ is routed to $(0,0,\tau+1 \bmod 2^t)$ and that $S_\tau \rightsquigarrow S_{\tau+1 \bmod 2^t}$. We thus obtain that $E_{\mathsf{t}} = \{((\tau, S_\tau), (\tau+1, S_{\tau+1 \bmod 2^t})) : (\tau, S_\tau), (\tau+1 \bmod 2^t, S_{\tau+1 \bmod 2^t}) \in V \text{ and } S_\tau \rightsquigarrow S_{\tau+1 \bmod 2^t}\}$. This fulfills the requirements posed by Item (i) of the valid part of Definition 7.4.

Next, as in the previous case, the first two items in Definition 7.12 enforce that each node on layer i is routed to exactly one node in layer $i+1 \bmod L$, that the configuration and time stamp parts of the nodes color remain unchanged throughout the layers and that the routing respects the edge relation of the underlying graph. In addition, any routing done in Item (iv) of Definition 7.12 induces a permutation on the nodes of the form $(1,0,\tau)$ in $G_M^{\vec{S}}$. Furthermore, we require that each configuration that does not contain a pload or a store instruction will be routed to itself. Thus we obtain that the routing also induces a permutation on the nodes of the form $(1,0,\tau)$ such that S_τ contains a pload or a store instruction. This fulfills the requirement of Item (iii) of Definition 7.4. Item (iv) and Item (v) of Definition 7.12 requires that if $(1,0,\tau')$ is routed to $(1,0,\tau)$ and both S_τ and $S_{\tau'}$ contain a pload or a store instructions then $(\tau', S_{\tau'}) \overset{\mathrm{m}}{\prec} (\tau, S_\tau)$. Therefore, for every edge $((\tau, S_\tau), (\tau', S_{\tau'})) \in E_{\mathsf{m}}$ we have that $(\tau', S_{\tau'}) \overset{\mathrm{m}}{\prec} (\tau, S_\tau)$. This fulfills the requirement of Item (ii) of Definition 7.4. □

In light of Claim 7.8 and Claim 7.13, we deduce the following:

Claim 7.14. *Let M be a memory-well-behaved random-access machine, $\vec{S} = (S_0, \ldots, S_{2^t-1})$ a sequence of configurations for M, and $(\mathtt{x}, \mathtt{w})$ two input strings. Then $\vec{S}$ is valid and accepting for M and consistent with $(\mathtt{x}, \mathtt{w})$ if and only if there exists a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$.*

Witness reduction. Note that not only does Claim 7.13 tell us an equivalence between the two decision problems, but its proof tells us how deduce a "witness" too. Specifically, given $G_M^{\vec{S}}$, we can construct a computation routing $R_M^{\vec{S}}$ by deducing the two permutations (for code consistency and memory consistency) by following part **(1)** of the proof. (As for the reverse direction, part **(2)** of the proof gives us that, but we are not interested in "going back".)

7.3 Step 3: From (double) De Bruijn graphs to succinct GCPs

We are now ready to discuss sGCP. We note that determining whether a computation routing is valid or not (see Definition 7.12) involves checking different constraints that depend on the vertex (this is what distinguishes constraints (i) through (v) of Definition 7.12), the color of the vertex, and the colors of the neighbors of the vertex. As there are not many different constraints that need to be enforced, we can "bundle up" these into a universal constraint that needs to be enforced at every vertex of the graph — and this is the essence of the of the satisfiability requirement of a sGCP problem.

We now discuss then how to derive the parameters of the sGCP problem corresponding to a given machine M. Of course, there are more details that need to be addressed than discussed in the previous paragraph; specifically, we need to be explicit about the "cost" of creating all the necessary objects, and we need to discuss how consistency with the input string $\mathtt{x}$ is achieved.

Thus, using using the template provided in Definition 6.15, we present the following construction:

Construction 7.15. Fix a random-access machine

$$M = \langle w, k, \mathbb{A}, \mathbb{C} \rangle \ .$$

Construct a choice of parameters

$$\mathsf{par}_{\text{sGCP}} = \Big((\mathsf{c}_{\mathbf{V}}, \mathsf{t}_{\mathbf{V}}, \mathbf{V}), (\alpha_{\mathbf{\Gamma}}, \mathsf{t}_{\mathbf{\Gamma}}, \mathbf{\Gamma}), (\mathsf{c}_{\mathbf{C}}, \mathbf{C}), (\mathsf{s}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathbf{K}), (\mathsf{t}_{\mathbf{W}}, \mathbf{W}), (\mathsf{t}_{\mathbf{F}}, \mathbf{F}) \Big)$$

for sGCP as follows:

1. **Constructing Parameter 1.** Consider the functions $\mathsf{c}_{\mathbf{V}}$ and $\mathsf{t}_{\mathbf{V}}$ associated with the family $\mathbf{V}$ defined below

   (a) define the (proper) cardinality function $\mathsf{c}_{\mathbf{V}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\mathsf{c}_{\mathbf{V}}(t) = 1 + \lceil \log(4t - 1) \rceil + t \ ,$$

   (b) define the (proper) time function $\mathsf{t}_{\mathbf{V}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\mathsf{t}_{\mathbf{V}}(t) = O(t).$$

2. **Constructing Parameter 2.** Define the family $\mathbf{V} = \{V_t\}_t$ and the algorithm FIND$\mathbf{V}$ as follows:

   (a) V_t is the vertex set of DDB$(t, 4t - 1)$.

   (b) The existence of a suitable FIND$\mathbf{V}$ algorithm easily follows from Definition 7.9.

3. **Constructing Parameter 3.** Consider the functions $\alpha_{\mathbf{\Gamma}}$ and $\mathsf{t}_{\mathbf{\Gamma}}$ associated with the family $\mathbf{\Gamma}$ defined below

   (a) Define the (proper) regularity function $\alpha_{\mathbf{\Gamma}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\alpha_{\mathbf{\Gamma}}(t) := 4 \ ,$$

   (b) Define the (proper) time function $\mathsf{t}_{\mathbf{V}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\mathsf{t}_{\mathbf{V}}(t) = O(t).$$

4. **Constructing Parameter 4.** Define the family $\mathbf{\Gamma} = \{\vec{\Gamma}_t\}_{t \in \mathbb{N}}$ and the algorithm FIND$\mathbf{\Gamma}$ as follows:

   (a) $\vec{\Gamma}_t = (\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4)$ where Γ_1 is the identify and Γ_2, Γ_3 and Γ_4 are the neighbor functions of a DDB$(t, 4t - 1)$,

   (b) The existence of a suitable FIND$\mathbf{\Gamma}$ algorithm easily follows from Definition 7.9.

   (Recall Definition 7.9, the definition of a double extended De Bruijn graph.) Note that indeed each G_t is $\alpha_{\mathbf{\Gamma}}(t)$-regular. Intuitively, G_t is the graph for a computation routing (Definition 7.11) of a computation of length at most 2^t.

5. **Constructing Parameter 5.** Define the cardinality function $\mathsf{c}_{\mathbf{C}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\mathsf{c}_{\mathbf{C}}(t) := 1 + t + (1 + k)w \ .$$

6. **Constructing Parameter 6.** Define the finite color set family $\mathbf{C} = \{C_t\}_{t\in\mathbb{N}}$ by

$$C_t = \{0,1\}^{\mathsf{c}_{\mathbf{C}}(t)} \ .$$

We have set C_t to be the set of colors of a computation routing. Indeed, recall from Definition 7.11 that a coloring of a computation routing provides for each vertex a bit, a timestamp, and a configuration; thus $\mathsf{c}_{\mathbf{C}}(t) = 1 + t + (1+k)w$, since a timestamp has length t and a configuration has length $(1+k)w$.

7. **Constructing Parameter 7.** Consider the functions $\mathsf{s}_{\mathbf{K}}$ and $\mathsf{t}_{\mathbf{K}}$ associated with the family $\mathbf{K}$ defined below

   (a) Define the (proper) size function $\mathsf{s}_{\mathbf{K}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\mathsf{s}_{\mathbf{K}}(t) := O(t) \ ,$$

   (b) Define the (proper) time function $\mathsf{t}_{\mathbf{K}} \colon \mathbb{N} \to \mathbb{N}$ by

   $$\mathsf{t}_{\mathbf{K}}(t) = O(t).$$

8. **Constructing Parameter 8.** Define the family $\mathbf{K} = \{K_t \colon T_t \times C_t^{\alpha_{\mathbf{\Gamma}}(t)} \to \{0,1\}\}_{t\in\mathbb{N}}$ to be the function

   $$K_t\big(\theta, c_1, \ldots, c_{\alpha_{\mathbf{\Gamma}}(t)}\big) \tag{3}$$

   that interprets the vertex type θ as a vertex $v \in V_t$, and verifies depending on v the appropriate constraint(s) in the first and third bullet of Definition 7.12.

   As for a circuit computing K_t, see Section B.2, where we give explicit circuits.

9. **Constructing Parameter 9.** Consider the function $\mathsf{t}_{\mathbf{W}}$ associated with the family $\mathbf{W}$ defined below

   $$\text{a time function } \mathsf{t}_{\mathbf{W}} \colon \mathbb{N} \to \mathbb{N} \colon \ \mathsf{t}_{\mathbf{W}}(t) := O(t) \ .$$

10. **Constructing Parameter 10.** Define the vertex subset family $\mathbf{W} = \{W_t\}_{t\in\mathbb{N}}$ by

    $$W_t := \big\{(0, 3i+2, 0)\big\}_{0\le i\le 2^{t-1}-1} \ .$$

    In other words, we set W_t to be the vertices in the 0-th column of the 0-th extended De Bruijn graph in $G_t = \mathrm{DDB}(t, 4t-1)$ (except the 0-th one) whose numbering is of the form $3i+2$. This is so because ultimately we will be interested in random-access machines that are memory-well-behaved (see Definition 7.7), so that the first memory cell is accessed in the first time step, and input-well-behaved, so that each input symbol is read, stored in memory and a counter is incremented as preperation towards reading the next input symbol. This process is done on the whole input (see Definition 7.16); in particular, in order to "observe" the input in the configurations we examine the first sufficiently-many configurations whose numbering is of the form $3i+2$ (after the initial one).

    Therefore, we can simply set $(0, 3i+2, 0) := \textsc{Find}\mathbf{W}(1^t, i)$, and thus $\mathsf{t}_{\mathbf{W}}(t)$ has the above claimed value.

11. **Constructing Parameter 11.** Consider the function $\mathtt{t}_{\mathbf{F}}$ associated with the family $\mathbf{F}$ defined below

$$\text{a time function } \mathtt{t}_{\mathbf{F}} \colon \mathbb{N} \to \mathbb{N} \colon \mathtt{t}_{\mathbf{F}}(t) := O(t + (1+k)w) \ .$$

12. **Constructing Parameter 12.** Define the extraction function family $\mathbf{F} = \{F_t \colon \{0,1\}^* \to \{0,1\}\}_{t \in \mathbb{N}}$ by

$$F_t(\mathtt{x}, c_1, \ldots, c_{|\mathtt{x}|}) := \text{“0 if and only if } \mathtt{x} = \rho_1 \cdots \rho_{|\mathtt{x}|}\text{”} \ ,$$

where $c_i = (s_i, \tau_i, S_i)$, $S_i = [\mathsf{pc}_i, r_0^i, \ldots, r_{k-1}^i]$, and $\rho_i = r_j^i$ if the instruction pointed to by $\mathsf{pc} - 1 \bmod 2^t$ is a read from the input tape into register j (else, ρ_i is the empty string). Note that indeed the size of ρ is w bits. In other words, the extraction function F_t, on input $\mathtt{x}$ and all the reads from tape A in order, checks that the concatenation of the reads is equal to $\mathtt{x}$.

Furthermore, $\textsc{Comp}\mathbf{F}(1^t, c)$ is trivial because F_t is merely a comparison between two strings.

As we need to check input consistency, we shall restrict (without loss of generality) our attention to random-access machines that read the whole input and store it into memory, without ever accessing the input tape again. This ensures that the input bits appear at “canonical” times of the computation.

Definition 7.16. *Let M be a random-access machine. We say that M is* **input-well-behaved** *if M, starting from the second step, reads all of the input on A and stores it into memory, and then never reads from the input again.*

In sum, a random-access machine is well-behaved if it is both memory-well-behaved and input-well-behaved.

Definition 7.17. *Let M be a random-access machine. We say that M is* well-behaved *if it is both memory-well-behaved and input-well-behaved.*

We now give the equivalence between finding computation routings and membership in $\text{sGCP}(\mathsf{par}_{\text{sGCP}})$ for parameters $\mathsf{par}_{\text{sGCP}}$ constructed as above.

Claim 7.18. *Let M be a well-behaved random-access machine, $\mathtt{x}$ an input string, and $t \in \mathbb{N}$. There exists another input string $\mathtt{w}$, a sequence of configurations $\vec{S} = (S_0, \ldots, S_{T-1})$ for M with $T \leq 2^t$, and a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$ if and only if $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$, where $\mathsf{par}_{\text{sGCP}}$ are constructed from M following Construction 7.15.*

Proof. We prove in **(1)** one direction and in **(2)** the other direction.

(1) Assume there exists another input string $\mathtt{w}$, a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M, and a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$. We need to prove prove that $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$.

We now argue that $R_M^{\vec{S}}$ itself is in fact a witness for $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$.

Following Definition 6.15, we show that the following two conditions hold:

- *Satisfiability of constraints.* The routing $R_M^{\vec{S}}$ satisfies the routing constraints induced by K_t, i.e., for every $v \in V_t$,

$$K_t\Big(v, \big(R_M^{\vec{S}} \circ \Gamma_{t,1}\big)(v), \ldots, \big(R_M^{\vec{S}} \circ \Gamma_{t,\alpha_{\mathbf{\Gamma}}(t)}\big)(v)\Big) = 0 \ , \tag{4}$$

where, for $i = 1, \ldots, \alpha_{\mathbf{\Gamma}}(t)$, $\Gamma_{t,i}$ is the i-th neighbor function of $G_t = \text{DDB}(t, 4t-1)$. And, indeed, since $R_M^{\vec{S}}$ is a valid computation routing by Item 8 of Construction 7.15 the above equation holds. Thus the coloring $C := R_M^{\vec{S}}$ satisfies the coloring constraints induced by K_t and M_t, as required by Item (i) of Definition 6.15.

- *Consistency with the instance.* The routing $R_M^{\vec{S}}$ is consistent with the instance $(\mathtt{x}, 1^t)$, i.e., for every index $i \in \{1, \ldots, |W_t|\}$, letting v_i be the i-th vertex in W_t, $F_t(\mathtt{x}, R_M^{\vec{S}}(v_1), \ldots, R_M^{\vec{S}}(v_{|\mathtt{x}|})) = 0$.

  Because M is well-behaved, all the input $\mathtt{x}$ will be read into memory at the start of the computation right after accessing the first memory cell, so instruction 2 through $3|\mathtt{x}| + 2$ will consist of a read_A followed by a store and an incrementation of the value of some register so that the input can be stored at the next available memory cell. This process is repeated $|\mathtt{x}|$ times. And, indeed, we have defined the i-th vertex of W_t to be the $(3i + 2)$-th vertex in the first column of one De Bruijn graph in V_t, and F_t to verify the appropriate read_A information from its configuration. Thus, once again the same coloring $C := R_M^{\vec{S}}$ is consistent with the instance $(\mathtt{x}, 1^t)$, as required by Item (ii) of Definition 6.15.

(2) Suppose that $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$, and let C be a coloring witnessing this. We need to prove that there exist another input string $\mathtt{w}$, a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M, and a computation routing $R_M^{\vec{S}}$ for M with respect to $\vec{S}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$.

For $i = 0, \ldots, 2^t - 1$, if $C(0, i, 0) = (0, i, S_i')$, define $S_i := S_i'$ and let $S_i = [\mathsf{pc}_i, r_0^i, \ldots, r_{k-1}^i]$. By Item 8 of Definition 7.15 and by the construction of $\vec{S}$, we know that C is a valid and accepting computation routing for M with respect to $\vec{S}$.

Regarding the consistency requirement, consider the string $\mathtt{w} = \sigma_1 \cdots \sigma_{2^t-1}$ where for any $i \in \{0, \cdots, 2^t - 1\}$ it holds that $\sigma_i = r_j$ if $\mathsf{pc}_i - 1$ points to a read_B j instruction and $\sigma_i' = \varepsilon$ otherwise. Notice that by construction it holds that C is consistent with $\mathtt{w}$. As for consistency with the input $\mathtt{x}$, since C is a witness to $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$ it must be the case that C is consistent with $\mathtt{x}$ also.

Thus we have obtained that C is a computation routing with respect to $\vec{S}$ that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$. □

In light of Claim 7.14 and Claim 7.18, we deduce the following:

Claim 7.19. *Let M be a well-behaved random-access machine, $\mathtt{x}$ an input string, and $t \in \mathbb{N}$. There exists another input string $\mathtt{w}$, and a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$ if and only if $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$, where $\mathsf{par}_{\text{sGCP}}$ are constructed from M following Construction 7.15.*

7.4 Step 4: The Levin reduction

We show that the parameter conversion discussed in Section 7.4 yields a Levin reduction (according to Definition 7) from BH_{RAM} to sGCP with respect to any class of random-access machines $\mathcal{P}_{\text{RAM}}$ (specifically, any random-access machine as defined in Section 6.4.2).

More precisely, consider the following definitions:

- Define $F_{\mathsf{p}} \colon \{0,1\}^* \to \{0,1\}^*$ to be the function that, on input a choice of random-access machine $M \in \mathcal{P}_{\text{RAM}}$, first modifies M to ensure it is well-behaved (see Definition 7.17) and then performs the parameter conversion described in Construction 7.15. More precisely, F_{p} works as follows:

$$M = \langle w, k, \mathbb{A}, \mathbb{C} \rangle \quad \xmapsto{F_{\mathsf{p}}} \quad \mathsf{par}_{\text{sGCP}} = \begin{pmatrix} (\mathsf{c}_{\mathbf{V}}, \mathsf{t}_{\mathbf{V}}, \text{FIND}\mathbf{V}), \\ (\alpha_{\mathbf{\Gamma}}, \mathsf{t}_{\mathbf{\Gamma}}, \text{FIND}\mathbf{\Gamma}), \\ \mathsf{c}_{\mathbf{C}}, \\ (\mathsf{s}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \text{FIND}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \text{FIND}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \text{COMP}\mathbf{F}) \end{pmatrix},$$

where the mapping is done by following the definitions of the various new complexity functions and algorithms (for sGCP) based on the old ones (for M).

- Define $F_{\mathsf{w}}\colon \{0,1\}^* \to \{0,1\}^*$ as follows: for every $M \in \mathcal{P}_{\mathrm{RAM}}$, $t \in \mathbb{N}$, $\mathtt{w} \in \{0,1\}^*$, and $\vec{S}$,

  $F_{\mathsf{w}}(M, 1^{2^t}, (\mathtt{w}, \vec{S})) \equiv$
  1. Route $\vec{S}$ on the graph $G_t := \mathrm{DDB}(\kappa, L)$, where $\kappa = t$ and $L = 4\kappa - 1$, by computing the routing for the code-consistency permutation and the memory-consistency permutation, and let C be the resulting coloring.
  2. Output C.

We prove the following theorem:

Theorem 7.20. *The pair of functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a Levin reduction from $\mathrm{BH}_{\mathsf{RAM}}$ to* sGCP *with respect to any class of random-access machines $\mathcal{P}_{\mathrm{RAM}}$.*

We divide the proof of Theorem 7.20 into three claims:
- in Claim 7.21, we explain why both F_{p} and F_{w} are polynomial-time computable;
- in Claim 7.22, we show the "completeness" and "soundness" of F_{p}; and
- in Claim 7.23, we show that F_{w} produces good witnesses.

Claim 7.21. *The functions F_{p} and F_{w} are polynomial-time computable.*

Proof. The efficiency of F_{p} easily follows by inspection of how the parameters are converted in Construction 7.15. (Essentially, all the new functions and algorithms are "easy combinations" of previous functions and algorithms, and thus not hard to write down.) Also note that ensuring that M is well-behaved is tantamount to small changes to the code of M. The efficiency of F_{w} easily follows from the fact that it can run in time that is polynomial in 1^{2^t}, which is plenty of time for computing the routing of $\vec{S}$ on G_t. (Indeed, recall Claim 6.5.) □

Claim 7.22. *For every choice of machine $M \in \mathcal{P}_{\mathrm{RAM}}$ and for every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$, $(\mathtt{x}, 1^t) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ if and only if $(\mathtt{x}, 1^t) \in \mathrm{sGCP}(F_{\mathsf{p}}(M))$.*

Proof. Follows directly from Claim 7.19 and Definition 6.14 (which says that $(\mathtt{x}, 1^t) \in \mathrm{BH}_{\mathsf{RAM}}(M)$ if and only if there is another input string $\mathtt{w}$, and a sequence of configurations $\vec{S} = (S_0, \ldots, S_{2^t-1})$ for M that is valid, accepting, and consistent with $(\mathtt{x}, \mathtt{w})$). □

Claim 7.23. *For every choice of machine $M \in \mathcal{P}_{\mathrm{RAM}}$ and for every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$, if $(\mathtt{w}, \vec{S})$ is a witness to "$(\mathtt{x}, 1^t) \in \mathrm{BH}_{\mathsf{RAM}}(M)$" then $F_{\mathsf{w}}(M, 1^{2^t}, (\mathtt{w}, \vec{S}))$ is a witness to "$(\mathtt{x}, 1^t) \in \mathrm{sGCP}(F_{\mathsf{p}}(M))$".*

Proof. The claim follows immediately from the fact that in the proof of the "completeness" direction of the statement from Claim 7.22 (or, more precisely, from the proof of Claim 7.18, which implies Claim 7.19, and in turn Claim 7.22), we have constructed, starting from a valid pair $(\mathtt{w}, \vec{S})$ and according to F_{p}, a valid coloring C for $(\mathtt{x}, 1^t)$: by routing $\vec{S}$ on G_t. □

8 From sGCP To sACSP

In Section 4 we discussed a high-level strategy for proving Theorem 2 in two steps (respectively discussed in Section 4.1 and Section 4.2). The goal of this section is to prove in detail the second step. In Section 7 we have already obtained a sGCP problem on (double) extended De Bruijn graphs. Our goal is to arithmetize it.

Given the "template" Definition 6.3, we give for convenience the following specialized definition for reducing from sGCP to sACSP:

Definition 8.1. *Fix a class of parameters $\mathcal{P}_{\text{sGCP}}$ for* sGCP*. We say that a pair of polynomial-time-computable functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a* Levin reduction from sGCP to sACSP with respect to $\mathcal{P}_{\text{sGCP}}$ *if the following three conditions are satisfied for every choice of parameters $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{sGCP}}$:*

1. *$F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}})$ is a choice of parameters for* sACSP*.*
2. *For every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$,*

$$(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}}) \text{ if and only if } (\mathtt{x}, 1^t) \in \text{sACSP}\big(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}})\big).$$

3. *For every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$,*

$$\begin{gathered}\text{if the coloring } C \text{ is a witness to "}(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})\text{",} \\ \text{then } F_{\mathsf{w}}(\mathsf{par}_{\text{sGCP}}, 1^{2^t}, C) \text{ is a witness to "}(\mathtt{x}, 1^t) \in \text{sACSP}\big(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}})\big)\text{".}\end{gathered}$$

The two functions F_{p} and F_{w} are respectively called as the "parameter reduction" and the "witness reduction".

Remark 8.2. We believe that the Levin reductions discussed in this section, as well as the arithmetization results developed in Section C, will provide a valuable "toolkit" for arithmetizing a variety of structured graphs (including butterfly networks and Beneš networks) for the purposes of engineering other Levin reductions from succinct constraint satisfaction problems on such graphs. Having this flexibility could allow for more direct reductions for other succinct models of computation (e.g., arithmetic formulas, finite automata), and thus ultimately lead to more efficient PCPs "specialized" for such models of computation.

We give here the definition of an affine graph used throughout this section:

Definition 8.3. *Let $\mathbb{F}$ be a field and $\mathcal{A}$ a set of affine functions over $\mathbb{F}$. The* affine graph $\text{AFF}(\mathbb{F}, \mathcal{A})$ *is a directed graph with vertex set $\mathbb{F}$ where each element $\alpha(\mathsf{x}) \in \mathbb{F}$ has a directed edge to the element $N(\alpha(\mathsf{x}))$ for each affine function $N \in \mathcal{A}$.*

For convenience, we provide again the definition of a double extended De Bruijn graph (which we had introduced in Definition 7.9). Recall that a double extended De Bruijn graph is simply two extended De Bruijn graphs "side by side", connected with bi-directional edges at corresponding vertices.

Definition 8.4. *Let κ and L be two positive integers. The (κ, L)* **double extended De Bruijn graph***, denoted $\text{DDB}(\kappa, L)$, is a 3-regular directed graph consisting of the Cartesian graph product of the directed two-cycle and $\text{DB}(\kappa, L)$. In other words, the vertex set V of $\text{DDB}(\kappa, L)$ consists of vertices $v = (b, i, w)$, where $b \in \{0,1\}$, $i \in \{0, \ldots, L-1\}$, and $w \in \{0,1\}^\kappa$, and the edge set E is induced by the following three neighbor functions:*

- $\Gamma_1\big((b,i,w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w))$, *i.e., one kind of De Bruijn edges;*
- $\Gamma_2\big((b,i,w)\big) = (b, i+1 \bmod L, \mathrm{sr}(w) \oplus e_1)$, *i.e., the other kind of De Bruijn edges; and*
- $\Gamma_3\big((b,i,w)\big) = (1 \oplus b, i, w)$, *i.e., edges between corresponding vertices of the two De Bruijn graphs.*

Remark 8.5. One could consider more general graph products of extended De Bruijn graphs, e.g., the Cartesian graph product with directed cycles larger than 2. These more complex structured graphs may very well be useful in Levin reductions from other models of computation. See Remark 8.13 for a short discussion of how our arithmetization techniques extend to these more general products.

Remark 8.6. Our arithmetization techniques and conversion of parameters also yield a Levin reduction from succinct graph coloring problems on *single* extended De Bruijn graphs to succinct algebraic constraint satisfaction problems. (A non-uniform version of this reduction was considered by [BSS08].) We happen not to use single extended De Bruijn graphs in this work, as a direct reduction from random-access machines seems to require the use of two routing networks; nonetheless, single extended De Bruijn graphs are quite expressive, and may indeed be useful in other Levin reductions.

Remark 8.7. We note that there are other "arithmetization" paths that could be used, say, starting from sGCPs on double extended De Bruijn graphs. For example, one could consider *multivariate* variants of succinct algebraic constraint satisfaction problems. Alternatively, arithmetizing separately different parts of the graph (e.g., each extended De Bruijn graph in the double extended De Bruijn graph), to obtain a univariate succinct algebraic constraint satisfaction problem with multiple polynomials as witnesses, and consistency checks between them. These other arithmetization paths would require a new class of succinct algebraic constraint satisfaction problems, thus requiring a new definition similar in spirit to Definition 6.18 (and ultimately a new construction of PCPs for it, which may or may not follow easily from the PCPs of [BSCGT12] for Definition 6.18).

Remark 8.8. The main theorem of this section, Theorem 8.16, can be thought of as a generalization of [BSGH$^+$05, Theorem 6.2] (and our proof easily implies an explicit proof to [BSGH$^+$05, Theorem 6.2], which was only stated without proof); the arithmetization needed in the proof is inspired by [BSS08, proof of Theorem 3.7], but requires a lot more care and involves more details.

This section is organized as follows: we first provide some algebraic lemmas establishing appropriate embeddings of double extended De Bruijn graphs into affine graphs, along with other properties (Section 8.1); then give the conversion of parameters (Section 8.2), and finally prove that the conversion of parameters yields a Levin reduction (Section 8.3).

8.1 An embedding and some lemmas for double extended De Bruijn graphs

We first prove several lemmas about (*single*) extended De Bruijn graphs, and then build on these to obtain analogous results for *double* extended De Bruijn graphs.

The single De Bruijn case. First, we show how to "arithmetize" (single) extended De Bruijn graphs by embedding them into appropriate affine graphs. More concretely, given a graph $\mathrm{DB}(\kappa, 2^\ell - 1) = (V, E)$ (where we take for convenience the number of columns to be a power of 2), we need to come up with a function $\widetilde{\Phi}_{f,\kappa,\ell}$ that maps vertices in V to field elements of $\mathbb{F}_{2^f}$ (for a sufficiently large f) and edges in E to edges of an affine graph over the elements of $\mathbb{F}_{2^f}$ induced by a small set of affine functions $\widetilde{\mathcal{A}}_{f,\kappa,\ell}$; we will call such a function a *graph embedding* (and the reason that it is not

a graph isomorphism is that generally the affine graph will contain many more vertices and edges than those induced by the mapping from $\mathrm{DB}(\kappa, 2^\ell - 1)$).

Recall from Definition 6.4 that a vertex $v \in V$ is labeled as (i, w), where $i \in \{0, \ldots, 2^\ell - 2\}$ and $w \in \{0,1\}^\kappa$. To encode the label as a field element in $\mathbb{F}_{2^f}$: we will encode the string w as the high order coefficients of (the polynomial representaiton of) a field element; the low-order coefficients of (the polynomial representaiton of) the field element will be reserved for the encoding of i, which is achieved by creating an "artificial" cyclic group of size $2^\ell - 1$ (all of whose elements are polynomials of degree less than ℓ and thus can be stored as low-order coefficients) and letting i be mapped to the i-th element of this cyclic group. The encoding of the label (i, w) was designed to be "friendly" to affine functions, in the sense that all the images of vertices that are connected by an edge in E can be related by a small set of affine functions.

More precisely, we prove the following lemma:

Lemma 8.9 (Arithmetization of Extended De Bruijn Graphs)**.** *Fix three positive integers f, κ, and ℓ with $f \geq \ell+\kappa+1$. Let I be the irreducible polynomial of degree f over $\mathbb{F}_2$ output by* FindIrrPoly(1^f)*, and let $\mathbb{F}_{2^f} := \mathbb{F}_2(\mathsf{x})$ where x is a root of I. Let $\Xi(x) \in \mathbb{F}_2[x]$ be the primitive polynomial of degree ℓ output by* FindPrimPoly(1^ℓ) *and let $\xi_0(x), \ldots, \xi_{(2^\ell-2)}(x) \in \mathbb{F}_2[x]$ be the $2^\ell - 1$ distinct polynomials of degree less than ℓ implied by invoking Claim C.5 with $\Xi(x)$.*

Define the two functions $h_{f,\kappa,\ell} \colon \{0, \ldots, 2^\ell - 2\} \to \mathbb{F}_{2^f}$ and $g_{f,\kappa,\ell} \colon \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ by

$$h_{f,\kappa,\ell}(i) := \xi_i(\mathsf{x}) \quad \text{and} \quad g_{f,\kappa,\ell}(w) := \mathsf{x}^{\ell-1} \cdot \left(\sum_{j=1}^{\kappa} w_j \mathsf{x}^j \right) \ ,$$

and define the function $\widetilde{\Phi}_{f,\kappa,\ell} \colon \{0, \ldots, 2^\ell - 2\} \times \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ by

$$\widetilde{\Phi}_{f,\kappa,\ell}(i, w) := h_{f,\kappa,\ell}(i) + g_{f,\kappa,\ell}(w) \ .$$

Then, $\widetilde{\Phi}_{f,\kappa,\ell}$ is an injective isomorphism from the De Bruijn graph $\mathrm{DB}(\kappa, 2^\ell - 1) = (V, E)$ into the affine graph $\mathrm{AFF}(\mathbb{F}_{2^f}, \widetilde{\mathcal{A}}_{f,\kappa,\ell})$, where

$$\widetilde{\mathcal{A}}_{f,\kappa,\ell} = \left\{ N_\sigma(z) = \mathsf{x} \cdot z + \sigma_1 \Xi(\mathsf{x}) + \sigma_2 \mathsf{x}^\ell + \sigma_3 \mathsf{x}^{\ell+\kappa} \in \mathbb{F}_{2^f}[z] \right\}_{\sigma \in \{0,1\}^3} \ .$$

Note that $|\widetilde{\mathcal{A}}_{f,\kappa,\ell}| = 8$.

Proof. Observe that $\deg(h_{f,\kappa,\ell}(i)) = \deg(\xi_i(\mathsf{x})) < \ell$ for every $i \in \{0, \ldots, 2^\ell - 2\}$ and $\deg(g_{f,\kappa,\ell}(w)) = \deg(\mathsf{x}^{\ell-1} \cdot (\sum_{j=1}^{\kappa} w_j \mathsf{x}^j)) \geq \ell$ for every not-all-zeros κ-bit string w. Hence, the function $\widetilde{\Phi}_{f,\kappa,\ell}$ is injective if both $h_{f,\kappa,\ell}$ and $g_{f,\kappa,\ell}$ are injective; injectivity of the second map is clear, and injectivity of the first map follows from Claim C.4 (which tells us that $i \neq i'$ implies that $h_{f,\kappa,\ell}(i) = \xi_i(\mathsf{x}) \neq \xi_{i'}(\mathsf{x}) = h_{f,\kappa,\ell}(i')$). Next, to prove that $\widetilde{\Phi}_{f,\kappa,\ell}$ is a graph isomorphism, we need to show that if (v, v') is an edge in $\mathrm{DB}(\kappa, 2^\ell - 1)$ then $(\widetilde{\Phi}_{f,\kappa,\ell}(v), \widetilde{\Phi}_{f,\kappa,\ell}(v'))$ is an edge in $\mathrm{AFF}(\mathbb{F}_{2^f}, \widetilde{\mathcal{A}}_{f,\kappa,\ell})$.

We first derive some relations. Recalling the definition of $h_{f,\kappa,\ell}$, from Corollary C.5 we deduce that

$$h_{f,\kappa,\ell}(i + 1 \bmod (2^\ell - 1)) = \xi_{(i+1 \bmod (2^\ell-1))}(\mathsf{x}) = \mathsf{x} \cdot \xi_i(\mathsf{x}) + Q(\mathsf{x}) \cdot \Xi(\mathsf{x}) = \mathsf{x} \cdot h_{f,\kappa,\ell}(i) + Q(\mathsf{x}) \cdot \Xi(\mathsf{x}) \ ,$$

where $Q(\mathsf{x})$ is the polynomial quotient (of degree less than 1, i.e., $Q(\mathsf{x}) = 0$ or $Q(\mathsf{x}) = 1$) when dividing $\mathsf{x} \cdot h_{f,\kappa,\ell}(i)$ by $\Xi(\mathsf{x})$. (Note that, again as above, in the computation, we have used the fact that the field $\mathbb{F}_{2^f}$ is "large enough".) Therefore,

$$h_{f,\kappa,\ell}(i + 1 \bmod (2^\ell - 1)) = \begin{cases} \mathsf{x} \cdot h_{f,\kappa,\ell}(i) = N_{000}\big(h_{f,\kappa,\ell}(i)\big) & \text{if } \deg\big(h_{f,\kappa,\ell}(i)\big) < \ell - 1 \\ \mathsf{x} \cdot h_{f,\kappa,\ell}(i) + \Xi(\mathsf{x}) = N_{100}\big(h_{f,\kappa,\ell}(i)\big) & \text{if } \deg\big(h_{f,\kappa,\ell}(i)\big) = \ell - 1 \end{cases} \ .$$

Similarly, recalling the definition of $g_{f,\kappa,\ell}$ and $\widetilde{\mathcal{A}}_{f,\kappa,\ell}$, we see that

$$g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\big) = \begin{cases} \mathsf{x}\cdot g_{f,\kappa,\ell}(w) = N_{000}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 0 \\ \mathsf{x}\cdot g_{f,\kappa,\ell}(w) + \mathsf{x}^\ell + \mathsf{x}^{\ell+\kappa} = N_{011}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 1 \end{cases}$$

and

$$g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\oplus e_1\big) = \begin{cases} \mathsf{x}\cdot g_{f,\kappa,\ell}(w) + \mathsf{x}^\ell = N_{010}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 0 \\ \mathsf{x}\cdot g_{f,\kappa,\ell}(w) + \mathsf{x}^{\ell+\kappa} = N_{001}\big(g_{f,\kappa,\ell}(w)\big) & \text{if } w_\kappa = 1 \end{cases} .$$

(Note that, in the above two equations, we have used the fact that the field $\mathbb{F}_{2^f}$ is "large enough". Specifically, by the choice $f \geq \ell+\kappa+1$, since all the polynomials in the computation have degree less than $\ell+\kappa+1$, we do not have to worry about reducing modulo $I(\mathsf{x})$.)

Thus, parsing v as (i,w), if (v,v') is an edge in $\mathrm{DB}(\kappa, 2^\ell-1)$, then either $v' = \big((i+1 \bmod (2^\ell-1)), \mathrm{sr}(w)\big)$ or $v' = \big((i+1 \bmod (2^\ell-1)), \mathrm{sr}(w)\oplus e_1\big)$, so that either $\widetilde{\Phi}_{f,\kappa,\ell}(v') = h_{f,\kappa,\ell}\big((i+1 \bmod (2^\ell-1))\big) + g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\big)$ or $\widetilde{\Phi}_{f,\kappa,\ell}(v') = h_{f,\kappa,\ell}\big((i+1 \bmod (2^\ell-1))\big) + g_{f,\kappa,\ell}\big(\mathrm{sr}(w)\oplus e_1\big)$. Either way, it is possible to write $\widetilde{\Phi}_{f,\kappa,\ell}(v') = N_\sigma(v)$ for some $\sigma \in \{0,1\}^3$ by referring to the computations above in order to deduce σ_h for the $h_{f,\kappa,\ell}$ term and σ_g for the $g_{f,\kappa,\ell}$ term, and setting $\sigma = \sigma_h \oplus \sigma_g$. □

We now discuss the computational properties of the (single) extended De Bruijn graph embedding $\widetilde{\Phi}_{f,\kappa,\ell}$ defined in Lemma 8.9. First, we note that it is efficiently computable, both via an algorithm and a certain polynomial:

Lemma 8.10. *Define* $\mathtt{t}_{\widetilde{\mathbf{\Phi}}}(f,\kappa,\ell) := \mathtt{t}_{\mathrm{PRIM}}(\ell) + \ell^2\log\ell + f$. *There exists a* $\mathtt{t}_{\widetilde{\mathbf{\Phi}}}$*-time algorithm* $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ *such that* $\widetilde{\Phi}_{f,\kappa,\ell}\big((i,w)\big) = \textsc{Comp}\widetilde{\mathbf{\Phi}}\big(1^f, 1^\kappa, 1^\ell, (i,w)\big)$, *for every three positive integers* f, κ, *and* ℓ *as in Lemma 8.9, and for every* $i \in \{0,\ldots,2^\ell-2\}$ *and* $w \in \{0,1\}^\kappa$.

Moreover, there exists a multilinear polynomial $P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}} \colon \mathbb{F}_{2^f}^{\ell+\kappa} \to \mathbb{F}_{2^f}$ *such that*

$$P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}(i,w) = \textsc{Comp}\widetilde{\mathbf{\Phi}}\big(1^f, 1^\kappa, 1^\ell, (i,w)\big)$$

for every $i \in \{0,\ldots,2^\ell-2\}$ *and* $w \in \{0,1\}^\kappa$*; moreover, this polynomial can be found and evaluated in time* $O(w+2^\ell)$.

Proof. The injective graph isomorphism $\widetilde{\Phi}_{f,\kappa,\ell} \colon \{0,\ldots,2^\ell-1\} \times \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ can be efficiently computed by the following algorithm:

$\textsc{Comp}\widetilde{\mathbf{\Phi}}\big(1^f, 1^\kappa, 1^\ell, (i,w)\big) \equiv$

1. Compute $\Xi := \textsc{FindPrimPoly}\big(1^\ell\big)$.
2. Compute $\alpha(\mathsf{x}) := \mathsf{x}^i \bmod \Xi(\mathsf{x})$.
3. Compute $\beta(\mathsf{x}) := \sum_{j=1}^{\kappa} w_j \mathsf{x}^{j+\ell-1}$.
4. Output $\alpha(\mathsf{x}) + \beta(\mathsf{x})$.

Note that $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ does not have to compute the degree-f irreducible polynomial I over $\mathbb{F}_2$ that defines the field $\mathbb{F}_{2^f}$, because both $\alpha(\mathsf{x})$ and $\beta(\mathsf{x})$, as well as $\alpha(\mathsf{x})+\beta(\mathsf{x})$, all have degree less than f. Moreover, $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ has the claimed complexity parameters.

The polynomial $P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}$ is the straightforward conversion of $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ into a circuit. Specifically, we can take

$$P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}(x_1,\ldots,x_{\ell+\kappa}) := \left(\sum_{i\in\{0,1\}^\ell} h_{f,\kappa,\ell}(i)\prod_{j=1}^{\ell}(x_j-(1-i_j))\right) + \mathsf{x}^{\ell-1}\cdot\left(\sum_{j=1}^{\kappa} x_{\ell+j}\mathsf{x}^j\right) .$$

□

The double De Bruijn case. Next, we build on the "single extended De Bruijn case", and show how to arithmetize double extended De Bruijn graphs (according to Definition 8.4). We will not have to work hard for this arithmetization, as most of the required arithmetization work went in to understanding the single extended De Bruijn graph case. More concretely, the basic intuition for arithmetizing a double extended De Bruijn graph is quite straightforward: we simply arithmetize each of the extended De Bruijn graphs separately, and put them side by side in the same finite field, ensuring that we have additional affine edges in the resulting affine graph to connect between them; the operations of "putting side by side" intuitively consist of doubling the field size, and putting the two affine graphs in two halves of the finite field, thus obtaining an extra "bit" to toggle between the two graphs. (Technically, we actually will not have to double the field size, but it is helpful to think about it this way.)

More precisely, we prove the following lemma:

Lemma 8.11 (Arithmetization of Double Extended De Bruijn Graphs)**.** *Fix three positive integers f, κ, and ℓ with $f \geq \ell + \kappa + 1$. Let I be the irreducible polynomial of degree f over $\mathbb{F}_2$ output by* $\textsc{FindIrrPoly}(1^f)$*, and let $\mathbb{F}_{2^f} := \mathbb{F}_2(\mathsf{x})$ where x is a root of I. Invoke Lemma 8.9 with the three positive integers f, κ, and ℓ to obtain the function $\widetilde{\Phi}_{f,\kappa,\ell}$ and affine edges $\widetilde{\mathcal{A}}_{f,\kappa,\ell}$.*

Define the function $\Phi_{f,\kappa,\ell} \colon \{0,1\} \times \{0,\ldots,2^\ell - 2\} \times \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ by

$$\Phi_{f,\kappa,\ell}(b,i,w) := \widetilde{\Phi}_{f,\kappa,\ell}(i,w) + b\mathsf{x}^{\ell+\kappa} \ .$$

Then, $\Phi_{f,\kappa,\ell}$ is an injective isomorphism from the double extended De Bruijn graph $\mathrm{DDB}(\kappa, 2^\ell) = (V,E)$ into the affine graph $\mathrm{AFF}(\mathbb{F}_{2^f}, \mathcal{A}_{f,\kappa,\ell})$, where

$$\mathcal{A}_{f,\kappa,\ell} = \{N_\times(z) = z + \mathsf{x}^{\kappa+\ell} \in \mathbb{F}_{2^f}[z]\} \cup \widetilde{\mathcal{A}}_{f,\kappa,\ell} \ .$$

Note that $|\mathcal{A}_{f,\kappa,\ell}| = 1 + |\widetilde{\mathcal{A}}_{f,\kappa,\ell}| = 1 + 8 = 9$.

Proof. The function $\Phi_{f,\kappa,\ell}$ is injective because $\widetilde{\Phi}_{f,\kappa,\ell}$ is injective and the degree of an element output by $\widetilde{\Phi}_{f,\kappa,\ell}$ is at most $\ell+\kappa-1$. To see why $\Phi_{f,\kappa,\ell}$ is a graph isomorphism we note that $\widetilde{\Phi}_{f,\kappa,\ell}$ is a graph isomorphism on the 0-th extended De Bruijn graph and $\mathsf{x}^{\ell+\kappa+1} + \widetilde{\Phi}_{f,\kappa,\ell}$ is a graph isomorphism on the extended 1-th De Bruijn graph; bi-directional edges between images of corresponding vertices $(0,i,w)$ and $(1,i,w)$ in the two De Bruijn graphs are indeed affine edges given by the affine function $N_\times$ in $\mathcal{A}_{f,\kappa,\ell}$ (where the "$\times$" symbol is intended to denote "crossing" between the two De Bruijn graphs). □

The double extended De Bruijn graph embedding $\Phi_{f,\kappa,\ell}$ from Lemma 8.11 is efficiently computable:

Lemma 8.12. *Define $\mathtt{t}_{\mathbf{\Phi}}(f,\kappa,\ell) := \mathtt{t}_{\mathrm{PRIM}}(\ell) + \ell^2 \log \ell + f$. There exists a $\mathtt{t}_{\mathbf{\Phi}}$-time algorithm* $\textsc{Comp}\mathbf{\Phi}$ *such that $\Phi_{f,\kappa,\ell}\big((b,i,w)\big) = \textsc{Comp}\mathbf{\Phi}\big(1^f, 1^\kappa, 1^\ell, (b,i,w)\big)$, for every three positive integers f, κ, and ℓ as in Lemma 8.11, and for every $b \in \{0,1\}$, $i \in \{0,\ldots,2^\ell-2\}$, and $w \in \{0,1\}^\kappa$.*

Moreover, there exists a multilinear polynomial $P^{f,\ell,\kappa}_{\mathbf{\Phi}} \colon \mathbb{F}_{2^f}^{1+\ell+\kappa} \to \mathbb{F}_{2^f}$ such that

$$P^{f,\ell,\kappa}_{\mathbf{\Phi}}(b,i,w) = \textsc{Comp}\mathbf{\Phi}\big(1^f, 1^\kappa, 1^\ell, (b,i,w)\big)$$

for every $b \in \{0,1\}$, $i \in \{0,\ldots,2^\ell-2\}$ and $w \in \{0,1\}^\kappa$.

Proof. The injective graph isomorphism $\Phi_{f,\kappa,\ell} \colon \{0,1\} \times \{0,\ldots,2^\ell-2\} \times \{0,1\}^\kappa \to \mathbb{F}_{2^f}$ can be efficiently computed by the following isomorphism:

$\textsc{Comp}\mathbf{\Phi}\big(1^f, 1^\kappa, 1^\ell, (b, i, w)\big) \equiv$

1. Compute $\alpha(\mathsf{x}) := \textsc{Comp}\widetilde{\mathbf{\Phi}}\big(1^f, 1^\kappa, 1^\ell, (i, w)\big)$. (Where $\textsc{Comp}\widetilde{\mathbf{\Phi}}$ comes from Lemma 8.10.)
2. If $b = 0$, output $\alpha(\mathsf{x})$; otherwise, output $\mathsf{x}^{\ell+\kappa} + \alpha(\mathsf{x})$.

The correctness and complexity guarantees of $\textsc{Comp}\mathbf{\Phi}$ immediately follow from those of $\textsc{Comp}\widetilde{\mathbf{\Phi}}$.

The polynomial $P^{f,\ell,\kappa}_{\mathbf{\Phi}}$ is the straightforward conversion of $\textsc{Comp}\mathbf{\Phi}$ into a circuit, as we can take:

$$P^{f,\ell,\kappa}_{\mathbf{\Phi}}(x_1, \ldots, x_{1+\ell+\kappa}) := x_1 \mathsf{x}^{\ell+\kappa} + P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}(x_2, \ldots, x_{1+\ell+\kappa}) \ ,$$

where $P^{f,\ell,\kappa}_{\widetilde{\mathbf{\Phi}}}$ comes from Lemma 8.10. □

Remark 8.13. The arithmetization techniques in this section extend to handle a variety of products of more complex graphs (other than the directed two-cycle) with De Bruijn graphs. For example, to arithmetize the product of a (small!) clique and a De Bruijn graph, we can simply add "toggle" high-order bits in Lemma 8.11 to keep track of which De Bruijn graph we are in. As another example, to arithmetize the product of a (possibly large) cycle with a De Bruijn graph, instead of using toggle bits, we can add another "artificial" cyclic group to keep track of which De Bruijn graph we are in (the savings on the field size comes from the fact that now we only have to worry about very few structured edges). Arithmetizing such *multiple* extended De Bruijn graphs may be useful for Levin reductions from other problems.

Remark 8.14. Our Lemma 8.9 is a modified (and improved) version of [BSS08, Proposition 5.10], which also gives an embedding of a single extended De Bruijn graph into an affine graph with eight affine edges. Our modifications include removing a "hole" in the encoding (thus halving the required field size) as well as making explicit the asymptotic dependence of the embedding on the two parameters κ and ℓ (which are the logarithm of the height and the width of the graph, respectively). The explicit asymptotic dependence on these parameters allows us to make precise statements about the complexity of the embedding (see Lemma 8.10); these computational statements are needed to eventually imply the Levin reduction that we seek.

8.2 The conversion of parameters for double extended De Bruijn graphs

We now give a conversion of parameters for sGCP with double extended De Bruijn graphs (satisfying certain conditions) to parameters for sACSP.

At high level, the conversion will have to arithmetize the double extended De Bruijn graph (and for this part we have already proved several lemmas in Section 8.1) and to arithmetize the boolean circuits representing the local coloring constraints on the graph; moreover, the parameter conversion has to keep track of how to represent and access large objects, as well as to arithmetize the requirement of input consistency.

We now proceed to the details of the parameter conversion, which are given in the following construction:

Construction 8.15. Fix four (proper) functions $\kappa, \ell, s_0, s_1 \colon \mathbb{N} \to \mathbb{N}$ and functions. Let

$$\Big((\mathsf{c}_{\mathbf{V}}, \mathsf{t}_{\mathbf{V}}, \mathbf{V}), (\alpha_{\mathbf{\Gamma}}, \mathsf{t}_{\mathbf{\Gamma}}, \mathbf{\Gamma}), (\mathsf{c}_{\mathbf{C}}, \mathbf{C}), (\mathsf{s}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \mathbf{K}), (\mathsf{t}_{\mathbf{W}}, \mathbf{W}), (\mathsf{t}_{\mathbf{F}}, \mathbf{F})\Big)$$

be a choice of parameters for sGCP, where, for every $t \in \mathbb{N}$,

- $\alpha_{\mathbf{\Gamma}}(t) = 4$;
- $\Gamma_{t,1}$ is the identity;
- $(V_t, (\Gamma_{t,2}, \Gamma_{t,3}, \Gamma_{t,4}))$ encodes the graph $\mathrm{DDB}(\kappa(t), 2^{\ell(t)} - 1)$;

- $F_t(\mathtt{x}, c_1 \cdots c_{|\mathtt{x}|})$ is the test $\mathtt{x} \stackrel{?}{=} c'_1 \cdots c'_{|\mathtt{x}|}$ where each c'_i is the substring of c_i from bit $s_0(t)$ to bit $s_1(t)$, and
- $\textsc{Find}\mathbf{W}(1^t, i)$ is the i-th vertex in the first column of the first De Bruijn graph in G_t and $|W_t| = 2^{t-(s_1(t)-s_0(t)+1)}$.

Construct a choice of parameters

$$\Big(f, (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \mathbf{H}), (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \mathbf{N}), (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \mathbf{P}), (\mathsf{t}_{\mathbf{I}}, \mathbf{I})\Big)$$

for sACSP as follows:

1. **Constructing Parameter 1.** Define the (proper) field size function $f\colon \mathbb{N} \to \mathbb{N}$ by

$$f(t) := \text{“smallest solution to Equation 2”} \ . \tag{5}$$

Recall that $\mathbb{F}_t = \mathbb{F}_2(\mathsf{x})$ and x is the root of I_t, which is the irreducible polynomial of degree $f(t)$ over $\mathbb{F}_2$ output by $\textsc{FindIrrPoly}(1^{f(t)})$.

Introducing useful families. Consider the following definitions:

- Define the family $\mathbf{X} = \big\{\Xi_t(x) \in \mathbb{F}_2[x]\big\}_{t\in\mathbb{N}}$ by $\Xi_t := \textsc{FindPrimPoly}\big(1^{\ell(t)}\big)$.
- Define the family $\mathbf{Y} = \{Y_t\}_{t\in\mathbb{N}}$ by

$$Y_t := \big\{\xi_{t,0}(x), \ldots, \xi_{t,(2^{\ell(t)}-2)}(x) \in \mathbb{F}_2[x]\big\} \ ,$$

where $\xi_{t,0}(x), \ldots, \xi_{t,(2^{\ell(t)}-2)}(x) \in \mathbb{F}_2[x]$ are the $2^{\ell(t)}-1$ distinct polynomials of degree less than $\ell(t)$ obtained by invoking Claim C.5 with the primitive polynomial $\Xi_t \in \mathbf{X}$. Note that all of these polynomials have degree less than $\deg(I_t) = f(t)$, and therefore $\xi_{t,0}(\mathsf{x}), \ldots, \xi_{t,(2^{\ell(t)}-2)}(\mathsf{x})$ are in $\mathbb{F}_t$.

Moreover, since $\xi_{t,i}(\mathsf{x}) \equiv \mathsf{x}^i \bmod \Xi_t(\mathsf{x})$, each of the $\xi_{t,i}(\mathsf{x})$ can be efficiently computed from the input $(1^t, i)$ by the following algorithm: for $i \in \{0, \ldots, 2^{\ell(t)} - 2\}$,

$\textsc{Find}\mathbf{Y}(1^t, i) \equiv$
(a) Compute $\Xi_t := \textsc{FindPrimPoly}\big(1^{\ell(t)}\big)$.
(b) Compute $\xi_{t,i}(\mathsf{x}) := \mathsf{x}^i \bmod \Xi_t(\mathsf{x})$.
(c) Output $\xi_{t,i}(\mathsf{x})$.

- Define the dimension function $\mathsf{m}_{\widehat{\mathbf{V}}}\colon \mathbb{N} \to \mathbb{N}$ by $\mathsf{m}_{\widehat{\mathbf{V}}}(t) := \ell(t) + \kappa(t) + 1$.
- Define the family $\widehat{\mathbf{V}} = \big\{\widehat{V}_t\big\}_{t\in\mathbb{N}}$ by

$$\begin{aligned}\widehat{V}_t &:= \left\{ \xi_{t,i}(\mathsf{x}) + \mathsf{x}^{\ell(t)-1} \cdot \left(\sum_{j=1}^{\kappa(t)} w_j \mathsf{x}^j\right) + b\mathsf{x}^{\ell(t)+\kappa(t)} \right\}_{\substack{i\in\{0,\ldots,2^{\ell(t)}-2\},\\ w\in\{0,1\}^{\kappa(t)},\\ b\in\{0,1\}}} \\ &= \operatorname{span}(\mathsf{x}^j)_{j\in\{0,\ldots,\ell(t)+\kappa(t)\}} \ ,\end{aligned} \tag{6}$$

so that $\widehat{V}_t$ is a $\mathsf{m}_{\widehat{\mathbf{V}}}(t)$-dimensional linear subset of $\mathbb{F}_t$ specified by a basis

$$\mathcal{B}_{\widehat{V}_t} := \Big(\alpha^{\widehat{\mathbf{V}}}_1(\mathsf{x}), \ldots, \alpha^{\widehat{\mathbf{V}}}_{\mathsf{m}_{\widehat{\mathbf{V}}}(t)}(\mathsf{x})\Big) = \big(1_{\mathbb{F}_t}, \mathsf{x}, \ldots, \mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}}(t)-1}\big) \ .$$

Observe that

$$\widehat{V}_t \supset \Phi_{f(t),\kappa(t),\ell(t)}(V_t) \ . \tag{7}$$

Recall that $\Phi_{f(t),\kappa(t),\ell(t)}$ is the injection from the double extended De Bruijn graph $\mathrm{DDB}(\kappa(t), 2^{\ell(t)}-1) = G_t = (V_t, E_t)$ into the affine graph $\mathrm{AFF}(\mathbb{F}_t, \mathcal{A}_{f(t),\kappa(t),\ell(t)})$, obtained by invoking Lemma 8.11 with the three positive integers $f(t)$, $\kappa(t)$ and $\ell(t)$. (Note that by our choice of parameters $f(t) \geq \ell(t) + \kappa(t) + 1$, as required by the invocation of the lemma.)

Moreover, the basis $\mathcal{B}_{\widehat{V}_t}$ for $\widehat{V}_t$ can be efficiently computed from the input 1^t by the following algorithm:

$\textsc{Find}\widehat{\mathbf{V}}(1^t) \equiv$
(a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$; it has root x.
(b) Compute the dimension function $\mathsf{m}_{\widehat{\mathbf{V}}}(t) := \ell(t) + \kappa(t) + 1$.
(c) Compute the elements $1_{\mathbb{F}_t}, \mathsf{x}, \ldots, \mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}}(t)-1}$ of $\mathbb{F}_2(\mathsf{x})$.
(d) Output the basis $\big(1_{\mathbb{F}_t}, \mathsf{x}, \ldots, \mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}}(t)-1}\big)$.

- Define the dimension function $\mathsf{m}_{\boldsymbol{\Theta}} \colon \mathbb{N} \to \mathbb{N}$ by

$$\mathsf{m}_{\boldsymbol{\Theta}}(t) := \lceil \log(\mathsf{c}_{\mathbf{C}}(t)) \rceil \ . \tag{8}$$

- Define the family $\boldsymbol{\Theta} = \big\{\theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x}) \in \mathbb{F}_t\big\}_{t\in\mathbb{N}}$ by $\theta_{t,i}(\mathsf{x}) := \mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}}(t)-1+i}$. Each $\theta_t(\mathsf{x})$ can be efficiently computed from the input 1^t and index i by the following algorithm:

$\textsc{Find}\boldsymbol{\Theta}(1^t, i) \equiv$
(a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$; it has root x.
(b) Compute the dimension function $\mathsf{m}_{\widehat{\mathbf{V}}}(t) := \ell(t) + \kappa(t) + 1$.
(c) Compute the element $\mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}}(t)-1+i}$ of $\mathbb{F}_2(\mathsf{x})$.
(d) Output $\mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}}(t)-1+i}$.

For convenience, for $k \in \{0, \ldots, 2^{\mathsf{m}_{\boldsymbol{\Theta}}(t)}-1\}$, we define $\theta^{(k)}(\mathsf{x})$ to be the element $\sum_{\ell=1}^{\mathsf{m}_{\boldsymbol{\Theta}}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x})$, where $\sigma_1 \cdots \sigma_{\mathsf{m}_{\boldsymbol{\Theta}}(t)}$ is the binary expansion of $k-1$. Note that $\theta^{(0)}(\mathsf{x}) = 0_{\mathbb{F}_t}$.

- Define the family $\widehat{\mathbf{V}}' = \big\{\widehat{V}'_t\big\}_{t\in\mathbb{N}}$ by

$$\widehat{V}'_t := \bigcup_{b_1,\ldots,b_{\mathsf{m}_{\boldsymbol{\Theta}}(t)}\in\{0,1\}} \big(\widehat{V}_t + b_1\theta_{t,1}(\mathsf{x}) + \cdots + b_{\mathsf{m}_{\boldsymbol{\Theta}}(t)}\theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\big) \ . \tag{9}$$

Observe that

$$\widehat{V}'_t = \mathrm{span}\Big(\alpha_1^{\widehat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}}(t)}^{\widehat{\mathbf{V}}}(\mathsf{x}), \theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\Big) \ ,$$

so that $\widehat{V}'_t$ is a $\mathsf{m}_{\widehat{\mathbf{V}}'}(t)$-dimensional linear subset of $\mathbb{F}_t$ specified by a basis

$$\mathcal{B}_{\widehat{V}'_t} := \Big(\alpha_1^{\widehat{\mathbf{V}}'}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}^{\widehat{\mathbf{V}}'}(\mathsf{x})\Big) = \Big(\alpha_1^{\widehat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}}(t)}^{\widehat{\mathbf{V}}}(\mathsf{x}), \theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\Big) \ .$$

Moreover, the basis $\mathcal{B}_{\widehat{V}'_t}$ for $\widehat{V}'_t$ can be efficiently computed from the input 1^t by the following algorithm:

$\textsc{Find}\widehat{\mathbf{V}}'(1^t) \equiv$
(a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$.
(b) Compute the basis $\Big(\alpha_1^{\widehat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}}(t)}^{\widehat{\mathbf{V}}}(\mathsf{x})\Big) := \textsc{Find}\widehat{\mathbf{V}}(1^t)$.
(c) For $i = 1, \ldots, \mathsf{m}_{\boldsymbol{\Theta}}(t)$, compute $\theta_{t,i}(\mathsf{x}) := \textsc{Find}\boldsymbol{\Theta}(1^t, i)$.
(d) Output the basis $\Big(\alpha_1^{\widehat{\mathbf{V}}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}}(t)}^{\widehat{\mathbf{V}}}(\mathsf{x}), \theta_{t,1}(\mathsf{x}), \ldots, \theta_{t,\mathsf{m}_{\boldsymbol{\Theta}}(t)}(\mathsf{x})\Big)$.

- Define the family $\mathbf{Z} = \big\{\zeta_t(\mathsf{x}) \in \mathbb{F}_t\big\}_{t\in\mathbb{N}}$ by $\zeta_t(\mathsf{x}) := \mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}$. Each $\zeta_t(\mathsf{x})$ can be efficiently computed from the input 1^t by the following algorithm:

  FIND$\mathbf{Z}(1^t) \equiv$
  (a) Compute the irreducible polynomial $I_t := \text{FINDIRRPOLY}\big(1^{f(t)}\big)$; it has root x.
  (b) Compute the dimension function $\mathsf{m}_{\widehat{\mathbf{V}}'}(t) := \ell(t) + \kappa(t) + 1 + \mathsf{m}_{\mathbf{\Theta}}(t)$.
  (c) Compute the element $\mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}$ of $\mathbb{F}_2(\mathsf{x})$.
  (d) Output $\mathsf{x}^{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}$.

2. **Constructing Parameter 2.** Define the following two (proper) functions:

$$\text{a dimension function } \mathsf{m}_{\mathbf{H}}\colon \mathbb{N}\to\mathbb{N}\colon\ \mathsf{m}_{\mathbf{H}}(t) := \mathsf{m}_{\widehat{\mathbf{V}}}(t) + \mathsf{m}_{\mathbf{\Theta}}(t) + 1 = \ell(t) + \kappa(t) + \mathsf{m}_{\mathbf{\Theta}}(t) + 2\ ,$$
$$\text{a time function } \mathsf{t}_{\mathbf{H}}\colon \mathbb{N}\to\mathbb{N}\colon\ \mathsf{t}_{\mathbf{H}}(t) := \mathsf{t}_{\text{IRR}}(f(t)) + O(\ell(t) + \kappa(t) + \mathsf{m}_{\mathbf{\Theta}}(t))\ .$$

3. **Constructing Parameter 3.** Define the family $\mathbf{H} = \big\{H_t\big\}_{t\in\mathbb{N}}$ by

$$H_t := \widehat{V}'_t \cap \big(\widehat{V}'_t + \zeta_t(\mathsf{x})\big)\ .$$

Observe that

$$H_t = \text{span}\Big(\alpha_1^{\widehat{\mathbf{V}}'}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}^{\widehat{\mathbf{V}}'}(\mathsf{x}), \zeta_t(\mathsf{x})\Big)\ ,$$

so that H_t is an $\mathsf{m}_{\mathbf{H}}(t)$-dimensional linear subset of $\mathbb{F}_t$ specified by a basis

$$\mathcal{B}_{H_t} := \Big(\alpha_1^{\mathbf{H}}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\mathbf{H}}(t)}^{\mathbf{H}}(\mathsf{x})\Big) = \Big(\alpha_1^{\widehat{\mathbf{V}}'}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}^{\widehat{\mathbf{V}}'}(\mathsf{x}), \zeta_t(\mathsf{x})\Big)\ .$$

Moreover, the basis $\mathcal{B}_{H_t}$ for H_t can be efficiently computed from the input 1^t by the following algorithm:

FIND$\mathbf{H}(1^t) \equiv$
(a) Compute the irreducible polynomial $I_t := \text{FINDIRRPOLY}\big(1^{f(t)}\big)$.
(b) Compute the basis $\Big(\alpha_1^{\widehat{\mathbf{V}}'}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}^{\widehat{\mathbf{V}}'}(\mathsf{x})\Big) := \text{FIND}\widehat{\mathbf{V}}'(1^t)$.
(c) Compute the element $\zeta_t(\mathsf{x}) := \text{FIND}\mathbf{Z}(1^t)$.
(d) Output the basis $\Big(\alpha_1^{\widehat{\mathbf{V}}'}(\mathsf{x}), \ldots, \alpha_{\mathsf{m}_{\widehat{\mathbf{V}}'}(t)}^{\widehat{\mathbf{V}}'}(\mathsf{x}), \zeta_t(\mathsf{x})\Big)$.

Note that FIND$\mathbf{H}$ is a $\mathsf{t}_{\mathbf{H}}$-time FIND$\mathbf{H}$-space algorithm, where:

$$\mathsf{t}_{\mathbf{H}}(t) := \mathsf{t}_{\text{IRR}}(f(t)) + O(\ell(t) + \kappa(t) + \mathsf{m}_{\mathbf{\Theta}}(t))\ .$$

matching the time complexity defined previously.

4. **Constructing Parameter 4.** Define the following three (proper) functions:

$$\text{a neighborhood size function } \mathsf{c}_{\mathbf{N}}\colon \mathbb{N}\to\mathbb{N}\colon\ \mathsf{c}_{\mathbf{N}}(t) := \mathsf{c}_{\mathbf{C}}(t)(1 + \big|\mathcal{A}_{f(t),\kappa(t),\ell(t)}\big|) + 1 = \mathsf{c}_{\mathbf{C}}(t)10 + 1\ ,$$
$$\text{a time function } \mathsf{t}_{\mathbf{N}}\colon \mathbb{N}\to\mathbb{N}\colon\ \mathsf{t}_{\mathbf{N}}(t) := \mathsf{t}_{\text{IRR}}(f(t)) + \mathsf{t}_{\text{PRIM}}(\ell(t)) + 2\ell(t) + 2\kappa(t) + 1\ ,$$
$$\text{a size function } \mathsf{s}_{\mathbf{N}}\colon \mathbb{N}\to\mathbb{N}\colon\ \mathsf{s}_{\mathbf{N}}(t) := 2\ .$$

Recall that $\mathcal{A}_{f(t),\kappa(t),\ell(t)}$ is the set of affine functions obtained by invoking Lemma 8.11 with the three positive integers $f(t)$, $\kappa(t)$, and $\ell(t)$.

5. **Constructing Parameter 5.** Define the family $\mathbf{N} = \{\vec{N}_t\}_{t\in\mathbb{N}}$ by $\vec{N}_t := \{N_{t,1}, \ldots, N_{t,\mathsf{c}_\mathbf{N}(t)} \colon \mathbb{F}_t \to \mathbb{F}_t\}$, where each $N_{t,i}$ is a degree-1 polynomial and is defined as follows:

   - $N_{t,k}(z) := z + \theta^{(k)}(\mathsf{x})$, for $k = 1, \ldots, 2^{\mathsf{m}_\mathbf{\Theta}(t)}$;
   - $N_{t,2^{\mathsf{m}_\mathbf{\Theta}(t)}+j+9(k-1)}(z) := N_{\mathrm{bin}(j-1)}(z) + \theta^{(k)}(\mathsf{x})$, for $j = 1, \ldots, 8$ and $k = 1, \ldots, \mathsf{c}_\mathbf{C}(t)$, where $N_{000}, \ldots, N_{111}$ are 8 among the 9 affine functions in the set $\mathcal{A}_{f(t),\kappa(t),\ell(t)}$ obtained by invoking Lemma 8.11 with the three positive integers $f(t)$, $\kappa(t)$, and $\ell(t)$;
   - $N_{t,2^{\mathsf{m}_\mathbf{\Theta}(t)}+9+9(k-1)}(z) := N_\times(z) + \theta^{(k)}(\mathsf{x})$, for $k = 1, \ldots, \mathsf{c}_\mathbf{C}(t)$, where $N_\times$ is the 9-th affine function in the set $\mathcal{A}_{f(t),\kappa(t),\ell(t)}$; and
   - $N_{t,10\cdot 2^{\mathsf{m}_\mathbf{\Theta}(t)}+1}(z) := z + \zeta_t(\mathsf{x})$.

   Note that $\mathsf{c}_\mathbf{N}(t) = \mathsf{c}_\mathbf{C}(t)(1 + \left|\mathcal{A}_{f(t),\kappa(t),\ell(t)}\right|) + 1$, so there are no more neighbor functions to define in $\vec{N}_t$.

   Moreover, the representation of each of the $\mathsf{c}_\mathbf{N}(t)$ affine functions in $\vec{N}_t$ can be efficiently computed from the input 1^t by the following algorithm: for $i \in \{1, \ldots, \mathsf{c}_\mathbf{N}(t)\}$,

   $\textsc{FindN}(1^t, i) \equiv$
   (a) Compute the irreducible polynomial $I_t := \textsc{FindIrrPoly}\big(1^{f(t)}\big)$; it has root x.
   (b) For $i = 1, \ldots, \mathsf{m}_\mathbf{\Theta}(t)$, compute the element $\theta_{t,i}(\mathsf{x}) := \textsc{Find}\mathbf{\Theta}(1^t, i)$.
   (c) Compute the element $\zeta_t(\mathsf{x}) := \textsc{Find}\mathbf{Z}(1^t)$.
   (d) Compute the primitive polynomial $\Xi_t := \textsc{FindPrimPoly}\big(1^{\ell(t)}\big)$.
   (e) If $i \in \{1, \ldots, \mathsf{c}_\mathbf{C}(t)\}$, then letting $\sigma_1 \cdots \sigma_{\mathsf{m}_\mathbf{\Theta}(t)} := \mathrm{bin}(i-1)$ output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (1_{\mathbb{F}_t}, \sum_{\ell=1}^{\mathsf{m}_\mathbf{\Theta}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x}))$.
   (f) If $i = 2^{\mathsf{m}_\mathbf{\Theta}(t)} + j + 9(k-1)$ for some $j \in \{1, \ldots, 9\}$ and $k \in \{1, \ldots, \mathsf{c}_\mathbf{C}(t)\}$, then letting $\sigma_1 \cdots \sigma_{\mathsf{m}_\mathbf{\Theta}(t)} := \mathrm{bin}(k-1)$ and $\theta^{(k)}(\mathsf{x}) := \sum_{\ell=1}^{\mathsf{m}_\mathbf{\Theta}(t)} \sigma_\ell \theta_{t,\ell}(\mathsf{x})$,
      i. If $j = 1$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \theta^{(k)}(\mathsf{x}))$.
      ii. If $j = 2$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.
      iii. If $j = 3$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \mathsf{x}^{\ell(t)} + \theta^{(k)}(\mathsf{x}))$.
      iv. If $j = 4$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \mathsf{x}^{\ell(t)} + \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.
      v. If $j = 5$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \theta^{(k)}(\mathsf{x}))$.
      vi. If $j = 6$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.
      vii. If $j = 7$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \mathsf{x}^{\ell(t)} + \theta^{(k)}(\mathsf{x}))$.
      viii. If $j = 8$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (\mathsf{x}, \Xi_t(\mathsf{x}) + \mathsf{x}^{\ell(t)} + \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.
      ix. If $j = 9$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (1_{\mathbb{F}_t}, \mathsf{x}^{\ell(t)+\kappa(t)} + \theta^{(k)}(\mathsf{x}))$.
   (g) If $i = 10 \cdot \mathsf{c}_\mathbf{C}(t) + 1$, then output $(a_{t,i}(\mathsf{x}), b_{t,i}(\mathsf{x})) = (1_{\mathbb{F}_t}, \zeta_t(\mathsf{x}))$.

   Note that $\textsc{Find}\mathbf{N}$ is a $\mathsf{t}_\mathbf{N}$-time algorithm, where:

$$\mathsf{t}_\mathbf{N}(t) := \mathsf{t}_{\textsc{Irr}}(f(t)) + \mathsf{t}_{\textsc{Prim}}(\ell(t)) + O(\ell(t) + \kappa(t) + \mathsf{m}_\mathbf{\Theta}(t)) \ .$$

   matching the time complexity defined previously.

6. **Constructing Parameter 6.** Define the following four (proper) functions:

$$\text{a size function } \mathsf{t}_\mathbf{P} \colon \mathbb{N} \to \mathbb{N} \colon \ \mathsf{t}_\mathbf{P}(t) := \mathsf{s}_\mathbf{K}(t) + \mathsf{c}_\mathbf{N}(t) + \mathsf{m}_\mathbf{H}(t) \ ,$$
$$\text{a time function } \mathsf{s}_\mathbf{P} \colon \mathbb{N} \to \mathbb{N} \colon \ \mathsf{s}_\mathbf{P}(t) := \mathsf{t}_\mathbf{K}(t) + \mathsf{c}_\mathbf{N}(t) + \mathsf{m}_\mathbf{H}(t) \ .$$

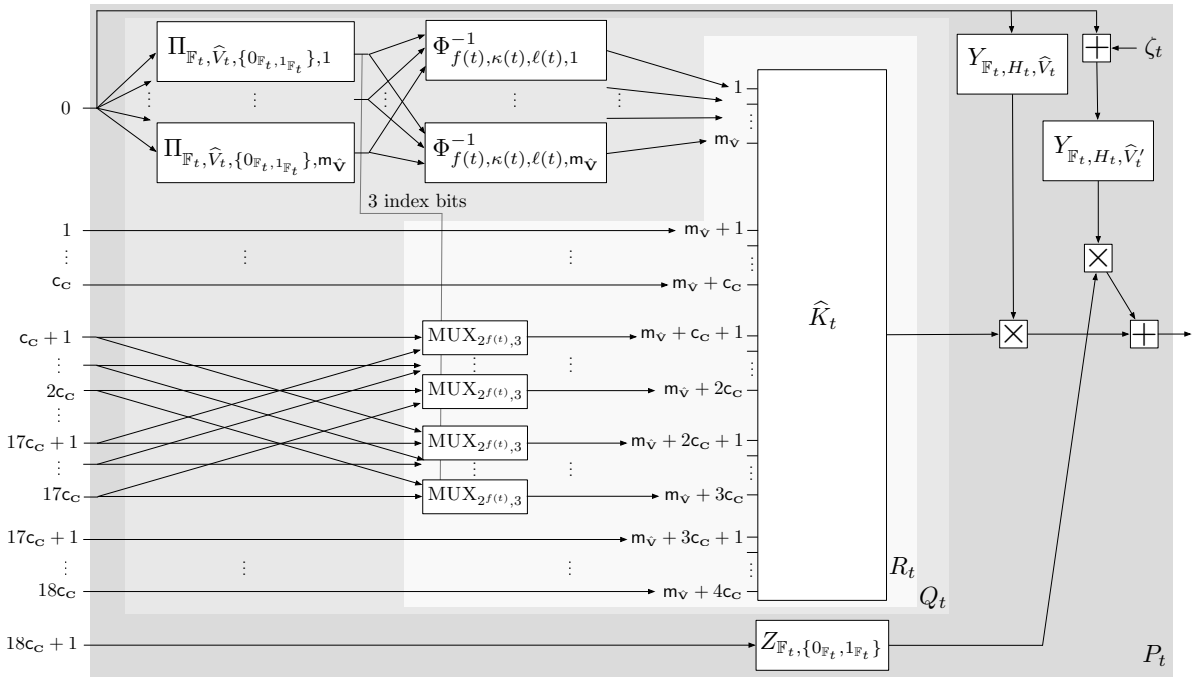


Figure 2: Summary of the construction of P_t starting from K_t.

7. **Constructing Parameter 7.** We now wish to construct $\mathbf{P}$ starting from $\mathbf{K}$. This transformation is given in Figure 2.

8. **Constructing Parameter 8.** Define the following (proper) function:

   a time function $\mathsf{t}_\mathbf{I}\colon \mathbb{N}\to\mathbb{N}$: $\mathsf{t}_\mathbf{I}(t) := \mathsf{t}_\mathbf{W}(t)+\mathsf{t}_\mathbf{\Phi}(t) = \mathsf{t}_\mathbf{W}(t)+\mathsf{t}_{\mathrm{PRIM}}(\ell(t))+\ell(t)^2\log\ell(t)+f(t)$.

   Recall that $\mathsf{t}_\mathbf{W}$ is a function from Parameter 9 in Definition 6.15; also, $\mathsf{t}_\mathbf{\Phi}$ is a function from Lemma 8.12 invoked with the three positive integers $f(t)$, $\kappa(t)$, and $\ell(t)$.

9. **Constructing Parameter 9.** Define the family $\mathbf{I} = \{\vec{I}_t\}_{t\in\mathbb{N}} = \{(I_{t,m})_{m=1}^t\}_{t\in\mathbb{N}}$ by

   $\textsc{Find}\mathbf{I}(1^t,1^m) \equiv$
   (a) If $m \le s_1(t)-s_0(t)+1$:
      i. For $i\in\{1,\ldots,m\}$, $\alpha_i^\mathbf{I}(\mathsf{x}) := \theta_{t,i-1+s_0(t)}(\mathsf{x})$.
   (b) Otherwise:
      i. For $i\in\{1,\ldots,s_1(t)-s_0(t)+1\}$, $\alpha_i^\mathbf{I}(\mathsf{x}) := \theta_{t,i-1+s_0(t)}(\mathsf{x})$.
      ii. For $i\in\{s_1(t)-s_0(t)+1,\ldots,m\}$, $\alpha_i^\mathbf{I}(\mathsf{x}) := \mathsf{x}^{\ell(t)-1}\cdot\mathsf{x}^{i-1-(s_1(t)-s_0(t)+1)}$.
   (c) Output the basis $\left(\alpha_1^\mathbf{I}(\mathsf{x}),\ldots,\alpha_m^\mathbf{I}(\mathsf{x})\right)$ and offset $1_{\mathbb{F}_t}$.

   Thus, the algorithm $\textsc{Find}\mathbf{I}$ is a $\mathsf{t}_\mathbf{I}$-time algorithm, where:

   $$\mathsf{t}_\mathbf{I}(t) := \mathsf{t}_\mathbf{W}(t)+\mathsf{t}_\mathbf{\Phi}(t) \ ,$$

   matching the time and space complexities defined previously.

The construction of the parameters for sACSP is now complete.

8.3 The Levin reduction for double extended De Bruijn graphs

We show that the parameter conversion discussed in Section 8.2 yields a Levin reduction (according to Definition 8.1) from sGCP to sACSP with respect to the class of parameters considered in Construction 8.15.

More precisely, consider the following definitions:

- Define $\mathcal{P}_{\mathrm{DDB}}$ to be the class of parameters for sGCP considered by Construction 8.15, i.e., those choices of parameters for which, for some proper functions $\kappa, \ell \colon \mathbb{N} \to \mathbb{N}$, for all $t \in \mathbb{N}$, $\alpha_{\mathbf{\Gamma}}(t) = 3$ and $G_t = \mathrm{DDB}(\kappa(t), 2^{\ell(t)} - 1)$.

- Define $F_{\mathsf{p}} \colon \{0,1\}^* \to \{0,1\}^*$ to be the function that, on input a parameter choice $\mathsf{par}_{\mathrm{sGCP}} \in \mathcal{P}_{\mathrm{DDB}}$, performs the parameter conversion described in Construction 8.15. More precisely, F_{p} works as follows:

$$\mathsf{par}_{\mathrm{sGCP}} = \begin{pmatrix} (\mathsf{c}_{\mathbf{V}}, \mathsf{t}_{\mathbf{V}}, \textsc{Find}\mathbf{V}), \\ (\alpha_{\mathbf{\Gamma}}, \mathsf{t}_{\mathbf{\Gamma}}, \textsc{Find}\mathbf{\Gamma}), \\ \mathsf{c}_{\mathbf{C}}, \\ (\mathsf{s}_{\mathbf{K}}, \mathsf{t}_{\mathbf{K}}, \textsc{Find}\mathbf{K}), \\ (\mathsf{t}_{\mathbf{W}}, \textsc{Find}\mathbf{W}), \\ (\mathsf{t}_{\mathbf{F}}, \textsc{Comp}\mathbf{F}) \end{pmatrix} \overset{F_{\mathsf{p}}}{\longmapsto} \mathsf{par}_{\mathrm{sACSP}} = \begin{pmatrix} f, \\ (\mathsf{m}_{\mathbf{H}}, \mathsf{t}_{\mathbf{H}}, \textsc{Find}\mathbf{H}), \\ (\mathsf{c}_{\mathbf{N}}, \mathsf{t}_{\mathbf{N}}, \mathsf{s}_{\mathbf{N}}, \textsc{Find}\mathbf{N}), \\ (\mathsf{t}_{\mathbf{P}}, \mathsf{s}_{\mathbf{P}}, \textsc{Find}\mathbf{P}), \\ (\mathsf{t}_{\mathbf{I}}, \textsc{Find}\mathbf{I}) \end{pmatrix},$$

  where the mapping is done by following the definitions of the various new complexity functions and algorithms (for sACSP) based on the old ones (for sGCP).

- Define $F_{\mathsf{w}} \colon \{0,1\}^* \to \{0,1\}^*$ as follows: for every $\mathsf{par}_{\mathrm{sGCP}} \in \mathcal{P}_{\mathrm{DDB}}$, $t \in \mathbb{N}$ and $C \colon V_t \to C_t$,

  $F_{\mathsf{w}}(\mathsf{par}_{\mathrm{sGCP}}, 1^{2^t}, C) \equiv$
  1. Define C'' based on C' (Equation **??**), in turn based on C (Equation **??**).
  2. Define the function $\widetilde{A} \colon \widehat{V}'_t \to \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$ by $\widetilde{A}\big(\alpha(\mathsf{x})\big) := \Big(C'' \circ \Phi^{-1}_{\widehat{V}'_t}\Big)\big(\alpha(\mathsf{x})\big)$ for all $\alpha(\mathsf{x}) \in \widehat{V}'_t$. (See Equation **??**.)
  3. Define the polynomial $A \colon \mathbb{F}_t \to \mathbb{F}_t$ as the low-degree extension of $\widetilde{A}$ in $\mathbb{F}_t$, i.e., $A := \mathrm{LDE}_{\mathbb{F}_t, 1, \widehat{V}_t}(\widetilde{A})$. (See Equation **??**.)
  4. Output A.

We prove the following theorem:

Theorem 8.16. *The pair of functions $(F_{\mathsf{p}}, F_{\mathsf{w}})$ is a Levin reduction from* sGCP *to* sACSP *with respect to $\mathcal{P}_{\mathrm{DDB}}$.*

We divide the proof of Theorem 8.16 into three claims:
- in Claim 8.17, we explain why both F_{p} and F_{w} are polynomial-time computable;
- in Claim 8.18, we show the "completeness" and "soundness" of F_{p}; and
- in Claim 8.19, we show that F_{w} produces good witnesses.

Claim 8.17. *The functions F_{p} and F_{w} are polynomial-time computable.*

Proof. The efficiency of F_{p} easily follows by inspection of how the parameters are converted in Construction 8.15. (Essentially, all the new functions and algorithms are "easy combinations" of previous functions and algorithms, and thus not hard to write down.) The efficiency of F_{w} easily follows from the fact that it can run in time that is polynomial in 1^{2^t}, which is plenty of time for computing the low-degree extension of $\widetilde{A}$ based on the input coloring C. □

Claim 8.18. *For every choice of parameters $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{DDB}}$ and for every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$, $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$ if and only if $(\mathtt{x}, 1^t) \in \text{sACSP}(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}}))$.*

Proof. Fix a choice of parameters $\mathsf{par}_{\text{sGCP}} \in \mathcal{P}_{\text{DDB}}$ for sGCP and an instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$.

Completeness. First, we prove the "completeness" direction of the statement: we need to prove that if $(\mathtt{x}, 1^t) \in \text{sGCP}(\mathsf{par}_{\text{sGCP}})$ then $(\mathtt{x}, 1^t) \in \text{sACSP}(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}}))$. So suppose that $(\mathtt{x}, 1^t) \in \text{sGCP}$, and let $C \colon V_t \to C_t$ be a coloring that witnesses this fact. Define $A := F_{\mathsf{w}}(\mathsf{par}_{\text{sGCP}}, 1^{2^t}, C)$. We now argue that A is a witness for $(\mathtt{x}, 1^t) \in \text{sACSP}(F_{\mathsf{p}}(\mathsf{par}_{\text{sGCP}}))$.

Following the definition of sACSP (Definition 6.18), there are two requirements to satisfy:

- *Satisfiability of constraints.* We need to show that the assignment polynomial A satisfies the constraint polynomial P_t, i.e.,

$$\forall\, \alpha(\mathsf{x}) \in H_t\,,\ P_t\Big(\alpha(\mathsf{x}), A\big(N_{t,1}(\alpha(\mathsf{x}))\big), \ldots, A\big(N_{t,\mathbf{c_N}(t)}(\alpha(\mathsf{x}))\big)\Big) = 0_{\mathbb{F}_t}\ .$$

  And, indeed, since $C \colon V_t \to C_t$ satisfies the coloring constraints induced by K_t and M_t,
  - C satisfies the coloring constraints induced by $\widehat{K}_t$ (see Equation ??), so that
  - C satisfies the coloring constraints induced by R_t (see Equation ??), so that
  - A satisfies the constraints polynomial Q_t (see Equation ??), so that
  - A satisfies the constraint polynomial P_t (see Equation ??),

  as desired.

- *Consistency with the instance.* For every index $i \in \{1, \ldots, |\mathtt{x}|\}$, let $m := \lceil \log |\mathtt{x}| \rceil$ and $\alpha_{i,m}(\mathsf{x})$ be the i-th element in $\text{span}(\alpha_1^{\mathbf{I}}(\mathsf{x}), \ldots, \alpha_m^{\mathbf{I}}(\mathsf{x}))$. We need to show that the assignment polynomial A is consistent with the instance $(\mathtt{x}, 1^t)$, i.e.,

$$\mathtt{x} = \text{bit}\big(A(\alpha_{1,m}(\mathsf{x}))\big) \cdots \text{bit}\big(A(\alpha_{|\mathtt{x}|,m}(\mathsf{x}))\big)\ ,$$

  And, indeed, since we know that $F_t\big(\mathtt{x}, \big(C(v_i)\big)_{i=1}^{|\mathtt{x}|}\big) = 0$, where v_i is the i-th element in W_t, and we also know by assumption that $F_t(\mathtt{x}, c_1 \cdots c_{|\mathtt{x}|})$ is the test $\mathtt{x} \stackrel{?}{=} c'_1 \cdots c'_{|\mathtt{x}|}$ where each c'_i is the substring of c_i from bit $s_0(t)$ to bit $s_1(t)$, by our definition of A the above equation holds: essentially, we have "distributed" the bit of each color of a vertex v_i among various field elements, and all of these field elements that are relevant for the test are accounted for in the definition of I_t (see Equation ??).

This concludes the proof of the "completeness" direction of the statement.

Soundness. Next, we prove the "soundness" direction of the statement: we need to prove that if $(\mathtt{x}, 1^t) \in \mathrm{sACSP}(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}}))$ then $(\mathtt{x}, 1^t) \in \mathrm{sGCP}(\mathsf{par}_{\mathrm{sGCP}})$. So suppose instead that we have $(\mathtt{x}, 1^t) \in \mathrm{sACSP}(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}}))$, and let $A \colon \mathbb{F}_t \to \mathbb{F}_t$ be a polynomial that witnesses this fact. Consider $A|_{\widehat{V}_t}$, the restriction of A to $\widehat{V}_t$. We argue that $A|_{\widehat{V}_t}$ takes on values in $\{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$. Indeed, assume by way of contradiction that there exists $\alpha(\mathtt{x}) \in \widehat{V}_t$ such that $A(\alpha(\mathtt{x})) \notin \{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$. Let $\alpha'(\mathtt{x}) := N_{t,\mathbf{c_N}(t)}(\alpha(\mathtt{x})) = \alpha(\mathtt{x}) + \zeta_t(\mathtt{x})$, and note that $\alpha'(\mathtt{x}) \in \widehat{V}_t + \zeta_t(\mathtt{x})$. By setting $\big(x_0, x_1, \ldots, x_{\mathbf{c_N}(t)}\big) := \big(\alpha(\mathtt{x}), A(N_{t,1}(\alpha(\mathtt{x}))), \ldots, A(N_{t,\mathbf{c_N}(t)}(\alpha(\mathtt{x})))\big)$, this means that the second summand in Equation ?? does not vanish; since the first summand does vanish, we reach a contradiction to the fact that A is a witness to $(\mathtt{x}, 1^t) \in \mathrm{sACSP}$ (because it does not satisfy the constraint polynomial P_t). We conclude that $A|_{\widehat{V}_t}$ takes on values in $\{0_{\mathbb{F}_t}, 1_{\mathbb{F}_t}\}$, as claimed. We can now define the function $C \colon V_t \to C_t$ by

$$\forall\, v \in V_t\,,\; C(v) := \Big(A\big(\Phi_{f(t),\kappa(t),\ell(t)}(v) + \theta^{(k)}\big)\Big)_{k=1,\ldots,\mathbf{c_G}(t)}\,. \tag{10}$$

It is easy to see that C is a witness for $(\mathtt{x}, 1^t) \in \mathrm{sGCP}(\mathsf{par}_{\mathrm{sGCP}})$ by inspection of the construction. This concludes the proof of the "soundness" direction of the statement. As both directions have now been shown, the proof is now complete. □

Claim 8.19. *For every choice of parameters $\mathsf{par}_{\mathrm{sGCP}} \in \mathcal{P}_{\mathrm{DDB}}$ and for every instance $(\mathtt{x}, 1^t) \in \{0,1\}^*$, if C is a witness to "$(\mathtt{x}, 1^t) \in \mathrm{sGCP}(\mathsf{par}_{\mathrm{sGCP}})$" then $F_{\mathsf{w}}(1^t, C)$ is a witness to "$(\mathtt{x}, 1^t) \in \mathrm{sACSP}(F_{\mathsf{p}}(\mathsf{par}_{\mathrm{sGCP}}))$".*

Proof. The claim follows immediately from the fact that in the proof of the "completeness" direction of the statement from Claim 8.18, we have constructed, starting from a valid coloring C and according to F_{p}, a valid assignment polynomial A for $(\mathtt{x}, 1^t)$: by first considering the function $\widetilde{A}$, and then takings its low-degree extension over $\widehat{V}_t$ in $\mathbb{F}_t$. □

A Routing Networks

We provide additional details and references for the facts mentioned in Definition 6.2. Concretely, we describe an algorithm for routing a given permutation over "three and a half" De Bruijn graphs connected in tandem; this will imply, in particular, a proof of Claim 6.5. While the routing properties of De Bruijn graphs are folklore, we have not been able to find explicit algorithms in the literature for routing, so we devote this section to deduce an explicit routing algorithm.

A.1 Butterfly Networks and Isomorphic Graphs of Interest

We begin by introducing a fundamental family of graphs studied in parallel systems: butterfly networks.

Definition A.1. *Let κ be a positive integer. The κ-dimensional* **butterfly network**, *denoted BN_κ, is a directed graph consisting of $\kappa+1$ "columns" each containing 2^κ vertices identified with κ-bit strings. A vertex v in layer $i \in \{0,\ldots,\kappa-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifier w and $w \oplus e_{i+1}$. (See Figure 3 for an example.)*

A *reversed* butterfly network is, as the name suggests, obtained by reversing the direction of the edges in a butterfly network:

Definition A.2. *Let κ be a positive integer. The κ-dimensional* **reversed butterfly network**, *denoted $\mathrm{BN}_\kappa^{\mathrm{r}}$, is a directed graph consisting of $\kappa+1$ "columns" each containing 2^κ vertices identified with κ-bit strings. A vertex v in layer $i \in \{0,\ldots,\kappa-1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifier w and $w \oplus e_{\kappa-i}$. (See Figure 4 for an example.)*

The reversed butterfly network $\mathrm{BN}_\kappa^{\mathrm{r}}$ is in fact isomorphic to the butterfly network BN_κ via a graph isomorphism permuting the rows by reversing the bits of a row's identifier.

Claim A.3. *Let κ be a positive integer. The map $\phi_\kappa\colon \{0,\ldots,\kappa\} \times \{0,1\}^\kappa \to \{0,\ldots,\kappa\} \times \{0,1\}^\kappa$ defined by $\phi_\kappa(i,w) = (i, w^{\mathrm{r}})$ is a graph isomorphism from $\mathrm{BN}_\kappa^{\mathrm{r}}$ to BN_κ.*

Note that indeed the graph isomorphism ϕ_κ is "row-rigid" in the sense that it only "shuffles" the rows of the reversed butterfly network, by mapping a row with identifier w to the row w^{r}.

Proof. Let $a = ((i,w),(i+1,w'))$ be an edge in $\mathrm{BN}_\kappa^{\mathrm{r}}$. Then:

- If $w' = w$, then $\phi_\kappa(a) = ((i,w^{\mathrm{r}}),(i+1,w^{\mathrm{r}}))$ is an edge in BN_κ.
- If $w' = w \oplus e_{\kappa-i}$, then $\phi_\kappa(a) = ((i,w^{\mathrm{r}}),(i+1,(w \oplus e_{\kappa-i})^{\mathrm{r}})) = ((i,w^{\mathrm{r}}),(i+1,w^{\mathrm{r}} \oplus e_{i+1}))$ is an edge in BN_κ.

Conversely, let $b = ((i,w),(i+1,w'))$ be an edge in BN_κ. Then:

- If $w' = w$, then $\phi_\kappa^{-1}(b) = ((i,w^{\mathrm{r}}),(i+1,w^{\mathrm{r}}))$ is an edge in $\mathrm{BN}_\kappa^{\mathrm{r}}$.
- If $w' = w \oplus e_{i+1}$, then $\phi_\kappa^{-1}(b) = ((i,w^{\mathrm{r}}),(i+1,(w \oplus e_{i+1})^{\mathrm{r}})) = ((i,w^{\mathrm{r}}),(i+1,w^{\mathrm{r}} \oplus e_{\kappa-i}))$ is an edge in $\mathrm{BN}_\kappa^{\mathrm{r}}$.

As there are the same number of edges in both $\mathrm{BN}_\kappa^{\mathrm{r}}$ and BN_κ, the proof of the claim is complete. □

We shall also be interested in De Bruijn graphs.

Definition A.4. *Let κ be a positive integer. The κ-dimensional* **De Bruijn graph**, *denoted* DB_κ, *is a directed graph consisting of $\kappa + 1$ "columns" each containing 2^κ vertices identified with κ-bit strings. A vertex v in layer $i \in \{0, \ldots, \kappa - 1\}$ with identifier $w \in \{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifier* $\mathrm{sr}(w)$ *and* $\mathrm{sr}(w) \oplus e_1$, *where* sr *denotes the cyclic "shift right" bit operation. (See Figure 5 for an example.)*

A De Bruijn graph DB_κ is also isomorphic to the butterfly network BN_κ via a graph isomorphism that cyclically shifts the bits of the identifier of a vertex depending on the index of the column in which the vertex lies.

Claim A.5. *Let κ be a positive integer. The map $\psi_\kappa \colon \{0, \ldots, \kappa\} \times \{0,1\}^\kappa \to \{0, \ldots, \kappa\} \times \{0,1\}^\kappa$ defined by $\psi_\kappa(i, w) = (i, \mathrm{sr}^{i-1}(w^\mathrm{r}))$ is a graph isomorphism from* BN_κ *to* DB_κ.

Note that, unlike the graph isomorphism from a reversed butterfly network to a butterfly network, the graph isomorphism from a butterfly network to a De Bruijn graph does not "mess up" the order of vertices in the first and last columns; this fact is important later in this section because it implies that the graph isomorphism easily extends to when networks are connected in tandem.

Proof. Let $a = ((i, w), (i+1, w'))$ be an edge in BN_κ. Then:

- If $w' = w$, then $\psi_\kappa(a) = ((i, \mathrm{sr}^{i-1}(w)^\mathrm{r}), (i+1, \mathrm{sr}^i(w^\mathrm{r})))$ is an edge in DB_κ.
- If $w' = w \oplus e_{i+1}$, then $\psi_\kappa(a) = ((i, \mathrm{sr}^{i-1}(w^\mathrm{r})), (i+1, \mathrm{sr}^i((w \oplus e_{i+1})^\mathrm{r}))) = ((i, \mathrm{sr}^{i-1}(w^\mathrm{r})), (i+1, \mathrm{sr}^i(w^\mathrm{r}) \oplus e_1))$ is an edge in DB_κ.

Conversely, let $b = ((i, w), (i+1, w'))$ be an edge in DB_κ. Then:

- If $w' = \mathrm{sr}(w)$, then $\phi_\kappa^{-1}(b) = ((i, \mathrm{sl}^{i-1}(w)^\mathrm{r}, (i+1, \mathrm{sl}^i(\mathrm{sr}(w))^\mathrm{r})) = ((i, \mathrm{sl}^{i-1}(w)^\mathrm{r}, (i+1, \mathrm{sl}^{i-1}(w)^\mathrm{r}))$ is an edge in BN_κ.
- If $w' = \mathrm{sr}(w) \oplus e_1$, then $\phi_\kappa^{-1}(b) = ((i, \mathrm{sl}^{i-1}(w)^\mathrm{r}, (i+1, \mathrm{sl}^i(\mathrm{sr}(w) \oplus e_1)^\mathrm{r})) = ((i, \mathrm{sl}^{i-1}(w)^\mathrm{r}, (i+1, \mathrm{sl}^{i-1}(w)^\mathrm{r} \oplus e_{i+1}))$ is an edge in BN_κ.

As there are the same number of edges in both BN_κ and DB_κ, the proof of the claim is complete. □

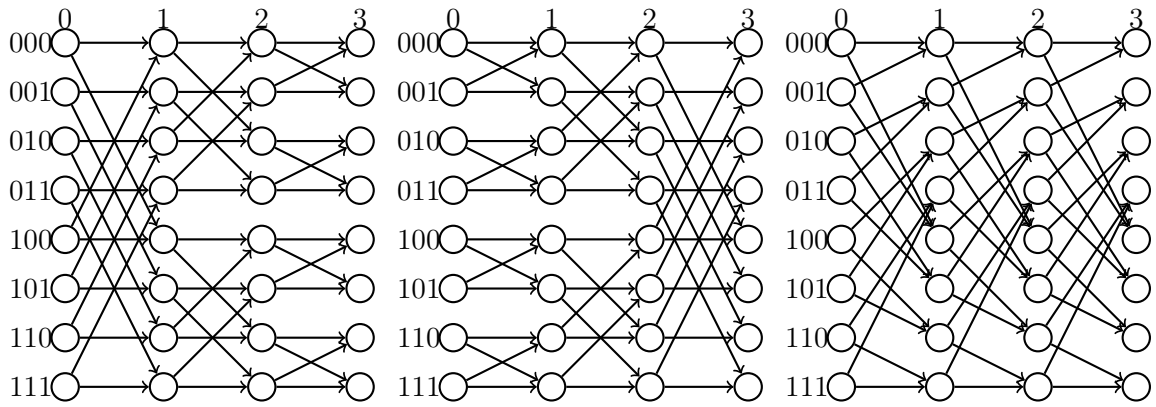


Figure 3: Three-dimensional butterfly network.

Figure 4: Three-dimensional reversed butterfly network.

Figure 5: Three-dimensional De Bruijn graph

A.2 Beneš Networks and Their Rearrangeability

A Beneš network is a routing network that is able to route *any* permutation [Ben65]; this property is known as *re-arrangeability*.

A Beneš network is the "concatenation" of a butterfly network and a reversed butterfly network:

Definition A.6. *Let κ be a positive integer. The κ-dimensional* Beneš network*, denoted* BENEŠ_κ*, is a κ-dimensional butterfly network and a κ-dimensional reversed butterfly network connected in tandem. More precisely,* BENEŠ_κ *is a directed graph with $2\kappa+1$ "columns" numbered $0,\ldots,2\kappa$, each containing 2^κ vertices identified with κ-bit strings; a vertex $v=(i,w)$ in layer $i\in\{0,1,\ldots,2\kappa-1\}$ with identifier $w\in\{0,1\}^\kappa$ has two neighbors in layer $i+1$ with identifiers w and $w\oplus a_i$ respectively, where $a_i\in\{0,1\}^\kappa$ is equal to e_{i+1} if $i\in\{0,\ldots,\kappa-1\}$ and is equal to $e_{2\kappa-i}$ if $i\in\{\kappa,\ldots,2\kappa-1\}$.*

More concretely, an κ-dimensional Beneš network can be used to route either $2^{\kappa+1}$ packets using edge disjoint paths (where each vertex in the network receives and sends two packets) [Lei92, Theorem 3.10] or to route 2^κ packets using vertex disjoint paths (where each vertex in the network receives and sends exactly one packet) [Lei92, Theorem 3.11].

As we are interested in the latter form of routing, we recall explicitly the theorem and its constructive proof:

Theorem A.7 ([Wak68, OTW71, Lei92])**.** *Let κ be a positive integer and $\pi\colon\{0,1\}^\kappa\to\{0,1\}^\kappa$ a permutation. There exists a set S_π of 2^κ vertex-disjoint paths such that each vertex $(0,w)$ in* BENEŠ_κ *is connected to $(2\kappa+1,\pi(w))$. Moreover, S_π can be found in $O(\kappa\cdot 2^\kappa)$ time and space.*

Nassimi and and Sahni [NS82] show how to modify the algorithm from the proof of Theorem A.7 to run in parallel time $O(\kappa^2)$.

Proof. The existence of the "routing" S_π will follow from the correctness of the algorithm that we present, which will always return a valid solution.

First let us describe the idea at high level. The idea is to use the recursive structure of a Beneš network; indeed, note that, by removing the "leftmost" and "rightmost" layers of a Beneš network (i.e., layer 0 and layer 2κ), we obtain two $(\kappa-1)$-dimensional Beneš networks — a "top" one and a "bottom" one. We can recursively solve any routing problem on the smaller Beneš networks, and thus we are left to reduce the routing on the original network to use the routing on the smaller sub-networks. (And, of course, the base case with $\kappa=1$ is trivial to solve.)

A fast algorithm immediately follows from the above intuition. Indeed, consider the algorithm BENEŠROUTE that, on input a positive integer κ and $\pi\colon\{0,1\}^\kappa\to\{0,1\}^\kappa$ (specified as a table), is defined as follows:

$\text{BENEŠROUTE}(\kappa,\pi)\equiv$

1. If $\kappa=1$, then:
   (a) If $\pi(0)=0$ and $\pi(1)=1$ then route both $(0,0)$ and $(0,1)$ using identity edges.
   (b) Otherwise, if $\pi(0)=1$ and $\pi(1)=0$ then route both $(0,0)$ and $(0,1)$ using cross edges.
2. If $\kappa>1$, then repeat the following until all vertices in the column 0 are routed:
   (a) Choose vertex $u=(0,w)$ that is not routed ant let $(2\kappa,w')$ where $w'=\pi(w)$.
   (b) Route u using the upper sub-network:
      i. "Forward" route u to $(1,w)$ if $w_1=0$ and to $(1,w\oplus e_1)$ otherwise.

ii. "Backward" route $(2\kappa, w')$ to $(2\kappa-1, w')$ if $w'_1 = 0$ and to $(2\kappa-1, w' \oplus e_1)$ otherwise.

(c) Set $\pi'(\widehat{w})$ to be $\widehat{w}'$ where $\widehat{w}$ and $\widehat{w}'$ are the least significant $\kappa-1$ bits of w and w' respectively.

(d) Route $(2\kappa, w' \oplus e_1)$ using the lower sub-network: Let $(0, w'')$ be the source of $(2\kappa, w')$ where $w'' = \pi^{-1}(w')$.

i. Route $(2\kappa, w')$ to $(2\kappa-1, w')$ if $w'_1 = 1$ and to $(2\kappa, w' \oplus e_1)$ otherwise.

ii. Route $(0, w'')$ to $(1, w'')$ if $w''_1 = 1$ and to $(1, w'' \oplus e_1)$ otherwise.

(e) Set $\pi''^{-1}(\widehat{w}')$ to be $\widehat{w}''$ where $\widehat{w}'$ and $\widehat{w}''$ are the least significant $\kappa-1$ bits of w' and w'' respectively.

(f) Set $u = (0, w'' \oplus e_1)$.

(g) If u is routed then goto Step 2 Otherwise goto Step 2b.

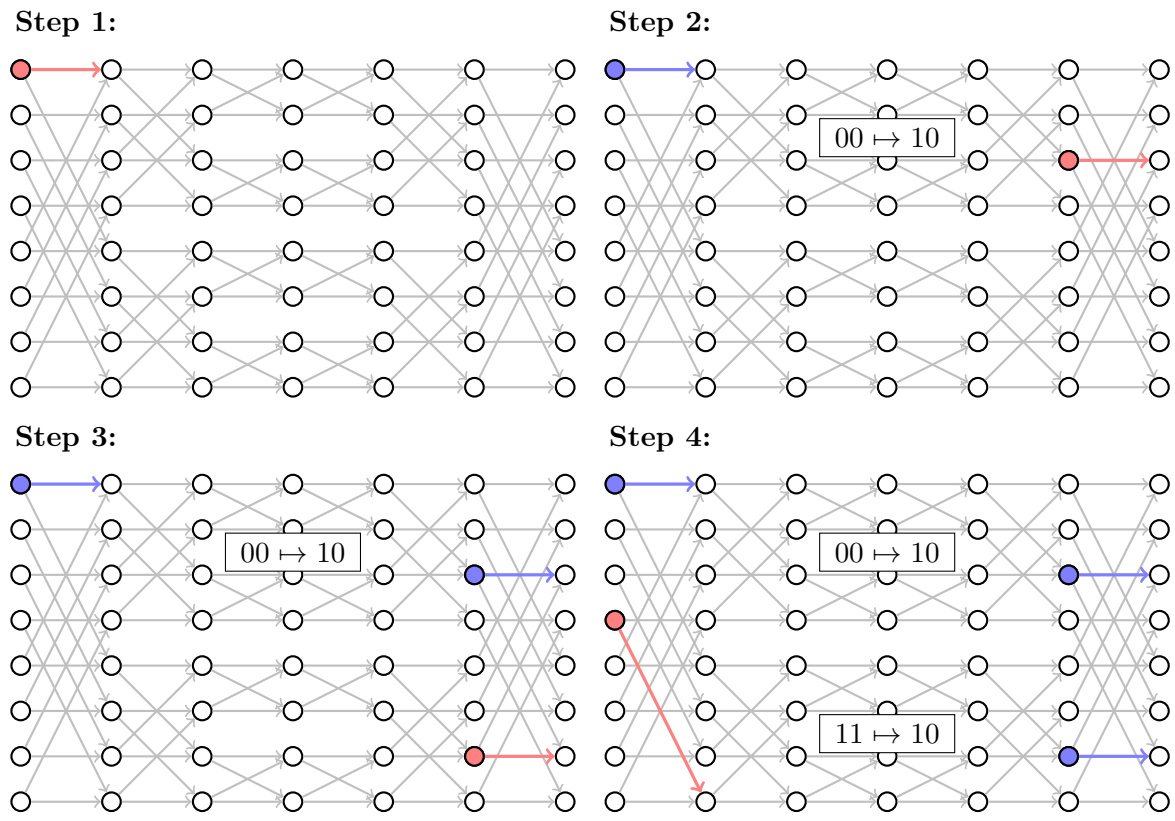
3. Run BenešRoute$(\kappa-1, \pi')$ recursively on the upper sub-network.

4. Run BenešRoute$(\kappa-1, \pi'')$ recursively on the lower sub-network.

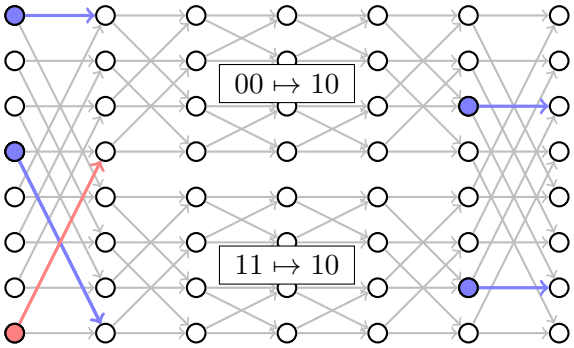
The correctness of BenešRoute easily follows from an induction argument. Furthermore, since the algorithm visits every vertex in Beneš$_\kappa$ a constant number of times and Beneš$_\kappa$ has $(\kappa+1)\cdot 2^\kappa$ vertices, we deduce that the space and time complexities are $O(\kappa \cdot 2^\kappa)$. □

Example A.1. We give a pictorial example for the computation of BenešRoute$(3, \pi)$ when π is given by the following permutation:

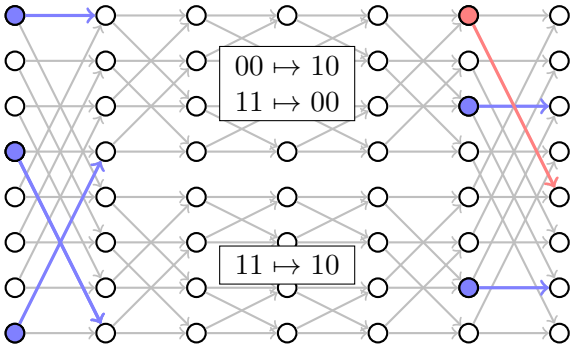
$000 \mapsto 010$, $001 \mapsto 011$, $010 \mapsto 101$, $011 \mapsto 110$, $100 \mapsto 000$, $101 \mapsto 001$, $110 \mapsto 111$, $111 \mapsto 100$.



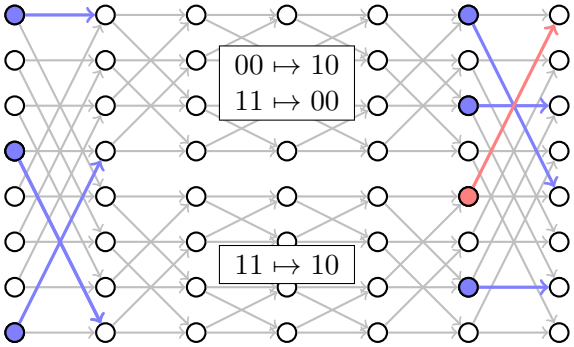
Step 5:



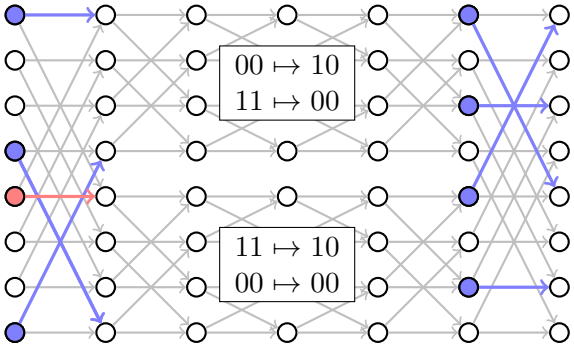
Step 6:



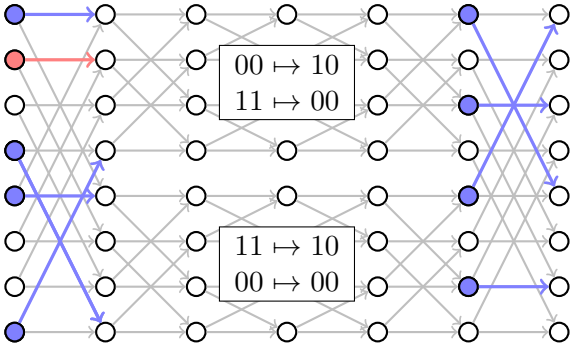
Step 7:



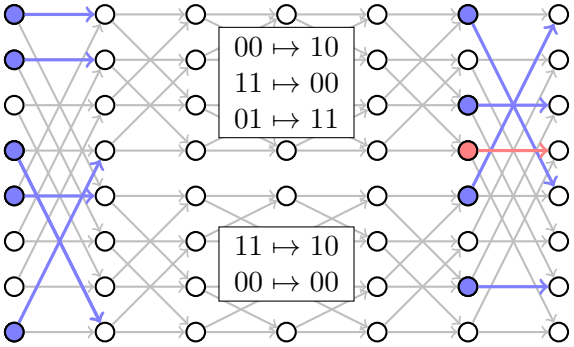
Step 8:



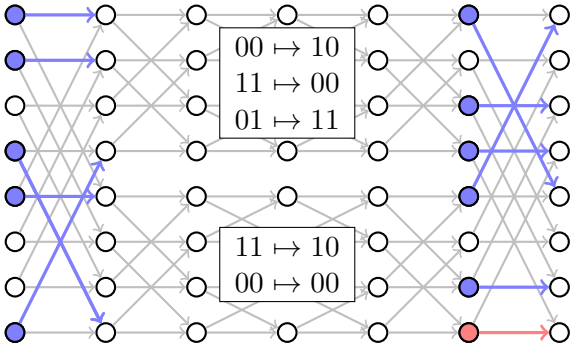
Step 9:



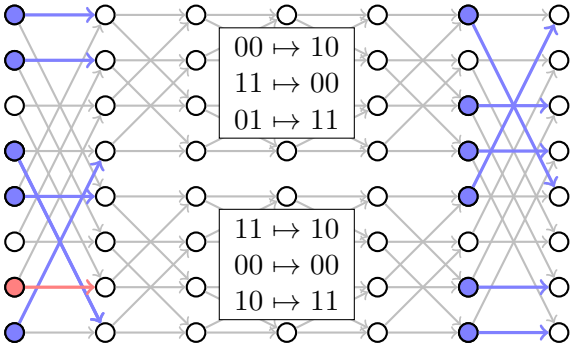
Step 10:



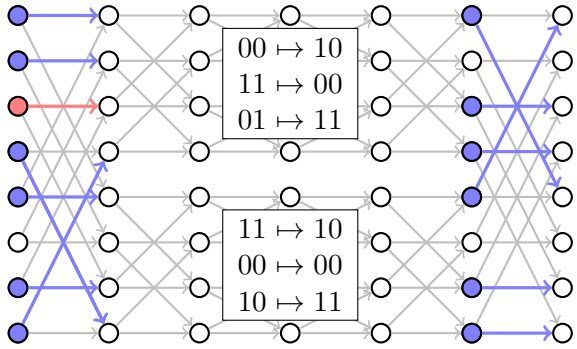
Step 11:



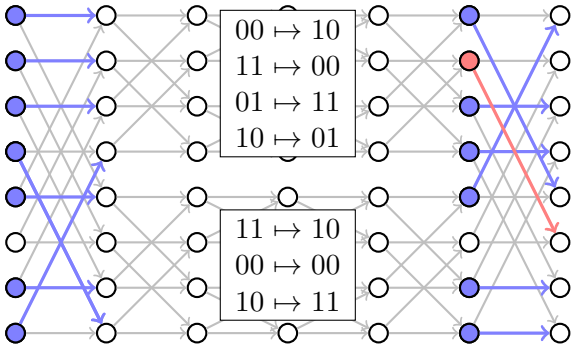
Step 12:



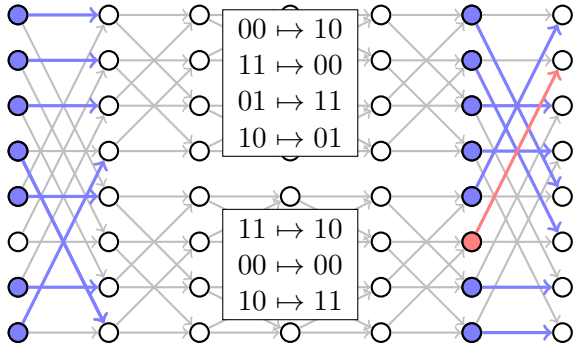
Step 13:



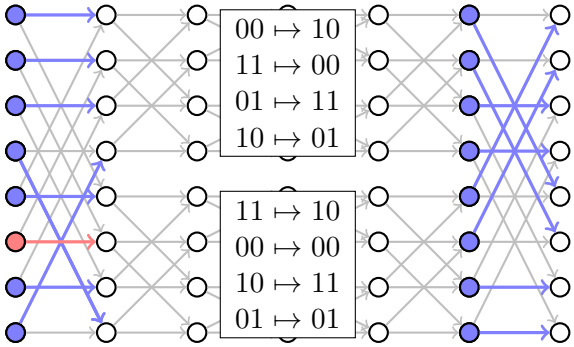
Step 14:



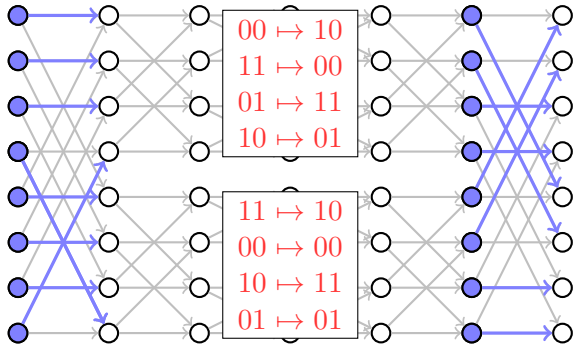
Step 15:



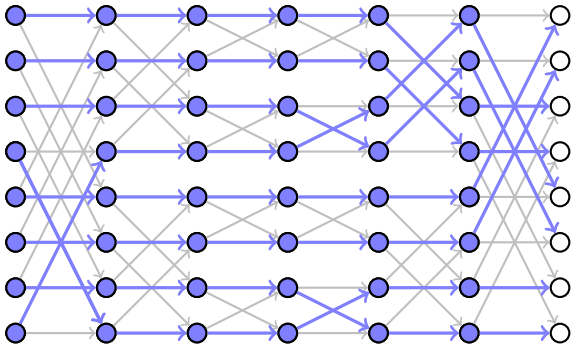
Step 16:



Step 17:



Step 18:



In the last step, the algorithm recursively routes the upper permutation through the upper sub-network and the lower permutation through the lower sub-network, and finally obtains the desired 8 vertex disjoint paths.

A.3 Routing Bit-Reversal Permutations

The κ-dimensional *bit-reversal* permutation $\mathrm{br}_\kappa \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ is the permutation that, for every $w \in \{0,1\}^\kappa$, is defined by $\mathrm{br}_\kappa(w) := w^{\mathrm{r}}$.

We observe that two butterfly networks connected in tandem are capable of "routing" the bit-reversing permutation.

Claim A.8. *Let κ be a positive integer. The permutation br_κ can be routed on two κ-dimensional butterfly networks connected in tandem. Moreover, the routing can be found in $O(\kappa \cdot 2^\kappa)$ time and space.*

Proof. Consider the operations of "fold left" and "fold right", denoted fl_κ and fr_κ respectively, on a κ-bit string w that are defined as follows: letting $w = \sigma_0||\sigma_1$ if κ is even and $w = \sigma_0||b||\sigma_1$ if κ is odd, with $|\sigma_0| = |\sigma_1|$ and $b \in \{0,1\}$,

	fold left		**fold right**	
even case	$\sigma_0\|\|\sigma_1$	$\longrightarrow (\sigma_0 \oplus \sigma_1^{\mathrm{r}})\|\|\sigma_1$	$\sigma_0\|\|\sigma_1$	$\longrightarrow \sigma_0\|\|(\sigma_1 \oplus \sigma_0^{\mathrm{r}})$
odd case	$\sigma_0\|\|b\|\|\sigma_1$	$\longrightarrow (\sigma_0 \oplus \sigma_1^{\mathrm{r}})\|\|b\|\|\sigma_1$	$\sigma_0\|\|b\|\|\sigma_1$	$\longrightarrow \sigma_0\|\|b\|\|(\sigma_1 \oplus \sigma_0^{\mathrm{r}})$

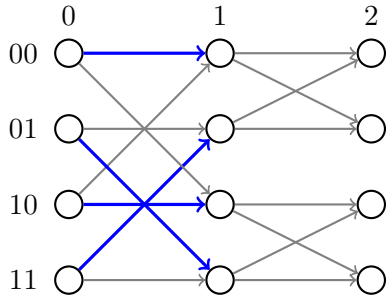
.

Now observe that $\mathrm{br}_\kappa(w) = \big(\mathsf{fl}_\kappa \circ \mathsf{fr}_\kappa \circ \mathsf{fl}_\kappa\big)(w)$.

We now show the following two facts:

1. One can route fl_κ using layers 0 through $\lceil \kappa/2 \rceil$ of a κ-dimensional butterfly network. The proof is by induction over the dimension κ:

   - For $\kappa = 1$, we have that fl_1 is the identity permutation and thus can be trivially routed using layers 0 and 1 of a 1-dimensional butterfly network by simply using the two straight edges.
   - For $\kappa = 2$, the routing is the following:



   - For $\kappa > 2$, define the function $f_\kappa \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ as follows

$$f_\kappa(w_1 \cdots w_\kappa) = w_1 \oplus w_\kappa || w_2 \cdots w_\kappa \ .$$

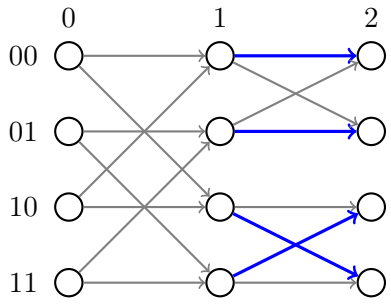
     For any $w \in \{0,1\}^\kappa$, $\mathsf{fl}_\kappa(w)$ can be computed using f_κ and $\mathsf{fl}_{\kappa-2}$ in the following way:
     (a) Compute $w_1 \oplus w_\kappa || w_2 \cdots w_\kappa \leftarrow f_\kappa(w)$.
     (b) Compute $w' = \mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1})$.
     (c) Output $w_1 \oplus w_\kappa || w' || w_\kappa$.

     Recall that each vertex (i, w) is connected to $(i+1, w)$ and $(i+1, w \oplus e_{i+1})$ in BN_κ. Thus, we can first route any $(0, w)$ to $(1, f_\kappa(w))$ using layers 0 and 1 of the butterfly network, by choosing the appropriate edge depending on w_κ. Then, using the induction hypothesis, we can route each vertex $(1, w_1 \cdots w_\kappa)$ to $(\lceil \kappa/2 \rceil, w_1 || \mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1}) || w_\kappa)$ using the two $(\kappa-1)$-dimensional subnetworks corresponding to the two different "fixed" values of w_1. This could be done since by the induction hypothesis for any $w_2 \cdots w_{\kappa-1}$ we can route $(1, w_2 \cdots w_{\kappa-1})$ to $(\lceil (\kappa-2)/2 \rceil, \mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1}))$ using a $(\kappa-2)$-dimensional network and therefore any $(1, w_2 \cdots w_\kappa)$ can be routed to $(\lceil (\kappa-1)/2 \rceil,$ $\mathsf{fl}_{\kappa-2}(w_2 \cdots w_{\kappa-1}) || w_\kappa)$ using a $(\kappa-1)$-dimensional butterfly network.

2. One can route fr_κ using layers $\lceil \kappa/2 \rceil$ through κ of a κ-dimensional butterfly network.

   - For $\kappa = 1$, we have that fr_1 is the identity permutation and thus can be trivially routed using layers 0 and 1 of a 1-dimensional butterfly network by simply using the two straight edges.

- For $\kappa = 2$, the routing is the following:



- For $\kappa > 2$, define the function $g_\kappa \colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ as follows

$$g_\kappa(w_1 \cdots w_\kappa) = w_1 \cdots w_{\kappa-1} || w_\kappa \oplus w_1 \ .$$

For any $w \in \{0,1\}^\kappa$, $\mathsf{fr}_\kappa(w)$ can be computed using g_κ and $\mathsf{fr}_{\kappa-2}$ in the following way:

(a) Compute $w'_1 \cdots w'_{\kappa-1} = \mathsf{fr}_{\kappa-2}(w_2 \cdots w_{\kappa-1})$.
(b) Compute $w_1 || w'_1 \cdots w'_{\kappa-1} || w_\kappa \oplus w_1 \leftarrow f_\kappa(w_1 || w'_1 \cdots w'_{\kappa-1} || w_\kappa)$.
(c) Output $w_1 || w'_1 \cdots w'_{\kappa-1} || w_\kappa \oplus w_1$.

By using the induction hypothesis, we know that we can route each vertex $(\lceil \kappa/2 \rceil, w_1 \cdots w_\kappa)$ to
$(\kappa - 1, w_1 || \mathsf{fr}_{\kappa-2}(w_2 \cdots w_{\kappa-1}) || w_\kappa)$ using the $2^{\lfloor \kappa/2 \rfloor}$ $\lceil \kappa/2 \rceil$-dimensional subnetworks corresponding to the four different "fixed" values for the tuple $(w_1, \cdots, w_{\lfloor \kappa/2 \rfloor})$. Recall that each vertex (i, w) is connected to $(i+1, w)$ and $(i+1, w \oplus e_{i+1})$ in BN_κ. Thus, we can next route any $(\kappa - 1, w)$ to $(\kappa, g_\kappa(w))$ using layers $\kappa - 1$ and κ of the butterfly network, by choosing the appropriate edge depending on w_κ.

Finally, we note that we are essentially done: to route $\mathrm{br}_\kappa = \mathsf{fl}_\kappa \circ \mathsf{fr}_\kappa \circ \mathsf{fl}_\kappa$ on two κ-dimensional butterfly networks connected in tandem, we first route fl_κ using the first half of the first network, then route fr_κ using the second half of the first network, then route fl_κ using the first half of the second network, and then finally route the identity permutation using the remaining second half of the second network. □

A.4 Simulating Beneš Networks with Butterfly Networks

Recall that a Beneš network is the concatenation of a butterfly network and a reversed butterfly network. For technical reasons, the reversed butterfly network is very inconvenient from an arithmetization standpoint (due to the need to keep the out-degree of the graph embedding in Section 8 as low as possible). Thus, we seek to do without it.

Specifically, we now show how, as far as routing is concerned, the reversed butterfly network can be "simulated" via five butterfly networks. In particular, because Beneš networks are re-arrangeable (see Theorem A.7), we deduce that so are six butterfly networks connected in tandem, a graph which we denote $\mathrm{BN}_\kappa^{\ddagger 6}$.

The high level idea is to simply use the the graph isomorphism from a reversed butterfly network to a butterfly network given by Claim A.3. However, the isomorphism "messes up" the connections with the preceding butterfly network. Fortunately, the isomorphism only shuffles rows of the reversed butterfly network according to a bit reversal permutation, and therefore we can "undo" the damage by preceding and following the butterfly network obtained by the isomorphism by bit-reversal permutations (each of which can be realized with two butterfly networks, as we saw in Claim A.8).

We therefore obtain the following claim:

Claim A.9. *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S'_π of 2^κ vertex-disjoint paths such that each vertex $(0,w)$ in $\mathrm{BN}^{\ddagger 6}_\kappa$ is connected to $(6\kappa+1,\pi(w))$. Moreover, S'_π can be found in $O(\kappa\cdot 2^\kappa)$ time and space.*

Proof. Invoking Theorem A.7, we know that we can find (efficiently) a set of paths S_π routing π on a κ-dimensional Beneš network. We show how to map the 2^κ vertex-disjoint paths S_π over $\textsc{Beneš}_\kappa$ to 2^κ vertex-disjoint paths S'_π over $\mathrm{BN}^{\ddagger 6}_\kappa$, by appropriately replacing the "second half" of each path p in S_π (that is the part of the path that would have traveled through the reversed butterfly network, which needs to be simulated).

Specifically, for each path p in S_π write $p = p_1p_2$ by "splitting" the path in half; note that p_2 uses the reversed butterfly network. Let w and w' be the starting vertex and ending vertex in the path p_2. Our strategy is to replace p_2 with a new (longer) path p'_2, over five butterfly networks, that is equivalent to p_2 as far as routing is concerned. Concretely, let $\tilde{p}_2$ be the path obtained by taking the image of p_2 under ϕ_κ (the graph isomorphism from $\mathrm{BN}^{\mathrm{r}}_\kappa$ to BN_κ); then set $p'_2 := q_a\tilde{p}_2q_b$ where q_a and q_b are respectively the paths (each over two butterfly networks) used to route w and $(w')^{\mathrm{r}}$ according to the bit-reversal permutation obtained via Claim A.8.

It is then easy to verify that the path $p' = p_1p'_2$ has the same "end points" as p but, instead of using a Beneš network, uses six butterfly networks connected in tandem. Note that the collection S'_π of paths p' obtained as above, each from a path p in S_π, is indeed vertex-disjoint (as each of the three "segments" q_a, $\tilde{p}_2$, and q_b of p'_2 were picked without replacement from vertex-disjoint sets of paths) and routes π.

Finally, the efficiency guarantees of Theorem A.7 and Claim A.8, as well as the efficiency of computing ϕ_κ, easily imply an algorithm with the claimed efficiency. □

In fact, we can improve Claim A.9 to use only *four* butterfly networks connected in tandem, by simply omitting the bit reversal permutation carried out by the last two butterfly networks, by simply routing a different permutation π' obtained from π which *already* reverses the bits of the output of π.

Claim A.10. *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S'_π of 2^κ vertex-disjoint paths such that each vertex $(0,w)$ in $\mathrm{BN}^{\ddagger 4}_\kappa$ is connected to $(4\kappa+1,\pi(w))$. Moreover, S'_π can be found in $O(\kappa\cdot 2^\kappa)$ time and space.*

Proof. We can modify the proof of Claim A.9 by routing the permutation $\pi' := \mathrm{br}_\kappa \circ \pi$ on the first four butterfly networks and neglecting to use the last two butterfly networks to "undo" the bit reversal over rows induced by the graph isomorphism ϕ_κ. □

A.5 De Bruijn Graphs and Their Rearrangeability

We finally deduce the fact that four De Bruijn graphs connected in tandem form a rearrangeable network.

Claim A.11. *Let κ be a positive integer and $\pi\colon \{0,1\}^\kappa \to \{0,1\}^\kappa$ a permutation. There exists a set S''_π of 2^κ vertex-disjoint paths such that each vertex $(0,w)$ in $\mathrm{DB}^{\ddagger 4}_\kappa$ is connected to $(4\kappa+1,\pi(w))$. Moreover, S''_π can be found in time and space $O(\kappa\cdot 2^\kappa)$ or parallel time $O(\kappa^2)$.*

Proof. From Claim A.5 (and the comment after it), we deduce that there is a graph isomorphism from $\mathrm{BN}^{\ddagger 4}_\kappa$ to $\mathrm{DB}^{\ddagger 4}_\kappa$ that preserves the order of the first and last column. Therefore, in order to find the desired set of paths S''_π we can simply take the image under the graph isomorphism of each path in the set S'_π guaranteed by Claim A.10. The efficiency of the algorithm follows from the efficiency guarantees of Claim A.5 and the efficiency of computing the graph isomorphism. Parallelism is preserved. □

How to route with only $3\kappa + \lceil \kappa/2 \rceil$ ("*three and a half*") De Bruijn graphs connected in tandem?

To begin with, we note that we can already "save a column": in light of the explicit algorithm in the proof of Theorem A.7, we could have defined Beneš networks (Definition A.6) to only have 2κ columns (by avoiding the repeated "minimal cross" at the juncture of the butterfly network and the reversed butterfly network), and the Beneš network routing algorithm could have still been made to work.

Then, to save a half of a De Bruijn, we do as follows:

1. Find first a set of paths routing the given permutation on the newly defined Beneš network.
2. Use the first $\kappa + 1$ De Bruijn columns (i.e., column 0 through κ) to hold the partial paths on the first $\kappa + 1$ Beneš columns.
3. Then use De Bruijn columns κ through $2\kappa + \lceil \kappa/2 \rceil$ to route a bit-reversing permutation, following Claim A.8.
4. Then use De Bruijn columns $2\kappa + \lceil \kappa/2 \rceil$ through $3\kappa + \lceil \kappa/2 \rceil - 1$ to hold the partial paths on the Beneš columns $\kappa + 1$ through $2\kappa - 1$.

Overall we have used $3\kappa + \lceil \kappa/2 \rceil$ De Bruijn columns.

B Circuit Diagrams[17]

Let $M = \langle w, k, \mathbb{A}, \mathbb{C} \rangle$ be a random-access machine. (See definitions in Section 6.4.) Let us recall and define some notation: w is the register width, k is the number of registers, $(1+k)w$ is the size of a configuration, n is the number of instructions, u is the width of an opcode. We provide details for the circuits we encounter in this paper:

- in Section B.1, we give the circuit for the transition function δ_M of M, and
- in Section B.2, we give the circuit for the coloring constraint function obtained in the reduction from $\mathrm{BH}_{\mathsf{RAM}}(M)$ to sGCP from Section 7.

Notation. We use AND, OR, XOR, and NOT gates, as well as standard components such as MUX, DEMUX, and CMP (comparator). For CMP taking two input wires x and y, we label the output by "=", ">", or "<", to mean that the output is 1 if $x = y$, $x > y$, and $x < y$ respectively. Also, IF a circuit that, on input a decision bit b, decides to output one of two input bits x or y.

B.1 Transition Function

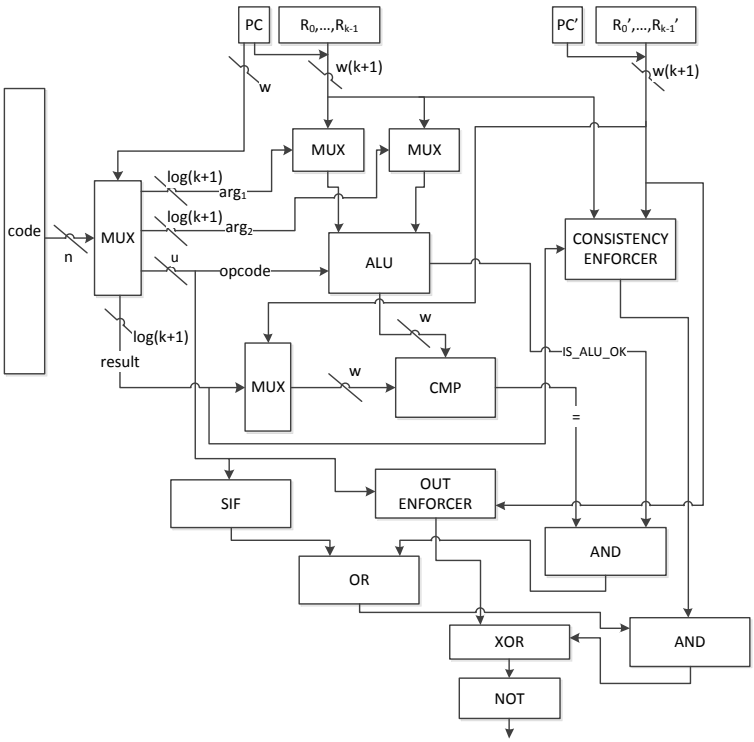


Figure 6: High-level view of a circuit for δ_M, the **transition function** of M. (See Definition 6.12.) There are three main "modules": the *consistency enforcer* is given in Figure 7, the *out enforcer* is given in Figure 8, and the *special-instruction flag* (SIF) is given in Figure 9. We do not provide a circuit for the arithmetic logic unit (ALU).

[17] We are grateful to Ohad Barta for carefully reviewing and providing corrections to the circuits of this section.

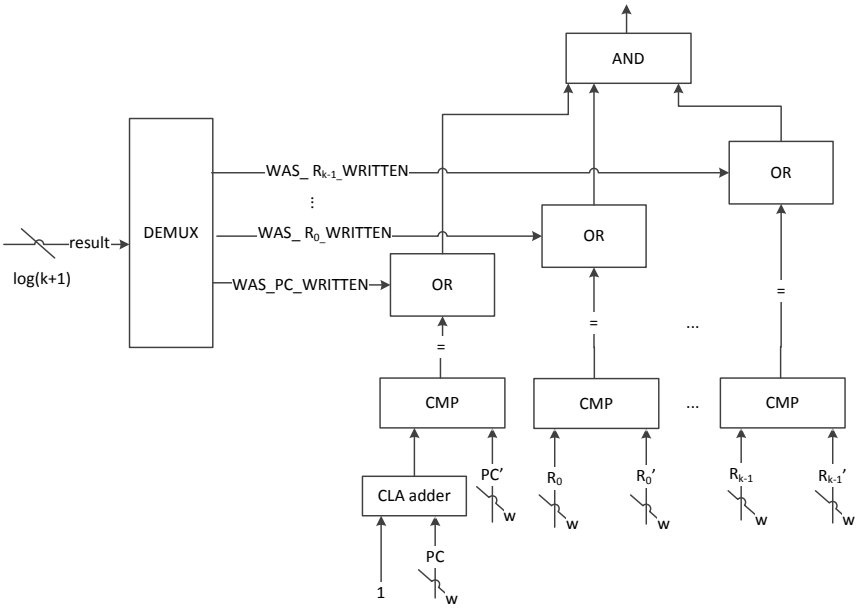


Figure 7: The **consistency enforcer** outputs 1 if and only if all the registers but the destination register are the same.

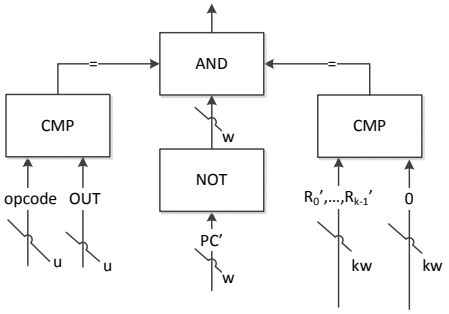


Figure 8: The **out enforcer** outputs 1 if and only if the opcode is out, $\mathsf{pc}' = 0$, and all the primed registers are 0.

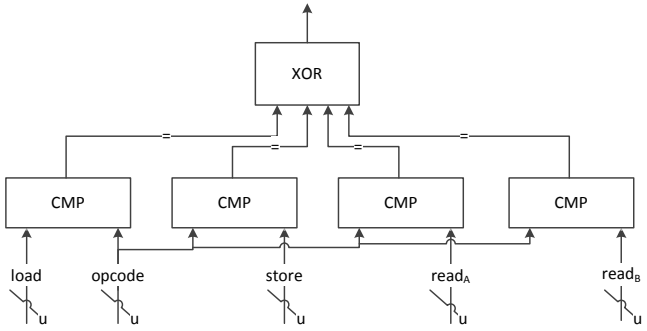


Figure 9: The **special-instruction flag** (SIF) outputs 1 if and only if the opcode is one of the "special" instructions (namely, read_A, read_B, load, and store).

B.2 Coloring Constraint Function For sGCP

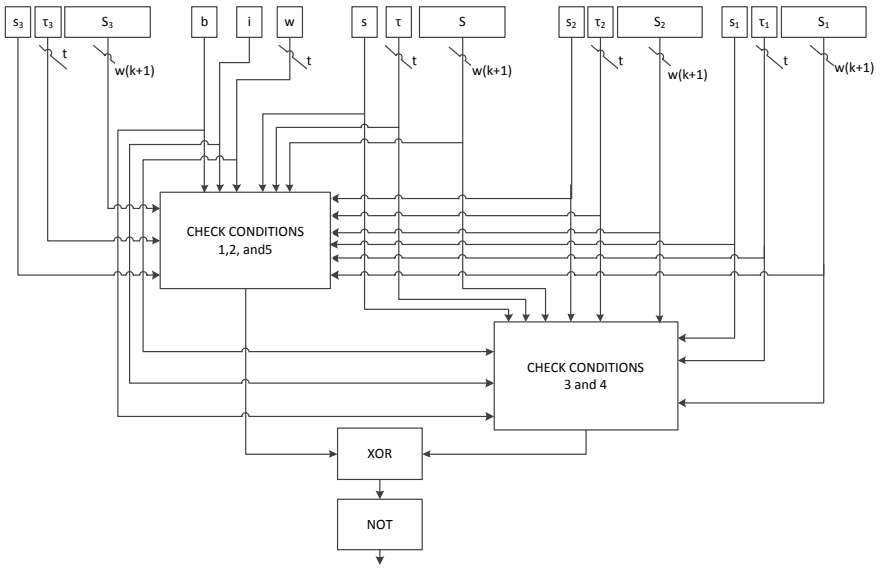


Figure 10: High-level view of a circuit for K_t (see Equation 3), i.e., the above circuit outputs 1 if and only if the colors in the neighborhood of a vertex (b, i, w) in an extended De Bruijn graph satisfy the requirements of Definition 7.12. The circuit has two main components, the first checking items 1, 2, and 5 of the definition (given in Figure 11), and the second checking items 3 and 4 of the definition (given in Figure 12).

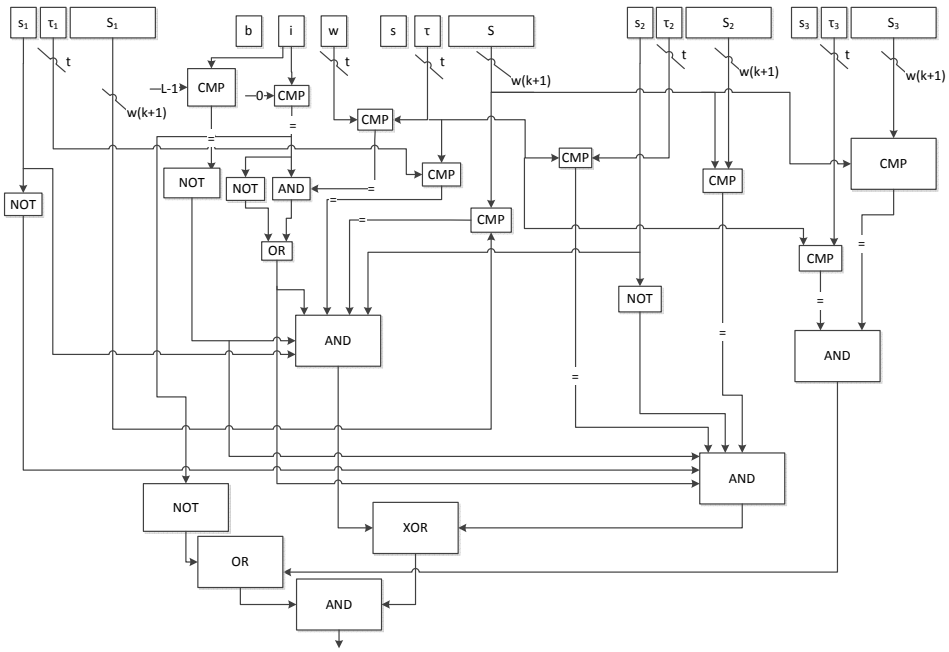


Figure 11: Outputs 1 if and only if two colors of neighbors in an extended De Bruijn graph satisfy conditions 1, 2, and 5 of the validity part in Definition 7.12.

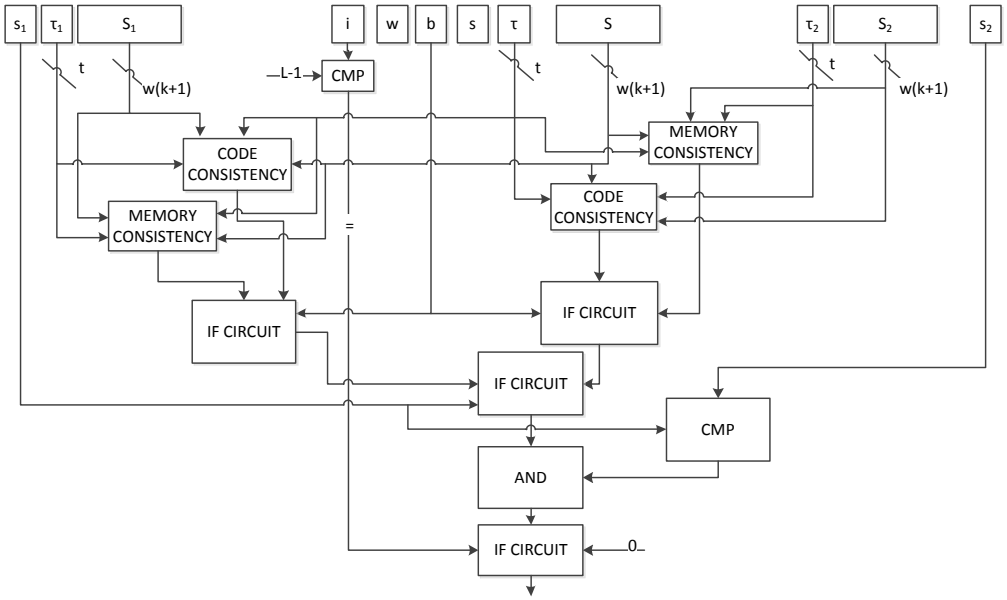


Figure 12: Outputs 1 if and only if two colors of neighbors in an extended De Bruijn graph satisfy conditions 3 and 4 of the validity part in Definition 7.12. The modules of *code consistency* and *memory consistency* are respectively given in Figure 13 and Figure 14.

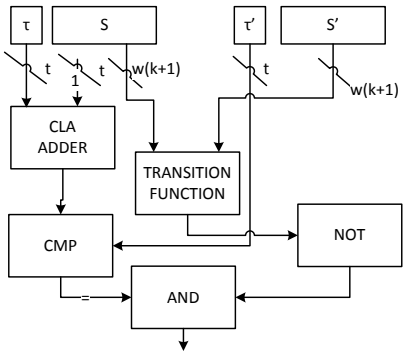


Figure 13: This circuit outputs 1 if and only if $\tau' = \tau + 1$ and $S \rightsquigarrow S'$.

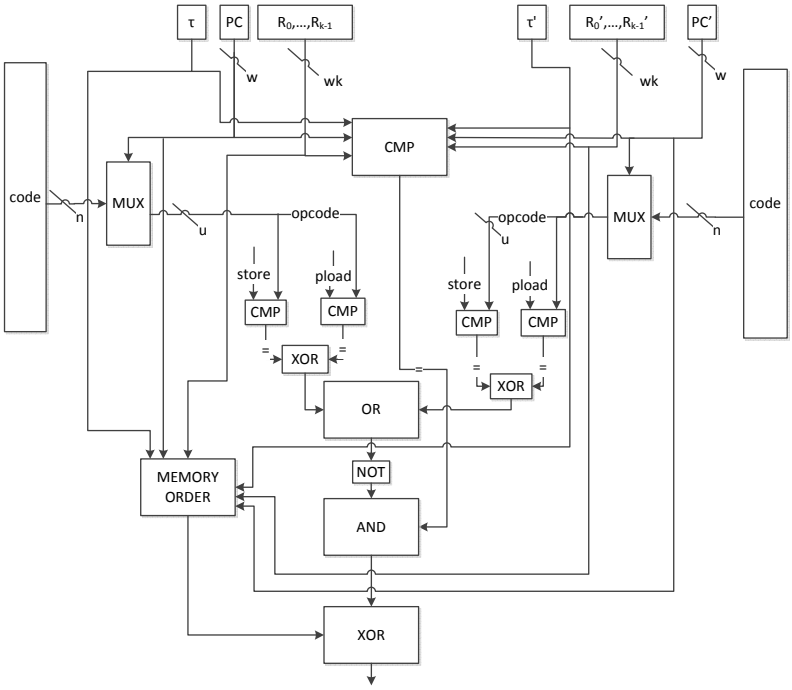


Figure 14: Outputs 1 if and only if both S and S' contain memory instructions and (τ', S) precedes (τ'', S') in memory (see Definition 7.2) or $(\tau'' = \tau' \wedge S' = S)$. The *memory-order* module is given in Figure 15.

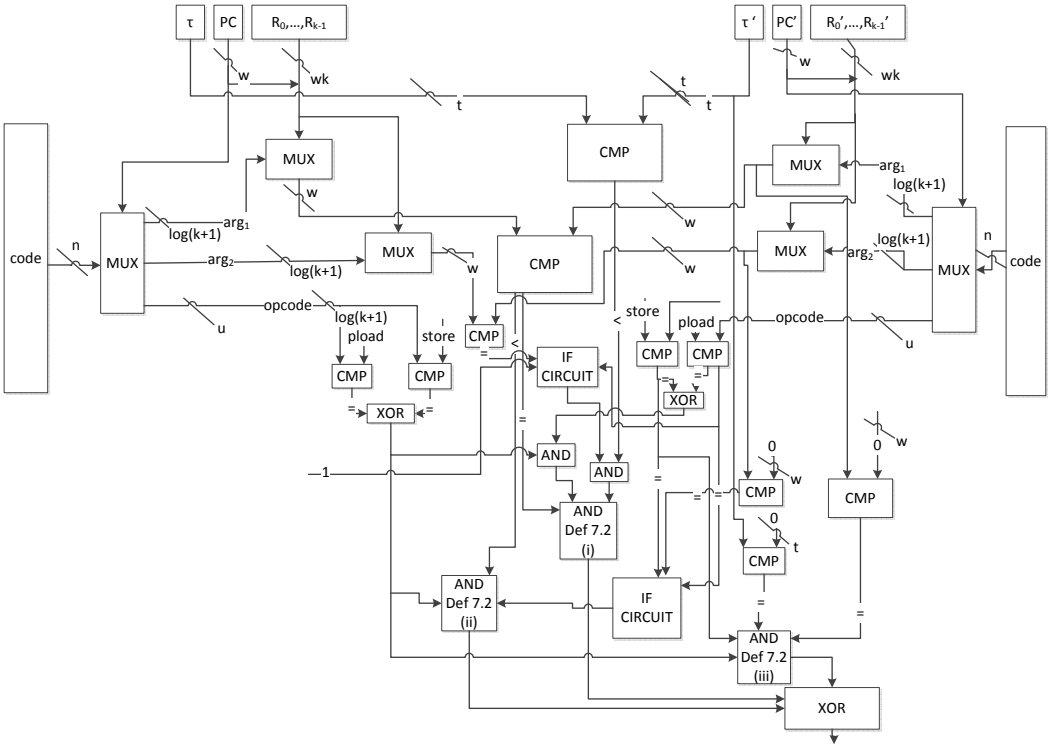


Figure 15: Outputs 1 if and only if the two input configurations are memory ordered as defined in Definition 7.2.

C Finite Fields and Efficient Computation

We review some computational properties of finite fields; we also develop a number of algorithmic results that are crucial for "arithmetizing" graph coloring problems in Section 8 in a way that ensures that certain high-degree polynomials have small arithmetic circuits computing them.

Throughout, *linearized polynomials* (see [LN97, Section 2.5]) will play a crucial role both for speeding up further already efficient computations (e.g., faster interpolation and evaluation algorithms in Section C.3 and Section C.4) as well as for ensuring that certain high-degree polynomials can be computed efficiently (in Section C.6 and Section C.7).

Whenever possible, we state results for a generic field characteristic p; we will be explicit when we must take $p = 2$ (which ultimately is the special case of our interest). Also, whenever we make statements about the time or space complexity of an algorithm, we will make the simplifying assumption that basic field operations can be performed in unit time and field elements can be stored at unit cost.

C.1 Irreducible and Primitive Polynomials

We represent elements of a finite field as polynomials modulo an irreducible polynomial of the appropriate degree. Specifically, to represent elements of $\mathbb{F}_q$, where $q = p^f$ and p is the characteristic of $\mathbb{F}_q$: we consider any *irreducible* polynomial I over $\mathbb{F}_p$ of degree f; I has a root x in $\mathbb{F}_q$ and thus $\mathbb{F}_q = \mathbb{F}_p(\mathsf{x})$, so that every element of $\mathbb{F}_q$ can be uniquely expressed as a polynomial in x over $\mathbb{F}_p$ of degree less than f. See [LN97, Section 2.5] for more details. In particular, this representation will allow us to perform field operations in time that is polylogarithmic in the field size and space that is logarithmic in the field size.

Irreducible polynomials of a given degree over a finite field can be found deterministically, in polynomial time if the characteristic is small [Sho88, Corollary 3.2]:

Theorem C.1 (Finding Irreducible Polynomials)**.** *There exists a deterministic algorithm* FindIrrPoly *that, on input $(p, 1^f)$ where p is a prime and f is a positive integer, computes an irreducible polynomial I of degree f over $\mathbb{F}_p$ in time $\mathsf{poly}(p, f)$. Specifically, there exists a universal constant $c > 0$ such that the running time of* FindIrrPoly$(p, 1^f)$ *is*

$$O\Big(p^{1/2+c} f^{3+c} + (\log p)^2 f^{4+c}\Big) \ ;$$

in particular, FindIrrPoly(1^f) := FindIrrPoly$(2, 1^f)$ *runs in polynomial time. (For convenience, we denote by t_{Irr} the time complexity of this algorithm.)*

We will only deal with (finite) fields of characteristic 2, so, for simpliciy, every time we invoke FindIrrPoly, we will leave implicit the first input $p = 2$, and write only FindIrrPoly(1^f).

Next, we recall the definition of a *primitive* polynomial; for more details about primitive polynomials, see [LN97, Section 3.1].

Definition C.2. *Let p be a prime and ℓ a positive integer. A polynomial Ξ of degree ℓ over $\mathbb{F}_p$ is* **primitive** *over $\mathbb{F}_p$ if Ξ is the minimal polynomial over $\mathbb{F}_p$ of a primitive element of $\mathbb{F}_{p^\ell}$.*

Finding primitive polynomials is not known to be easy:

Theorem C.3 (Finding Primitive Polynomials)**.** *There exists a deterministic algorithm* FindPrimPoly *that, on input $(p, 1^\ell)$ where where p is a prime and ℓ is a positive integer, computes a primitive polynomial Ξ of degree ℓ over $\mathbb{F}_p$ in time $\mathsf{poly}(p^\ell)$. We define* FindPrimPoly(1^ℓ) := FindPrimPoly$(2, 1^\ell)$. *(For convenience, we denote by t_{PRIM} time complexity of this algorithm.)*

Proof. Shoup [Sho99] shows how to compute a minimal polynomial of degree ℓ in time $\mathsf{poly}(\ell)$. Hence, the "hard" part is to find a primitive element of $\mathbb{F}_{p^\ell}$. Shparlinski [Shp96] shows that a primitive element of $\mathbb{F}_{p^\ell}$ can be found in time approximately $O(p^{\ell/4})$. □

Nonetheless, we will usually be interested in primitive polynomials of low degree, so the inefficiency of finding them will not matter. (And, in practice, one always relies on pre-computed tables anyways; see Remark C.6.)

We use primitive polynomials in Section 8 to create "artificial" cyclic groups inside a finite field of characteristic 2, in order to embed a certain graph into an affine graph over the finite field. The ability to create cyclic structure is given by the following claim:

Claim C.4. *Let ℓ be a positive integer, and let Ξ be a primitive polynomial of degree ℓ over $\mathbb{F}_2$. Then, for any non-negative integers i and j, x^i and x^j are congruent modulo the polynomial Ξ if and only if i and j are congruent modulo $2^\ell - 1$.*

Proof. Recall the following fact:

> **Theorem** ([LN97, Theorem 3.18])**.** *Let p be a prime. The monic polynomial P of positive degree m over $\mathbb{F}_p$ is a primitive polynomial over $\mathbb{F}_p$ if and only if $(-1)^m P(0)$ is a primitive element of $\mathbb{F}_p$ and the least positive integer r for which x^r is congruent modulo P to some element of $\mathbb{F}_p$ is $r = \frac{p^m-1}{p-1}$. In case P is primitive over $\mathbb{F}_q$, we have $x^r \equiv (-1)^m P(0) \bmod P(x)$.*

We invoke the above theorem with $p := 2$, $P := \Xi$, and $m := \ell$. We obtain that $r = 2^\ell - 1$. Moreover, since Ξ is primitive, $\Xi(0) \neq 0$ (by [LN97, Theorem 3.16]) and thus, over a field of characteristic 2, we have that $(-1)^m \Xi(0) = \Xi(0) = 1$. We deduce that $1, x, \ldots, x^{2^\ell - 2} \not\equiv 0, 1 \bmod \Xi(x)$ and $x^{2^\ell - 1} \equiv 1 \bmod \Xi(x)$, thereby proving the claim. □

The above claim gives us the following simple corollary:

Corollary C.5. *Let ℓ and Ξ as in Claim C.4. Define $\xi_i(x) := x^i \bmod \Xi(x)$ for $i = 0, \ldots, 2^\ell - 2$. Then, for any $i \in \{0, \ldots, 2^\ell - 2\}$ and positive integer c,*

$$\xi_{(i+c \bmod (2^\ell - 1))}(x) = x^c \cdot \xi_i(x) + Q(x) \cdot \Xi(x) \ ,$$

where $Q(x)$ is the (unique) polynomial quotient in $\mathbb{F}_2[x]$ when dividing $x^c \cdot \xi_i(x)$ by $\Xi(x)$.

Proof. The corollary easily follows from the definition of the $\xi_0(x), \ldots, \xi_{(2^\ell - 2)}(x)$, Claim C.4, and the definition of congruence under division of univariate polynomials:

$$\begin{aligned}
\xi_{(i+c \bmod (2^\ell-1))}(x) &= x^{(i+c \bmod (2^\ell-1))} \mod \Xi(x) && \text{(by definition of } \xi_{(i+c \bmod (2^\ell-1))}(x)) \\
&\equiv x^{i+c} \mod \Xi(x) && \text{(since } x^i \equiv x^j \bmod \Xi(x) \Leftrightarrow i \equiv j \bmod (2^\ell - 1)) \\
&\equiv x^c \cdot x^i \mod \Xi(x) \\
&\equiv x^c \cdot \xi_i(x) \mod \Xi(x) \ ,
\end{aligned}$$

so that $\xi_{(i+c \bmod (2^\ell-1))}(x) = x^c \cdot \xi_i(x) + Q(x) \cdot \Xi(x)$, as desired. □

Remark C.6. Ultimately we are interested in practical implementations of the Levin reductions discussed in this paper. In practice, one keeps tables of irreducible and primitive polynomials, so that we will not worry much about the time needed to compute such polynomials of the correct degree.

C.2 Linear Maps and Sparse Polynomials

Ben-Sasson et al. [BSGH$^+$05, Section 5] discuss the computational advantages of working with linear subspaces of finite fields; while some of the underlying *algebraic* facts were already used in [BSS08] and [BSGH$^+$06], their *computational* properties were only used in [BSGH$^+$05], where they are critical to argue for an efficient verifier.

In this section, we recall some of the results of Ben-Sasson et al. [BSGH$^+$05, Section 5], and complement them with further details and new results altogether (which ultimately are needed for an explicit description of an efficient PCP verifier, as constructed by [BSCGT12]). For a more in-depth discussion of the theory of linear algebra for vector spaces over finite fields, see [LN97, Chapter 3.4].

Throughout this section, we let $\mathbb{B} \subseteq \mathbb{F}$ be two fields of sizes $|\mathbb{B}| = p$ and p^f respectively. For a linear subset $H \subseteq \mathbb{F}$ of dimension h over the (smaller) field $\mathbb{B}$ (i.e., there is a basis $(\alpha_1, \ldots, \alpha_h)$ of elements in $\mathbb{F}$ such that every element of H can be expressed as $\sum_{i=1}^{h} c_i \alpha_i$ with $c_1, \ldots, c_h \in \mathbb{B}$), we say that a function $g\colon H \to \mathbb{F}$ is a $\mathbb{B}$-*linear map* if $g(ax + by) = ag(x) + bg(y)$ for every $x, y \in H$ and $a, b \in \mathbb{B}$.

A first observation is that $\mathbb{B}$-linear maps can be represented as sparse low-degree polynomials. (In other words, they have sparse *low-degree extensions*; see Section C.4.)

Claim C.7 ([BSGH$^+$05, Proposition 5.1])**.** *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the (smaller) field $\mathbb{B}$ and $g\colon H \to \mathbb{F}$ a $\mathbb{B}$-linear map. Then there exists a (unique) polynomial $\hat{g}\colon \mathbb{F} \to \mathbb{F}$ of the form*

$$\hat{g}(x) = \sum_{i=0}^{h-1} c_i x^{p^i} \quad ,$$

where $c_0, \ldots, c_{h-1} \in \mathbb{F}$, such that $\hat{g}$ agrees with g on all of H. Moreover, given the evaluations of g on any basis $\mathcal{B}_H$ for H, the coefficients $c_0, \ldots, c_{h-1}$ can be computed with $\mathsf{poly}(h, \log p)$ arithmetic operations over $\mathbb{F}$.

Remark C.8. By inspecting the proof of Claim C.7, we see that the $\mathbb{B}$-linearity of f implies that the coefficients $c_0, \ldots, c_{h-1}$ are the (unique) solution to the following linear system: letting $\mathcal{B}_H = (e_1, \ldots, e_h)$,

$$\begin{pmatrix} e_1 & e_1^p & e_1^{p^2} & \cdots & e_1^{p^{h-1}} \\ e_2 & e_2^p & e_2^{p^2} & \cdots & e_2^{p^{h-1}} \\ \vdots & \vdots & \ddots & \vdots & \\ e_h & e_h^p & e_h^{p^2} & \cdots & e_h^{p^{h-1}} \end{pmatrix} \cdot \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{h-1} \end{pmatrix} = \begin{pmatrix} g(e_1) \\ g(e_2) \\ \vdots \\ g(e_h) \end{pmatrix} \quad . \tag{11}$$

Hence, given $g(e_1), \ldots, g(e_h)$, the coefficients $c_0, \ldots, c_{h-1}$ can indeed be found using $O(h^2 \log p + h^3)$ arithmetic operations over $\mathbb{F}$, because finding the entries of the matrix from $\mathcal{B}_H$ requires $O(h^2 \log p)$ arithmetic operations via repeated squaring, and then $O(h^3)$ arithmetic operations are needed for Gaussian elimination. Later, in Section C.4.4, we shall discuss an $O(h^2 \log h \cdot \log p)$ recursive algorithm for finding the coefficients $c_0, \ldots, c_{h-1}$.

Once the coefficients $c_0, \ldots, c_{h-1}$ have been computed, however, evaluating $\hat{g}$ at a given point $\alpha \in \mathbb{F}$ is a *very simple* arithmetic circuit, involving only the use of repeated squaring to find $\alpha, \alpha^p, \ldots, \alpha^{p^{h-1}}$, multiplying each α_i by the respective coefficient c_i, and adding up the results.

A very important class of polynomials are *vanishing polynomials* [BSGH$^+$05, Definition 5.2]:

Definition C.9. *For any subset S of $\mathbb{F}$, the* **S-vanishing polynomial in $\mathbb{F}$**, *denoted $Z_S(x)$, is defined to be the polynomial in $\mathbb{F}[x]$ whose zeros are precisely the elements of S, that is,*

$$Z_S(x) = \prod_{s \in S} (x - s) \ .$$

It is easy to see that when S is a vector space over the base field $\mathbb{B}$, then $Z_S \colon \mathbb{F} \to \mathbb{F}$ is a $\mathbb{B}$-linear map, and we call Z_S a *subspace polynomial*:

Claim C.10 ([BSGH+05, Proposition 5.3])**.** *If S is a vector space over the base field $\mathbb{B}$ then $Z_S \colon \mathbb{F} \to \mathbb{F}$ is a $\mathbb{B}$-linear map, that is,*

- *for all $u, v \in \mathbb{F}$, $Z_S(u + v) = Z_S(u) + Z_s(v)$, and*
- *for all $a \in \mathbb{B}$ and $v \in \mathbb{F}$, $Z_S(a \cdot v) = a \cdot Z_S(v)$.*

Because of the above properties, a subspace polynomial is also known as a *linearized polynomial*. In particular, one can show, using Claim C.7, that, whenever S is a vector space over the base field $\mathbb{B}$, Z_S is sparse and its coefficients can be found fast:

Claim C.11 ([BSGH+05, Proposition 5.4])**.** *If S is a d-dimensional vector space over the base field $\mathbb{B}$, then there exist $c_1, \ldots, c_{d-1} \in \mathbb{F}$ such that*

$$Z_S(x) = x^{p^d} + \sum_{i=0}^{d-1} c_i x^{p^i} \ .$$

Moreover, the coefficients $c_0, \ldots, c_{d-1}$ can be computed with $\mathsf{poly}(d, \log p)$ arithmetic operations over $\mathbb{F}$, when given as input a basis $\mathcal{B}_S$ for S.

Example C.1. There are important special cases where finding a subspace polynomial is *very* easy (and one need not even solve any linear system to find the coefficients of the polynomial):

- If $S = \mathbb{F}$, then $Z_S(x) = x^{p^f} - x$.
- If p divides f, then $Z_S(x) = x^{p^{f/p}} - x$.

Furthermore, if the characteristic of the field is $p = 2$, then $x^{p^{f/p}} - x = x^{p^{f/p}} + x$, which is the field trace function from $\mathbb{F}_{p^f}$ to $\mathbb{F}_{p^{f/p}}$.

Whenever possible, we will try to work with such special cases, to take advantage of "super-sparse" subspace polynomials, and thus greatly speed up computations.

Remark C.12. For notational convenience, we denote by FindSubspPoly the algorithm that, on input (an irreducible polynomial representing) $\mathbb{F}$ and a basis $(e_1, \ldots, e_d)$ for S, outputs an arithmetic circuit $[Z_S]^{\mathsf{A}}$ for computing Z_S; note that $[Z_S]^{\mathsf{A}}$ has size $O(d)$.

Since Claim C.11 relies on Claim C.7, FindSubspPoly could in principle require $O(h^2 \log p + h^3)$ field operations (see Remark C.8). Furthermore, we are not "allowed" to benefit from the even faster $O(h^2 \log h \cdot \log p)$-time $O(h)$-space algorithm mentioned in Remark C.8 (and discussed in Section C.4.4) for finding the desired coefficients in Claim C.7, because FindSubspPoly is a subroutine of that faster algorithm.

Indeed, in this special case, there is instead a simple recursive algorithm, which requires only $O(d^2 \log p)$ running time and $O(d \log p)$ space.

Specifically, first note that, for $d > 1$,

$$\begin{aligned}
Z_S(x) &= Z_{\mathrm{span}(e_1,\ldots,e_d)}(x) \\
&= \prod_{i=1}^{p} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})+(i-1)e_d}(x) \\
&= \prod_{i=1}^{p} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x-(i-1)e_d) \\
&= \prod_{i=1}^{p} \left(Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) - (i-1)\cdot Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)\right) \\
&= \prod_{i=1}^{p} \left(Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) + (i-1)\cdot Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)\right) \qquad \text{(by rearranging)} \\
&= \sum_{i=1}^{p} \left(\sum_{\substack{r_1<\cdots<r_i \\ r_1,\ldots,r_i\in\{1,\ldots,i\}}} r_1\cdots r_i \right) Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)^{p-i} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x)^i \\
&= Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x)^p + (p-1) Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)^{p-1} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) \qquad \text{(since } p \text{ is prime)} \\
&= Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x^p) + (p-1) Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(e_d)^{p-1} Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}(x) \ . \qquad (12)
\end{aligned}$$

Therefore, the following algorithm computes the coefficients for $Z_S(x)$: on input an irreducible polynomial I for representing $\mathbb{F}$ and the basis $(e_1, \ldots, e_d)$ for S,

FindSubspPoly$(I, e_1, \ldots, e_d) \equiv$

1. If $d = 1$, output $x^p + (p-1)e_1^{p-1}x$.
2. If $d > 1$, do the following:
   (a) Run FindSubspPoly$(e_1, \ldots, e_{d-1})$ to generate $[Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}]^{\mathsf{A}}$.
   (b) Using $[Z_{\mathrm{span}(e_1,\ldots,e_{d-1})}]^{\mathsf{A}}$, compute $[Z_{\mathrm{span}(e_1,\ldots,e_d)}]^{\mathsf{A}}$ by following Equation 12.
   (c) Output $[Z_{\mathrm{span}(e_1,\ldots,e_d)}]^{\mathsf{A}}$.

The correctness of FindSubspPoly easily follows from the derivation of Equation 12 above, and its time complexity of $O(d^2 \log p)$ and space complexity of $O(d \log p)$ easily follows from its simple recursive structure.

C.3 Polynomial Evaluation

The problem of *polynomial evaluation* is the following:

- **input:** a field $\mathbb{F}$, a subset $S \subseteq \mathbb{F}$, and a polynomial $P(x) \in \mathbb{F}[x]$ of degree at most $|S| - 1$;
- **output:** a function $p\colon S \to \mathbb{F}$ such that $p(\alpha) = P(\alpha)$ for every $\alpha \in S$; p is known as the *evaluation* of P over S.

Note that the problem of polynomial evaluation easily generalizes to finding multi-variate evaluations of multi-variate polynomials; this generalization will not be of interest to us for the purpose of constructing Levin reductions (though will briefly arise in [BSCGT12] for optimizing the speed of the PCP prover).

A naïve evaluation of a polynomial of degree d at a point takes $O(d^2)$ field operations; using Horner's method, only $O(d)$ field operations are required. Still, if d is on the order of $|S|$, then evaluation of the polynomial over S would take $O(|S|^2)$ field operations.

Not surprisingly better algorithms are known:

Theorem C.13 ([vzGG03, Corollary 10.8]). *A polynomial evaluation over S of a polynomial of degree at most $|S|-1$ can be computed in $O(\mathsf{M}(|S|)\log|S|)$ field operations, where $\mathsf{M}(n)$ is the time to multiply two polynomials of degree at most n. (Recall that, ithout additional assumptions on S, the best upper bound on $\mathsf{M}(n)$ is $O(n^{\log 3}) \approx O(n^{1.585})$, via Karatsuba's algorithm.)*

Fortunately, in the applications that we have in mind, the sets S in which we will be interested do satisfy additional properties: they will be linear subsets of the field. Therefore, we will be able to benefit from faster evaluation algorithms that use additive FFT methods to obtain great *quasilinear* running times.

Specifically, in the special case (of our interest) where $\mathbb{F} = \mathbb{F}_{2^f}$ for some $f \in \mathbb{N}$ and S is a linear subset of $\mathbb{F}$, much faster algorithms are known. This is the problem of *polynomial evaluation over linear subsets*.

For example, the additive FFT of the von zur Gathen and Gerhard [vzGG96] has complexity $O(|S|(\log|S|)^2)$; this algorithm derives its speed from the use of linearized polynomials. (This algorithm is given explicitly in [BSCGT12].)

Algorithms with even better asymptotic complexity are known; see for example [Mat08, Chapters 3]. Nonetheless, the von zur Gathen and Gerhard additive FFT can be implemented very easily and does not "hide" any big constants in its practical running time.

We note that Bhattacharyya [Bha05, Section 2.1] also developed a fast algorithm for evaluation over linear subsets, but his algorithm in practice performs worse than the von zur Gathen and Gerhard additive FFT.

C.4 Polynomial Interpolation

The problem of *polynomial interpolation* is the following:

- **input:** a field $\mathbb{F}$, a function $p\colon S \to \mathbb{F}$ with $S \subseteq \mathbb{F}$;
- **output:** a polynomial $P(x) \in \mathbb{F}[x]$ of degree at most $|S|-1$ that agrees with p on S; P is known as the *low-degree extension* of p in $\mathbb{F}$.

Note that the problem of polynomial interpolation easily generalizes to finding multi-variate low-degree extensions of multi-variate functions.

In this section we discuss computational properties of the problem of polynomial interpolation, especially in some special cases that play a particularly important role in the applications that we consider.

C.4.1 Existence and uniqueness of low-degree extensions

We begin by recalling that low-degree extensions exist and are unique; for completeness, we shall review the proof of this fact.

Theorem C.14. *Let $\mathbb{F}$ be a finite field, m a positive integer, $s_1,\ldots,s_m$ positive integers, and $S_1,\ldots,S_m$ subsets of $\mathbb{F}$ with respective cardinalities $s_1,\ldots,s_m$.*

For any function $f\colon S_1\times\cdots\times S_m \to \mathbb{F}$ there exists a unique m-variate polynomial $\hat{f}\colon \mathbb{F}^m \to \mathbb{F}$ such that the following two conditions are satisfied:

(i) Low Degree: *for $i=1,\ldots,m$, the degree of $\hat{f}$ in the i-th variable is less than s_i, and*

(ii) Consistency: *$\hat{f}$ agrees with f on $S_1\times\cdots\times S_m$, that is, for every $(\alpha_1,\ldots,\alpha_m)\in S_1\times\cdots\times S_m$, it is the case that $\hat{f}(\alpha_1,\ldots,\alpha_m) = f(\alpha_1,\ldots,\alpha_m)$.*

We call $\hat{f}$ the **low-degree extension of f in $\mathbb{F}$**, *and denote it* $\mathrm{LDE}_{\mathbb{F},m,(S_1,\ldots,S_m)}(f)$.

In other words, the low-degree extension of a function is simply a polynomial that has the function "embedded" into it; moreover, this polynomial is unique.

The first step in the proof of Theorem C.14 is to establish that the low-degree extension of the zero function (which certainly exists) is unique:

Lemma C.15. *Let $\mathbb{F}$, m, $s_1, \ldots, s_m$, and $S_1, \ldots, S_m$ be as in Theorem C.14, and let $z\colon S_1 \times \cdots \times S_m \to \mathbb{F}$ be the zero function on $S_1 \times \cdots \times S_m$. Then $\mathrm{LDE}_{\mathbb{F},m,(S_1,\ldots,S_m)}(z)$ is the unique (m-variate) identically-zero polynomial over $\mathbb{F}$.*

Proof. By induction on m. In the case $m = 1$, the lemma follows from the fact that a univariate polynomial with positive degree d cannot have more than d roots: indeed, since the polynomial $\mathrm{LDE}_{\mathbb{F},1,(S_1)}(z)$ is required to vanish on all s_1 elements of S_1, and yet have degree less than s_1, it must be the case that $\mathrm{LDE}_{\mathbb{F},1,(S_1)}(z)$ is the (unique) identically-zero polynomial. In the case $m > 1$, assume the lemma holds for all positive integers m' less than m, and consider any m-variate polynomial $\hat{z}$ that is a low-degree extension of z in $\mathbb{F}$. Let α be any element in $\mathbb{F}$, and consider the $(m-1)$-variate polynomial $\hat{z}_\alpha$ over $\mathbb{F}$ defined by $\hat{z}_\alpha(x_2, \ldots, x_m) := \hat{z}(\alpha, x_2, \ldots, x_m)$. By the inductive assumption, $\hat{z}_\alpha$ is the unique $(m-1)$-variate identically-zero polynomial over $\mathbb{F}$, because it is a low-degree extension in $\mathbb{F}$ of the function z_α, where $z_\alpha(x_2, \ldots, x_m) := z(\alpha, x_2, \ldots, x_m)$. Next, let $\vec{\beta} = (\beta_2, \ldots, \beta_m)$ be any element in $\mathbb{F}^{m-1}$, and consider the univariate polynomial $\hat{z}_{\vec{\beta}}$ defined by $\hat{z}_{\vec{\beta}}(x_1) := \hat{z}(x_1, \beta_2, \ldots, \beta_m)$. Again, by the inductive assumption, $\hat{z}_{\vec{\beta}}$ is the unique univariate identically-zero polynomial over $\mathbb{F}$, because it is a low-degree extension in $\mathbb{F}$ of the function $z_{\vec{\beta}}$, where $z_{\vec{\beta}}(x_1) := z(x_1, \beta_2, \ldots, \beta_m)$. We conclude that $\hat{z}$ vanishes everywhere on $\mathbb{F}^m$, and thus is the unique m-variate identically-zero polynomial over $\mathbb{F}$. □

The next step is to use the previous lemma to deduce that low-degree extensions are unique, when they exist:

Corollary C.16. *Let $\mathbb{F}$, m, $s_1, \ldots, s_m$, and $S_1, \ldots, S_m$ be as in Theorem C.14. Given a function $f\colon S_1 \times \cdots \times S_m \to \mathbb{F}$, if a low-degree extension of f in $\mathbb{F}$ exists, then it is unique.*

Proof. Suppose that two *distinct* m-variate polynomials $\hat{f}_1$ and $\hat{f}_2$ are low-degree extensions of f over $\mathbb{F}$. Then, the polynomial $\hat{z}' := \hat{f}_1 - \hat{f}_2$ would be a not-identically-zero low-degree extension of the all-zero function on $S_1 \times \cdots \times S_m$. This contradicts Lemma C.15, which guarantees that the only low-degree extension of the all-zero function on $S_1 \times \cdots \times S_m$ is the m-variate identically-zero polynomial over $\mathbb{F}$. □

We are left to show that low-degree extensions actually exist. We do this for point functions first:

Lemma C.17. *Let $\mathbb{F}$, m, $s_1, \ldots, s_m$, and $S_1, \ldots, S_m$ be as in Theorem C.14, and let $\vec{\alpha} = (\alpha_1, \ldots, \alpha_m)$ be an element in $S_1 \times \cdots \times S_m$. There exists a (unique) m-variate polynomial $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}\colon \mathbb{F}^m \to \mathbb{F}$ such that:*

- *for $i = 1, \ldots, m$, the degree of $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}$ in the i-th variable is $s_i - 1$,*
- *$\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{\alpha}) = 1_{\mathbb{F}}$, and*
- *$\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(S_1 \times \cdots \times S_m - \{\vec{\alpha}\}) = \{0_{\mathbb{F}}\}$.*

Proof. Define the "Lagrange interpolant" polynomial $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}\colon \mathbb{F}^m \to \mathbb{F}$ by

$$\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{x}) := \prod_{i=1}^{m} \left(\prod_{\beta_i \in S_i - \{\alpha_i\}} \frac{x_i - \beta_i}{\alpha_i - \beta_i} \right) \quad .$$

For every $i = 1, \ldots, m$,

$$\deg_{x_i}\left(\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{x})\right) = \deg_{x_i}\left(\prod_{\beta_i \in S_i - \{\alpha_i\}} \frac{x_i - \beta_i}{\alpha_i - \beta_i}\right) = s_i - 1 \ .$$

Moreover, $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{\alpha}) = \prod_{i=1}^{m} \prod_{\beta_i \in S_i - \{\alpha_i\}} \frac{\alpha_i - \beta}{\alpha_i - \beta} = 1$. Finally, for every $\vec{\gamma} \in S_1 \times \cdots \times S_m - \{\vec{\alpha}\}$, $\vec{\gamma}$ and $\vec{\alpha}$ differ in at least one coordinate $i \in \{1, \ldots, m\}$, so that $\gamma_i \in S_i$; hence, $\prod_{\beta_i \in S_i - \{\alpha_i\}} \frac{\gamma_i - \beta_i}{\alpha_i - \beta_i} = 0_{\mathbb{F}}$, because, when $\beta_i = \gamma_i$, the numerator of the ratio becomes zero. (Note that, since $\beta_i \in S_i - \{\alpha_i\}$, the denominator never vanishes!)

The uniqueness of $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}$ follows from Corollary C.16, because $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}$ is a low-degree extension of the "point" function that is equal to one on $\{\vec{\alpha}\}$ and zero everywhere else on $S_1 \times \cdots \times S_m$. □

Finally, constructing low-degree extensions for general functions easily follows from the previous lemma, by taking appropriate linear combinations of low-degree extensions of point functions:

Proof of Theorem C.14. For every $\vec{\alpha} = (\alpha_1, \ldots, \alpha_m)$ in $S_1 \times \cdots \times S_m$, let $\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}} \colon \mathbb{F}^m \to \mathbb{F}$ be the polynomial guaranteed by Lemma C.17. Define the polynomial $\hat{f} \colon \mathbb{F}^m \to \mathbb{F}$ by

$$\hat{f}(x_1, \ldots, x_m) = \sum_{\vec{\alpha} \in S_1 \times \cdots \times S_m} f(\vec{\alpha}) \cdot \delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(x_1, \ldots, x_m) \ .$$

Indeed, for every $i = 1, \ldots, m$,

$$\deg_{x_i}\left(\hat{f}(x_1, \ldots, x_m)\right) \leq \max_{\vec{\alpha} \in S_1 \times \cdots \times S_m} \left\{ \deg_{x_i}\left(\delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(x_1, \ldots, x_m)\right)\right\} = s_i - 1 < s_i \ ;$$

also, for every $\vec{\alpha}' \in S_1 \times \cdots \times S_m$,

$$\hat{f}(\vec{\alpha}') = \sum_{\vec{\alpha} \in S_1 \times \cdots \times S_m} f(\vec{\alpha}) \cdot \delta_{\mathbb{F},m,(S_1,\ldots,S_m),\vec{\alpha}}(\vec{\alpha}) = f(\vec{\alpha}') \cdot 1_{\mathbb{F}} = f(\vec{\alpha}') \ .$$

The uniqueness of $\hat{f}$ follows from Corollary C.16, because $\hat{f}$ is, by construction, a low-degree extension of f. □

C.4.2 Complexity of the general case

We now briefly discuss the computational properties of low-degree extensions, in the general case.

Univariate case. One way to compute a low-degree extension is to simply follow the constructive proof of Theorem C.14, a process known as *Lagrange interpolation*. The Lagrange interpolant polynomials can be pre-computed for a given set of evaluating points S (and can thus be used for distinct functions f defined over S); this can be done in $|S|^2$ time. Then, the linear combination of the Lagrange interpolant polynomials can also be computed in $|S|^2$ time [vzGG03, Theorem 5.1]

An alternative to Lagrange interpolation, with the same time complexity, is *Newton interpolation* [vzGG03, Exercise 5.11].

Also, we recall that, in the univariate case, polynomial interpolation can alternatively be viewed as having to find the solution of a linear system of $|S|$ equations involving a Vandermonde matrix.

However, even when no additional properties are assumed about S, there are faster divide-and-conquer algorithms to compute a low-degree extension:

Theorem C.18 ([vzGG03, Corollary 10.12])**.** *A low-degree extension can be computed in* $O(\mathsf{M}(|S|)\log|S|)$ *field operations, where* $\mathsf{M}(n)$ *is the time to multiply two polynomials of degree at most* n*. (Recall that, ithout additional assumptions on* S*, the best upper bound on* $\mathsf{M}(n)$ *is* $O(n^{\log 3}) \approx O(n^{1.585})$*, via Karatsuba's algorithm.)*

Fortunately, in the applications that we have in mind, the sets S in which we will be interested do satisfy additional properties: they will be linear subsets of the field. Therefore, we will be able to benefit from faster interpolation algorithms that use additive inverse FFT methods to obtain great *quasilinear* running times. (See Section C.4.3.)

Multivariate case. Once again, one way to proceed is to follow the constructive proof of Theorem C.14: the Lagrange interpolant polynomials from the proof of Theorem C.14 can be pre-computed for a given set of evaluating points $S_1 \times \cdots \times S_m$ in $\prod_{i=1}^{m} |S_i|^2$ time; then, the linear combination of the Lagrange interpolant polynomials can also be found in $\prod_{i=1}^{m} |S_i|^2$ time.

C.4.3 Interpolation over linear subsets

In the special case (of our interest) where $\mathbb{F} = \mathbb{F}_{2^f}$ for some $f \in \mathbb{N}$ and S is a linear subset of $\mathbb{F}$, much faster algorithms are known. This is the problem of *polynomial interpolation over linear subsets.*

For example, the additive inverse FFT that is the "dual" of the von zur Gathen and Gerhard additive FFT [vzGG96] has complexity $O(|S|(\log|S|)^2)$; this algorithm also derives its speed from the use of linearized polynomials. (This algorithm is given explicitly in [BSCGT12].)

Algorithms with even better asymptotic complexity are known; see for example [Mat08, Chapters 4.4 and 4.5]. Nonetheless, the von zur Gathen and Gerhard additive inverse FFT can be implemented very easily and does not "hide" any big constants in its practical running time.

We note that Bhattacharyya [Bha05, Section 2.1] also developed a fast algorithm for interpolation over linear subsets, but his algorithm in practice performs worse than the von zur Gathen and Gerhard additive inverse FFT.

C.4.4 Linearized interpolation

Another special case (of our interest) is a problem that we call *linearized polynomial interpolation*:

- **input:** a basis $\mathcal{B} = (\alpha_1, \ldots, \alpha_h)$ and a vector of values $\vec{\beta} = (\beta_1, \ldots, \beta_h)$;
- **output:** a linearized polynomial $P(x) = \sum_{j=0}^{h-1} \gamma_j x^{2^j} \in \mathbb{F}[x]$ such that $P(\alpha_i) = \beta_i$.

In other words, we are asked to solve the following linear system:

$$\begin{pmatrix} \alpha_1 & \alpha_1^2 & \alpha_1^4 & \cdots & \alpha_1^{2^{h-1}} \\ \alpha_2 & \alpha_2^2 & \alpha_2^4 & \cdots & \alpha_2^{2^{h-1}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_h & \alpha_h^2 & \alpha_h^4 & \cdots & \alpha_h^{2^{h-1}} \end{pmatrix} \begin{pmatrix} \gamma_0 \\ \gamma_1 \\ \vdots \\ \gamma_{h-1} \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_h \end{pmatrix} . \tag{13}$$

Note that it is indeed important for $\alpha_1, \ldots, \alpha_h$ to be linearly independent, otherwise the system may not always have a solution.

Recall from Remark C.8 that the problem of linearized polynomial interpolation arises when finding the coefficients of the (sparse) low-degree extensions of maps that are linear over the base field; finding these sparse low-degree extensions is important when we intend to carefully arithmetize boolean circuits. See Section C.7 for more details.

Kopparty suggested to us an approach mimicking the Vandermonde algorithm [Kop10]:

Theorem C.19. *There exists an algorithm* SolveLPIP *such that, on input an irreducible polynomial I, basis $\mathcal{B} = (\alpha_1, \ldots, \alpha_h)$, and vector $\vec{\beta} = (\beta_1, \ldots, \beta_h)$, solves the linearized polynomial interpolation problem in $O(h^2 \log h \cdot \log p)$ time and $O(h)$ space.*

Proof. The idea is to use linearized polynomials and solve the problem recursively. We describe the desired algorithm SolveLPIP in three main steps:

1. Find two linearized polynomials $Q(x)$ and $R(x)$ of respective degrees at most $2^{\lfloor h/2 \rfloor}$ and $2^{\lceil h/2 \rceil}$ such that:

   $$Q(\alpha_1) = Q(\alpha_2) = \cdots = Q(\alpha_{\lfloor h/2 \rfloor}) = 0_{\mathbb{F}} \quad \text{and} \quad R(\alpha_{\lfloor h/2 \rfloor + 1}) = R(\alpha_{\lfloor h/2 \rfloor + 2}) = \cdots = R(\alpha_h) = 0_{\mathbb{F}} \ .$$

   Note that $Q(x)$ and $R(x)$ can be found in $O(h^2 \log p)$ field operations via divide and conquer, e.g., by setting $R(x) := \text{FindSubspPoly}(I, (\alpha_1, \ldots, \alpha_{\lfloor h/2 \rfloor}))$ and $R(x) := \text{FindSubspPoly}(I, (\alpha_{\lfloor h/2 \rfloor + 1}, \ldots$ (See Remark C.12.)

2. Compute

   $$\alpha'_1 := R(\alpha_1), \alpha'_2 := R(\alpha_2), \ldots, \alpha'_{\lfloor h/2 \rfloor} := R(\alpha_{\lfloor h/2 \rfloor}) \quad \text{and}$$
   $$\alpha'_{\lfloor h/2 \rfloor + 1} := Q(\alpha_{\lfloor h/2 \rfloor + 1}), \alpha'_{\lfloor h/2 \rfloor + 2} := Q(\alpha_{\lfloor h/2 \rfloor + 2}), \ldots, \alpha_h := Q(\alpha_h) \ .$$

3. Recursively solve two (smaller) linearized polynomial interpolation problems by computing

   $$R'(x) := \text{SolveLPIP}(I, \mathcal{B}^{(0)}, \vec{\beta}^{(0)}) \quad \text{and} \quad Q'(x) := \text{SolveLPIP}(I, \mathcal{B}^{(1)}, \vec{\beta}^{(1)})$$

   where

   $$\mathcal{B}^{(0)} := (\alpha'_1, \ldots, \alpha'_{\lfloor h/2 \rfloor}) \quad \text{and} \quad \vec{\beta}^{(0)} := (\beta_1, \ldots, \beta_{\lfloor h/2 \rfloor})$$
   $$\mathcal{B}^{(1)} := (\alpha'_{\lfloor h/2 \rfloor + 1}, \ldots, \alpha'_h) \quad \text{and} \quad \vec{\beta}^{(1)} := (\beta_{\lfloor h/2 \rfloor + 1}, \ldots, \beta_h) \ .$$

4. Compute the polynomial $P(x) := Q'(Q(x)) + R'(R(x))$, and output $P(x)$.

Note that:

- for $i = 1, \ldots, \lfloor h/2 \rfloor$, $P(\alpha_i) = Q'(Q(\alpha_i)) + R'(R(\alpha_i)) = Q'(0_{\mathbb{F}}) + R'(\alpha'_i) = 0_{\mathbb{F}} + \beta_i = \beta_i$, and
- for $i = \lceil h/2 \rceil, \ldots, h$, $P(\alpha_i) = Q'(Q(\alpha_i)) + R'(R(\alpha_i)) = Q'(\alpha'_i) + R'(0_{\mathbb{F}}) = \beta_i + 0_{\mathbb{F}} = \beta_i$.

Moreover, the composition and the addition of two linearized polynomials is still a linearized polynomial; in particular, since $Q(x)$, $Q'(x)$, $R(x)$, and $R'(x)$ are all linearized polynomials, so is $Q'(Q(x))$ and $R'(R(x))$, and thus $P(x)$ as well. By induction, the degrees of R' and Q' are respectively at most $2^{\lfloor h/2 \rfloor - 1}$ and $2^{\lceil h/2 \rceil - 1}$, so that $Q'(Q(x))$ and $R'(R(x))$ have degrees that are respectively at most at most $2^{\lfloor h/2 \rfloor} 2^{\lceil h/2 \rceil - 1} = 2^{h-1}$ and $2^{\lceil h/2 \rceil} 2^{\lfloor h/2 \rfloor - 1} = 2^{h-1}$.

As for the number of field operations, the recursion is $T(h) = h^2 \log p + 2T(h/2) + h^2$ so that the overall number of required field operations is $O(h^2 \log h \log p)$. A space complexity of $O(h)$ can be achieved by using an analogous iterative algorithm. □

C.5 A Canonical Embedding

We give a general purpose way of embedding a set into large enough finite extension fields. Crucial to us is the simple observation that if the set has cardinality that is a power of the characteristic of the field, then the image of the set is a *subspace* over the base field. The lemma also spells out the computational cost of performing and reversing this embedding.

Lemma C.20 (Canonical Embedding of Finite Sets into Finite Fields)**.** *Let p be a prime, f a positive integer, and $\mathbb{F}$ a field extension of $\mathbb{F}_p$ of degree f, represented via an irreducible polynomial I with root x. Define $\Psi_f \colon \{0,1\}^f \to \mathbb{F}$ to be the function that maps a p-ary string $s_1 \cdots s_f$ to the element $\sum_{j=0}^{f-1} s_j \mathsf{x}^j$ in $\mathbb{F}$.*

Then, Ψ_f is a bijection from $\{0,1\}^f$ to $\mathbb{F}$, and we call Ψ_f the canonical embedding of $\{0,1\}^f$ into the finite field of degree f over $\mathbb{F}_p$*.*

In particular, for any set A with $|A| \le p^r$ for some positive integer r not larger than f, Ψ_f injects A into $\mathbb{F}$. Furthermore, if $|A| = p^r$, then $\Psi_f(A)$ is a linear subset of $\mathbb{F}$ of dimension r over $\mathbb{F}_p$, specified by a basis $(1_{\mathbb{F}}, \mathsf{x}, \ldots, \mathsf{x}^{r-1})$.

Moreover, Ψ_f and Ψ_f^{-1} are efficiently computable: there exist linear-time algorithms $\textsc{Comp}\boldsymbol{\Psi}$ *and* $\textsc{Comp}\boldsymbol{\Psi}^{-1}$ *such that*

$$\Psi_f(s_1 \cdots s_f) = \textsc{Comp}\boldsymbol{\Psi}\big(1^f, s_1 \cdots s_f\big) \quad \textit{and} \quad \Psi_f^{-1}\left(\sum_{j=0}^{f-1} s_j \mathsf{x}^j\right) = \textsc{Comp}\boldsymbol{\Psi}^{-1}\left(1^f, \sum_{j=0}^{f-1} s_j \mathsf{x}^j\right) \ .$$

Alternatively: there exist boolean circuit families $\{\textsc{Comp}\boldsymbol{\Psi}_f\}_{f\in\mathbb{N}}$ and $\{\textsc{Comp}\boldsymbol{\Psi}_f^{-1}\}_{f\in\mathbb{N}}$ for computing Ψ_f and Ψ_f^{-1}, and the circuits in both families have linear size, constant depth, and can be constructed in time linear in the circuit size.

Proof. That Ψ_f is a bijection is clear from its definition. If $|A| = p^r$, then

$$\Psi_f(A) = \left\{ \sum_{j=0}^{r-1} a_j \mathsf{x}^j \ : \ a_1, \ldots, a_{r-1} \in \{0, \ldots, p-1\} \right\} = \mathrm{span}_{\mathbb{F}_p}(1, \mathsf{x}, \ldots, \mathsf{x}^{r-1}) \ ,$$

so that $\Psi_f(A)$ is an r-dimensional linear subset of $\mathbb{F}$ over $\mathbb{F}_p$. Next, the algorithms that compute Ψ_f and Ψ_f^{-1} can be defined as follows:

$\textsc{Comp}\boldsymbol{\Psi}(1^f, s_1 \cdots s_f) \equiv$

1. Compute $\alpha(\mathsf{x}) := \sum_{j=0}^{f-1} s_j \mathsf{x}^j$ in the finite field $\mathbb{F}$.
2. Output $\alpha(\mathsf{x})$.

$\textsc{Comp}\boldsymbol{\Psi}^{-1}(1^f, \alpha(\mathsf{x})) \equiv$

1. For $i = 1, \ldots, f$: set s_i to be the coefficient of x^{i-1} in $\alpha(\mathsf{x})$.
2. Output $s_1 \cdots s_f$.

The corresponding circuit families can be easily deduced from the algorithms. □

C.6 Some Useful Families of Polynomials

We discuss here some (families of) polynomials, along with their complexities, that we shall find useful in our arithmetization constructions of Section 8.

We begin with *CMP polynomials*, which are direct arithmetizations of corresponding boolean comparator circuits. For example, given a 3-bit string $\sigma = 100$, the σ-CMP polynomial over $\mathbb{F}$ is given by $\mathrm{CMP}_{\mathbb{F},\sigma}(x_1, x_2, x_3) = x_1(1_{\mathbb{F}} - x_2)(1_{\mathbb{F}} - x_3)$; for $\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x}), \alpha_3(\mathsf{x}) \in \{0_{\mathbb{F}}, 1_{\mathbb{F}}\}$, $\mathrm{CMP}_{\mathbb{F},\sigma}(\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x}), \alpha_3(\mathsf{x})) = 1_{\mathbb{F}}$ if and only if $\alpha_1(\mathsf{x}), \alpha_2(\mathsf{x}), \alpha_3(\mathsf{x})$ correspond to the bits of σ.

Definition C.21 (CMP Polynomial)**.** *Let $\mathbb{F}$ be a finite field, m a positive integer, and σ an m-bit string. The* **σ-CMP polynomial over $\mathbb{F}$***, denoted $\mathrm{CMP}_{\mathbb{F},\sigma}$, is the m-variate polynomial over $\mathbb{F}$ defined by*

$$\mathrm{CMP}_{\mathbb{F},\sigma}(x_1, \ldots, x_m) := y_1^{(\sigma_1)} \cdots y_m^{(\sigma_m)} \ ,$$

where $y_i^{(\sigma_i)}$ is defined to be x_i if $\sigma_i = 1$ and $(1_{\mathbb{F}} - x_i)$ if $\sigma_i = 0$ for $i = 1, \ldots, m$. Note that $\mathrm{CMP}_{\mathbb{F},\sigma}$ is multilinear and, moreover, can be computed by a $\lceil \log(m) \rceil$-depth $\mathbb{F}$-arithmetic circuit with $m - \mathrm{weight}(\sigma)$ field subtractions and $m - 1$ field multiplications (and this circuit can be constructed in time that is linear in its size).

Next, we introduce *MUX polynomials* that, as the name suggests, are simply arithmetizations of corresponding boolean MUX circuits. For example, the 2-bit multiplexer polynomial over $\mathbb{F}$ is the polynomial

$$\mathrm{MUX}_{\mathbb{F},2}\big(s_1, s_2, x_0, x_1, x_2, x_3\big) = (1 - s_1)(1 - s_2)x_0 + (1 - s_1)s_2x_1 + s_1(1 - s_2)x_2 + s_1s_2x_3 \ .$$

Definition C.22 (MUX Polynomial)**.** *Let $\mathbb{F}$ be a finite field and m a positive integer. The* **m-bit multiplexer polynomial over $\mathbb{F}$***, denoted $\mathrm{MUX}_{\mathbb{F},m}$, is the $(m + 2^m)$-variate polynomial over $\mathbb{F}$ defined by*

$$\mathrm{MUX}_{\mathbb{F},m}\big(s_1, \ldots, s_m, x_0, \ldots, x_{(2^m-1)}\big) := \sum_{\sigma \in \{0,1\}^m} \mathrm{CMP}_{\mathbb{F},\sigma}(s_1, \ldots, s_m) \cdot x_\sigma \ .$$

Note that $\mathrm{MUX}_{\mathbb{F},m}$ is a multilinear homogeneous polynomial of degree $(m+1)$ and, moreover, can be computed by a $(\lceil \log(m) \rceil + 1 + m)$-depth $\mathbb{F}$-arithmetic circuit with m field subtractions, $2^m \cdot ((m-1)+1)$ field multiplications, and 2^m field additions; furthermore, this circuit can be constructed in time that is linear in its size by giving the input $(\mathbb{F}, m)$ to an algorithm FindMUX*.*

Another class of polynomials that we use are *alternator polynomials*; roughly, given two subsets $T, S \subseteq \mathbb{F}$ with $T \subseteq S$, we are interested in the polynomial that is equal to 1 on T but vanishes everywhere on $S - T$. Similarly to vanishing polynomials, when T and S are vector spaces, the corresponding alternator polynomial has a small arithmetic circuit that computes it (though, unlike in the case of subspace polynomials, the polynomial itself will not be sparse).

Theorem C.23. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the (smaller) field $\mathbb{B}$, and let $K \subseteq H$ be a vector space of dimension k over $\mathbb{B}$. Let $Y_{\mathbb{F},H,K} : \mathbb{F} \to \mathbb{F}$ the low-degree extension of the function over H that is equal to 1 over K but vanishes everywhere on $H - K$; we call $Y_{\mathbb{F},H,K}$ the* **alternator polynomial** *of K in H over $\mathbb{F}$.*

There exists a $O(\max\{k^2, (h-k)^2\} \log p)$-time $O(h \log p)$-space algorithm FindAlternator *that, on input (an irreducible polynomial representing) $\mathbb{F}$, a basis $(\mu_1, \ldots, \mu_h)$ for H, and a dimension k with $k \leq h$, computes a $O(h)$-size p^h-degree $\mathbb{F}$-arithmetic circuit $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$ that computes $Y_{\mathbb{F},H,K}$.*

Proof. If $K = H$, then the theorem trivially follows by letting $Y_{\mathbb{F},H,K}$ be the constant polynomial that is everywhere equal to 1.

So assume that $K \subsetneq H$ (so that $k < h$), in which case we argue as follows. Let $\mathcal{B}_K = (\mu_1, \ldots, \mu_k)$ be a basis for K, and let $\mathcal{B}_H = (\mu_1, \ldots, \mu_k, \mu_{k+1}, \ldots, \mu_h)$ be its completion to a basis for H. Let $Z_K(x) \in \mathbb{F}[x]$ be the vanishing polynomial for K (cf. Definition C.9), and recall that $Z_K(x)$ is $\mathbb{B}$-linear (cf. Claim C.10). Furthermore, by Lemma C.11, $Z_K(x)$ is sparse and its coefficients can be efficiently computed when given the basis $\mathcal{B}_K$; specifically, we know from Remark C.12 that there exists an algorithm FindSubspPoly that, on input $(\mathbb{F}, \mathcal{B}_K)$, in time $O(k^2 \log p) = \mathsf{poly}(k, \log p)$ outputs $c_0^{(K)}, \ldots, c_{k-1}^{(K)} \in \mathbb{F}$ such that

$$Z_K(x) = x^{p^k} + \sum_{i=0}^{k-1} c_i^{(K)} x^{p^i} \quad .$$

For $i \in \{k+1, \ldots, h\}$, define $\nu_i := Z_K(\mu_i)$. Observe that $\nu_{k+1}, \ldots, \nu_h$ are linearly independent for, if not, then, using the $\mathbb{B}$-linearity of Z_K it is possible to show that Z_K has more than p^k roots, which is a contradiction because the degree of Z_K is only p^k.

So let L be the $(h-k)$-dimensional $\mathbb{B}$-linear subset spanned by $\nu_{k+1}, \ldots, \nu_h$, and let $Z_L(x) \in \mathbb{F}[x]$ be its vanishing polynomial. Again invoking Claim C.10 and then Lemma C.11, we deduce that $Z_L(x)$ has the form

$$Z_L(x) = x^{p^{h-k}} + \sum_{i=0}^{h-k-1} c_i^{(L)} x^{p^i} \quad ,$$

where the coefficients $c_0^{(L)}, \ldots, c_{h-k-1}^{(L)} \in \mathbb{F}$ may be computed in time $O((h-k)^2 \log p) = \mathsf{poly}(h - k, \log p)$ by the algorithm FindSubspPoly on input $(\mathbb{F}, \mathcal{B}_L)$.

Define the polynomial $P \in \mathbb{F}[x]$ as[18]

$$P(x) := \frac{Z_L(x)}{c_0^{(L)} x} = x^{p^{h-k}-1} + \sum_{i=0}^{h-k-1} \frac{c_i^{(L)}}{c_0^{(L)}} x^{p^i-1} \quad .$$

Finally, define the polynomial $Q \in \mathbb{F}[x]$ as

$$Q(x) := P\big(Z_K(x)\big) = \big(Z_K(x)\big)^{p^{h-k}-1} + \sum_{i=0}^{h-k-1} \frac{c_i^{(L)}}{c_0^{(L)}} \big(Z_K(x)\big)^{p^i-1} \quad .$$

We claim that $Y_{\mathbb{F},H,K}(x) = Q(x)$. Indeed,

$$\deg(Q) = \deg(Z_K) \cdot \deg(P) < \deg(Z_K) \cdot \deg(Z_L) = p^k \cdot p^{h-k} = p^h \quad .$$

Moreover,

- for any $\alpha \in K$, $Q(\alpha) = P(Z_K(\alpha)) = P(0_{\mathbb{F}}) = c_0^{(L)}/c_0^{(L)} = 1_{\mathbb{F}}$; and
- for any $\alpha \in H - K$, then $Z_K(\alpha)$ is a non-zero element of L, so that $Z_L(\alpha) = 0$, and thus $Q(\alpha) = 0$.

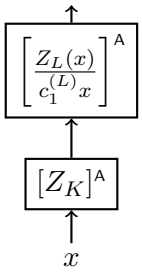
[18] Note that the coefficient of x in a subspace polynomial is never equal to zero. Indeed, suppose by way of contradiction that $Z_S(x)$ is a subspace polynomial where the coefficient of x is equal to zero. Then, $Z_S(x) = R(x^p) = (R'(x))^p$ for some linearized polynomials R and R'. Note that every root of $Z_S(x)$ is a root of $R'(x)$, but the degree of $R'(x)$ is smaller than the number of roots of $Z_S(x)$, which is a contradiction.

So that indeed Q agrees on H with the function that is equal to 1 over K and is equal to 0 on $H - K$ and, moreover, is of degree at most $|H| = p^h$. Thus, by the uniqueness of low-degree extensions (cf. Theorem C.14), it must indeed be the case that $Y_{\mathbb{F},H,K}(x) = Q(x)$.

Now that we have an explicit form for $Y_{\mathbb{F},H,K}(x)$, we can say something about what kind of circuit is needed for computing $Y_{\mathbb{F},H,K}(x)$, and how fast such a circuit may be generated. Note that, unlike a subspace polynomial, $Y_{\mathbb{F},H,K}(x)$ is *not* sparse, because it involves raising a sum of terms (namely, $Z_K(x)$) to powers $p^i - 1$ for $i \in \{1, \ldots, h-k\}$, and $p^{h-k} - 1$ is large.[19] Nonetheless, given an element $\alpha \in \mathbb{F}$, $Y_{\mathbb{F},H,K}(\alpha)$ may be computed efficiently, by first computing $\beta := Z_K(\alpha)$, and then computing $P(\beta)$, because both Z_K and P are sparse. (And being able to compute the polynomial efficiently is all that we will need.) Thus, we can define the algorithm FindAlternator as follows:

FindAlternator$\big(\mathbb{F}, (\mu_1, \ldots, \mu_h), k\big) \equiv$

1. If $k = h$, output the constant circuit $1_{\mathbb{F}}$; otherwise continue.
2. Define $\mathcal{B}_K := (\mu_1, \ldots, \mu_k)$.
3. Compute $[Z_K]^{\mathsf{A}} := \text{FindSubspPoly}(\mathbb{F}, \mathcal{B}_K)$.
4. For $i = k+1, \ldots, h$, compute $\nu_i := Z_K(\mu_i)$.
5. Define $\mathcal{B}_L := (\nu_{k+1}, \ldots, \nu_h)$.
6. Compute $[Z_L]^{\mathsf{A}} := \text{FindSubspPoly}(\mathbb{F}, \mathcal{B}_L)$.
7. Compute $[\frac{Z_L(x)}{c_0^{(L)} x}]^{\mathsf{A}}$.
8. Compute $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$ as the composition of $[Z_K]^{\mathsf{A}}$ and $[\frac{Z_L(x)}{c_0^{(L)} x}]^{\mathsf{A}}$.
9. Output $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$.



Note that $[Y_{\mathbb{F},H,K}]^{\mathsf{A}}$ has size $O(h)$. Furthermore, the time complexity of FindAlternator is given by $O(k^2 \log p) + O((h-k)k \log p) + O((h-k)^2 \log p)$ and its space complexity is given by $O(h \log p)$. □

Finally, we discuss polynomials that represent *projection functions over linear subspaces.*

Definition C.24. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the base field $\mathbb{B}$, and consider the basis $\mathcal{B}_H = (1, \mathsf{x}, \ldots, \mathsf{x}^{h-1})$ for H. (Recall that x is a root of the irreducible polynomial I used to represent $\mathbb{F}$.) For $j \in \{1, \ldots, h\}$, define $p_{\mathbb{F},H,\mathbb{B},j} \colon H \to \mathbb{F}$ to be the function that, for any $\lambda = \sum_{i=1}^{h} \lambda_i \mathsf{x}^{i-1} \in H$, outputs $\lambda_j \in \mathbb{B}$.*

Let us first establish $\mathbb{B}$-linearity:

Lemma C.25. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the base field $\mathbb{B}$. Then, for every $j \in \{1, \ldots, h\}$, the function $p_{\mathbb{F},H,\mathbb{B},j}$ from Definition C.24 is $\mathbb{B}$-linear.*

Proof. Fix any two elements $\lambda = \sum_{i=1}^{h} \lambda_i \mathsf{x}^{i-1}$ and $\gamma = \sum_{i=1}^{h} \gamma_i \mathsf{x}^{i-1}$ in H. Then, for any two elements α and β in $\mathbb{B}$,

$$\begin{aligned} p_{\mathbb{F},H,\mathbb{B},j}(\alpha\lambda + \beta\gamma) &= p_{\mathbb{F},H,\mathbb{B},j}\left(\sum_{i=1}^{h} (\alpha\lambda_i + \beta\gamma_i)\mathsf{x}^{i-1}\right) \\ &= \alpha\lambda_j + \beta\gamma_j \\ &= \alpha \cdot p_{\mathbb{F},H,\mathbb{B},j}\left(\sum_{i=1}^{h} \lambda_i \mathsf{x}^{i-1}\right) + \beta \cdot p_{\mathbb{F},H,\mathbb{B},j}\left(\sum_{i=1}^{h} \gamma_i \mathsf{x}^{i-1}\right) \end{aligned}$$

[19] Indeed, note that $(\alpha+\beta)^{p^i-1} = \frac{1}{q-1}\prod_{j=0}^{i-1}(\alpha^{p^j} + \beta^{p^j})$ is a product of 2^i terms.

$$= \alpha \cdot p_{\mathbb{F},H,\mathbb{B},j}(\lambda) + \beta \cdot p_{\mathbb{F},H,\mathbb{B},j}(\gamma) \ ,$$

as desired. □

Because a projection function is $\mathbb{B}$-linear, we can express it as a sparse polynomial:

Lemma C.26. *Let $H \subseteq \mathbb{F}$ be a vector space of dimension h over the base field $\mathbb{B}$. Then, for every $j \in \{1, \ldots, h\}$, there exists a unique polynomial $\Pi_{\mathbb{F},H,\mathbb{B},j} \colon \mathbb{F} \to \mathbb{F}$ of the form*

$$\Pi_{\mathbb{F},H,\mathbb{B},j}(x) := \sum_{i=0}^{h-1} \alpha_i x^{p^i} \ , \tag{14}$$

where $\alpha_0, \ldots, \alpha_{h-1} \in \mathbb{F}$, such that $\Pi_{\mathbb{F},H,\mathbb{B},j}$ agrees on all of H with the function $p_{\mathbb{F},H,\mathbb{B},j}$ from Definition C.24; we call $\Pi_{\mathbb{F},H,\mathbb{B},j}$ the **projection polynomial** *for the j-th bit in H with respect to $\mathbb{B}$ over $\mathbb{F}$. Moreover, the coefficients $\alpha_0, \ldots, \alpha_{h-1}$ can be computed in $O(h^2 \log h \cdot \log p) = \mathsf{poly}(h, \log p)$ time and $O(h)$ space.*

In particular, there exists a $\mathsf{poly}(h, \log p)$-time algorithm FindΠ *that, on input (an irreducible polynomial representing) $\mathbb{F}$, a basis $\mathcal{B}_H$ for H, and an index $j \in \{1, \ldots, h\}$, computes a $O(h \log p)$-size $\mathbb{F}$-arithmetic circuit $[\Pi_{\mathbb{F},H,\mathbb{B},j}]^{\mathsf{A}}$ that computes $\Pi_{\mathbb{F},H,\mathbb{B},j}$.*

Proof. Fix every $j \in \{1, \ldots, h\}$. By Lemma C.25, $p_{\mathbb{F},H,\mathbb{B},j}$ is $\mathbb{B}$-linear. Hence, by Claim C.7, the equality in Equation 14 follows. That the coefficients $\alpha_0, \ldots, \alpha_{h-1}$ may be computed in the claimed efficiency follows from Remark C.8 (and note that in this case, since $g = p_{\mathbb{F},H,\mathbb{B},j}$, $g(e_1), \ldots, g(e_h)$ can all be easily computed in $O(h^2)$ time because $e_i = \mathsf{x}^{i-1}$ and $g(e_j) = 1_{\mathbb{F}}$ and $g(e_j) = 0_{\mathbb{F}}$ for $i \neq j$). □

C.7 Efficient Algebraic Computation

Let $\mathbb{F}$ be an extension field of $\mathbb{F}_2 = \{0_{\mathbb{F}}, 1_{\mathbb{F}}\}$, H an h-dimensional linear subset of $\mathbb{F}$, and $\mathcal{B}_H = (e_1, \ldots, e_h)$ a basis for H.

Define $\mathrm{bit} \colon \mathbb{F} \to \{0,1\} \cup \{\perp\}$ to be the function that maps $\mathbb{F}_2$ to the two boolean values and $\mathbb{F} - \mathbb{F}_2$ to $\{\perp\}$, that is, $\mathrm{bit}(0_{\mathbb{F}}) = 1$, $\mathrm{bit}(1_{\mathbb{F}}) = 1$, and $\mathrm{bit}(\alpha) = \perp$ for every $\alpha \in \mathbb{F} - \mathbb{F}_2$.

Define $\mathrm{bin}_{\mathcal{B}_H} \colon H \to \{0,1\}^h$ to be the function that gives the representation of elements in H in terms of the basis $\mathcal{B}_H$; that is, for any $\alpha = \sum_{j=1}^{h} \lambda_j e_j \in H$, with $\lambda_1, \ldots, \lambda_h \in \mathbb{F}_2$ and $e_1, \ldots, e_h \in \mathbb{F}$, it holds that $\mathrm{bin}_{\mathcal{B}_H}(\alpha) = (\mathrm{bit}(\lambda_1), \ldots, \mathrm{bit}(\lambda_h))$. Note that $\mathrm{bin}_{\mathcal{B}_H}$ is injective, so that referring to $\mathrm{bin}_{\mathcal{B}_H}$ as a representation of the elements in H (with respect to the basis $\mathcal{B}_H$) is indeed justified.

The function $\mathrm{bin}_{\mathcal{B}_H}$ can naturally be extended to give the basis representation of elements in a *product* of linear subsets: for $i = 1, \ldots, m$, let H_i be an h_i-dimensional linear subset of $\mathbb{F}$, and let $\mathcal{B}_{H_i} = (e_1^{(i)}, \ldots, e_{h_i}^{(i)})$ be a basis for H_i; then the function

$$\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}} \colon H_1 \times \cdots \times H_m \to \{0,1\}^{h_1 + \cdots + h_m}$$

is defined by

$$\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m) := \mathrm{bin}_{\mathcal{B}_{H_1}}(\alpha_1) \circ \cdots \circ \mathrm{bin}_{\mathcal{B}_{H_m}}(\alpha_m) \ ,$$

where $\circ$ is the string concatenation operator. Thus, it is natural to call $\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m)$ a *binary representation* of an element $(\alpha_1, \ldots, \alpha_m) \in H_1 \times \cdots \times H_m$.

Ben-Sasson and Sudan [BSGH+05, Theorem 5.5] showed that any small-depth, small-size, single-bit-output boolean circuit operating on the binary representation of elements in a product space H^m, for some linear subset H of $\mathbb{F}$, can be converted into an equivalent arithmetic circuit of small size and moderate degree (exponential in the depth) over $\mathbb{F}$. (In particular, any bit of the binary representation of an element in H^m can be computed efficiently.)

An unfortunate yet seemingly inherent cost of arithmetizing boolean circuits is that retrieving any single bit of a field element induces a polynomial of very high degree; thus we shall use this tool very sparingly in our reductions. Indeed, it is because of this cost that in Section 8, when arithmetizing sGCPs, we choose to "stripe" the bit of a color across many field elements instead of simply "packing" these in one field element and retrieve them later with a polynomial; we still have to invoke this result, though, to "know" where we are in the graph.

Here we prove a simple but useful generalization of [BSGH$^+$05, Theorem 5.5] and take the opportunity to state its improved complexity in light of the fast algorithm of Section C.4.4. Specifically, we allow for the product space to consist of different (linear) subsets and we allow for the output of the boolean circuit to consist of multiple bits. We also choose to state the theorem in terms of the multiplicative degree (see Section 6) of the boolean circuit to be arithmetized, as multiplicative degree is in our setting a finer and more convenient complexity metric.

Theorem C.27 (Arithmetizing boolean Circuits over Linear Spaces)**.** *Let $\mathbb{F}$ be an extension field of $\mathbb{F}_2$, m a positive integer, and, for $i = 1, \ldots, m$, H_i an h_i-dimensional linear subset of $\mathbb{F}$ with a basis $\mathcal{B}_{H_i} = (e_1^{(i)}, \ldots, e_{h_i}^{(i)})$.*

Fix a positive integer ρ. For any boolean function $g\colon \{0,1\}^{h_1+\cdots+h_m} \to \{0,1\}^\rho$ computed by a boolean circuit C of size s and (multiplicative) degree D, there exist ρ polynomials $\hat{g}_1, \ldots, \hat{g}_\rho\colon \mathbb{F}^m \to \mathbb{F}$ such that:

1. *For $i = 1, \ldots, m$ and $j = 1, \ldots, \rho$, the degree in the i-th variable of the j-th polynomial is*

$$\frac{|H_i|}{2} \max_{k - \sum_{r=0}^{i-1} h_r \in \{1,\ldots,h_i\}} \mathsf{D}[k \to j]$$

   where we defined $h_0 := 0$;
2. *All of the polynomials are simultaneously computable by a multi-element-output $\mathbb{F}$-arithmetic circuit $\widehat{C}$ of size $O(\sum_{i=1}^m h_i^2 + \mathsf{s})$; and*
3. *For every $(\alpha_1, \ldots, \alpha_m)$ in $H_1 \times \cdots \times H_m$,*

$$\mathrm{bit}\big(\hat{g}_1(\alpha_1, \ldots, \alpha_m)\big) \circ \cdots \circ \mathrm{bit}\big(\hat{g}_\rho(\alpha_1, \ldots, \alpha_m)\big) = g\big(\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m)\big) \ .$$

Moreover, $\widehat{C}$ can be constructed in time $O(\sum_{l=1}^m h_l^3 \log h_l + \mathsf{s})$ and space $O(\sum_{i=1}^m h_i^2 + \mathsf{s})$ when given as input the basis $\mathcal{B}_{H_l}$ for each linear space H_l and C.

First, we prove that any bit of the binary representation of an element in $H_1 \times \cdots \times H_m$ can be computed efficiently; specifically, any individual bit, say a bit of an element in the subspace H_l, can be extracted by a polynomial of degree at most $2^{h_l - 1} = |H_l|/2$ that is computable by an $\mathbb{F}$-arithmetic circuit of size at most $O(h_l)$:

Lemma C.28. *Define $h_0 := 0$. Fix any $l \in \{1, \ldots, m\}$ and $\iota \in \{1, \ldots, h_l\}$, and consider the special case where g is the projection to the $(\iota + \sum_{i=0}^{l-1} h_i)$-th bit of the input. Then there exists a polynomial $\hat{g}_{l,\iota}\colon \mathbb{F}^m \to \mathbb{F}$ of degree at most $|H_l|/2$ computable by an $\mathbb{F}$-arithmetic circuit $\widehat{C}_{l,\iota}$ of size $O(h_l)$ such that, for every $(\alpha_1, \ldots, \alpha_m)$ in $H_1 \times \cdots \times H_m$,*

$$\mathrm{bit}\big(\hat{g}_{l,\iota}(\alpha_1, \ldots, \alpha_m)\big) = g\big(\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha_1, \ldots, \alpha_m)\big) \ .$$

Moreover, given the basis $\mathcal{B}_{H_l} = (e_1^{(l)}, \ldots, e_{h_l}^{(l)})$, $\widehat{C}_{l,\iota}$ can be constructed in time $O(h_l^2 \log h_l)$.

Proof. Consider the function $\tilde{g}\colon H_1 \times \cdots \times H_m \to \mathbb{F}$ satisfying $(\mathrm{bit} \circ \tilde{g}) = (g \circ \mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}})$.[20] Observe that $(\mathrm{bit} \circ \tilde{g})(H_1 \times \cdots \times H_m) = \{0,1\}$, so we deduce that $\tilde{g}(H_1 \times \cdots \times H_m) = \mathbb{F}_2$; moreover, for any two $\alpha = \sum_{i=1}^{m}\sum_{j=1}^{h_i} \lambda_j^{(i)} e_j^{(i)}$ and $\beta = \sum_{i=1}^{m}\sum_{j=1}^{h_i} \gamma_j^{(i)} e_j^{(i)}$ in $H_1 \times \cdots \times H_m$,

$$\begin{aligned}(\mathrm{bit} \circ \tilde{g})(\alpha+\beta) &= g\big(\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha+\beta)\big)\\ &= \mathrm{bit}(\lambda_\iota^{(l)}) + \mathrm{bit}(\gamma_\iota^{(l)})\\ &= g\big(\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\alpha)\big) + g\big(\mathrm{bin}_{\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}}(\beta)\big)\\ &= (\mathrm{bit} \circ \tilde{g})(\alpha) + (\mathrm{bit} \circ \tilde{g})(\beta)\ ,\end{aligned}$$

implying that $\tilde{g}$ is a $\mathbb{F}_2$-linear map.

Invoking Claim C.7 with $\mathbb{F}$, $\mathbb{B} = \mathbb{F}_2$, $H = H_l$, $h = h_l$, and the (also $\mathbb{F}_2$-linear) function $\tilde{g}|_{H_l}$, we deduce that there exists a (unique) low-degree extension $\hat{g}'_{l,\iota}\colon \mathbb{F} \to \mathbb{F}$ of $\tilde{g}|_{H_l}$ that agrees with $\tilde{g}|_{H_l}$ on H_l and has the form $\hat{g}'_{l,\iota}(x_l) = \sum_{j=0}^{h_l-1} c_j x_l^{2^j}$, where $c_0,\ldots,c_{h_l-1} \in \mathbb{F}$ is a solution to the following system of h_l linear equations:

$$\left\{ \sum_{j=0}^{h_l-1} c_j \pi_j(e_s^{(l)}) = \tilde{g}|_{H_l}(e_s^{(l)}) \ :\ s = 1,\ldots,h_l \right\}\ , \tag{15}$$

where π_j is the mapping $x_l \mapsto x_l^{2^j}$. Given the basis $\mathcal{B}_{H_l} = (e_1^l,\ldots,e_{h_l}^{(l)})$, we can compute $\tilde{g}|_{H_l}(e_1^{(l)}),\ldots,\tilde{g}|_{H_l}(e_{h_l}^{(l)})$ in $O(h_l^2)$ time and then, by Remark C.8, compute the coefficients $c_0,\ldots,c_{h_l-1}$ in $O(h_l^2 \log h_l)$ time.

Therefore, $\hat{g}'_{l,\iota}$ can be computed by the $\mathbb{F}$-arithmetic circuit $\widehat{C}'_{l,\iota}$ of size $O(h_l)$ (computable in $O(h_l^2 \log h_l)$ time from $\mathcal{B}_{H_l}$) that, on input x_l, (1) computes the powers $x_l, x_l^2, \ldots, x_l^{2^{h_l-1}}$ (by repeated squaring); and (2) outputs the linear combination $\sum_{j=0}^{h_l-1} c_j x_l^{2^j}$.

To finish the proof of the lemma, we note that we can simply let $\widehat{C}$ be the circuit that works with that component of $\mathbb{F}^m$ in which the projected bit is present, and ignore the remaining components of $\mathbb{F}^m$; specifically, we let $\hat{g}_{l,\iota}(x_1,\ldots,x_m) := \hat{g}'_{l,\iota}(x_l)$ and $\widehat{C}_{l,\iota}(x_1,\ldots,x_m) := \widehat{C}'_{l,\iota}(x_l)$. □

The case for a general boolean function g is obtained by constructing an arithmetic circuit that first extracts all the individual bits and then uses a straightforward arithmetization of the original boolean circuit C:

Proof of Theorem C.27. For each $l \in \{1,\ldots,m\}$ and $\iota \in \{1,\ldots,h_l\}$, invoking Lemma C.28 with l and ι, we deduce that there exists a polynomial $\hat{g}_{l,\iota}$ of degree at most $|H_l|/2$ (which is computable by an $\mathbb{F}$-arithmetic circuit $\widehat{C}_{l,\iota}$ of $O(h_l)$ size) that agreees with $\pi_j^{(l)}$, the projection to the $(\iota+\sum_{i=0}^{l-1} h_i)$-th bit of the input, on $H_1 \times \cdots \times H_m$; moreover, $\widehat{C}_{l,\iota}$ can be constructed in $O(h_l^2 \log h_l)$ time from $\mathcal{B}_{H_l}$.

We deduce that there exists an $\mathbb{F}$-arithmetic circuit $\widehat{C}_{\mathrm{bin}}$ of size $\sum_{i=1}^{m} h_i O(h_i) = O(\sum_{i=1}^{m} h_i^2)$ that extracts *all* the individual bits of the input: we define $\widehat{C}_{\mathrm{bin}}$ to be the multi-output circuit that computes each $\widehat{C}_{l,\iota}$,

$$\widehat{C}_{\mathrm{bin}}(x_1,\ldots,x_m) = \times_{l=1}^{m} \times_{\iota=1}^{h_l} \widehat{C}_{l,\iota}(x_1,\ldots,x_m)\ .$$

Note that $\widehat{C}_{\mathrm{bin}}$ has $\sum_{l=1}^{m} h_l$ outputs, and, for $l \in \{1,\ldots,m\}$ and $\iota \in \{1,\ldots,h_l\}$, the $(\iota + \sum_{i=0}^{l-1} h_i)$-th output contains only variable x_l, with degree at most $|H_l|/2$. Moreover, from the definition of $\widehat{C}_{\mathrm{bin}}$ and the constructibility of each $\widehat{C}_{l,\iota}$, we deduce that $\widehat{C}_{\mathrm{bin}}$ can be constructed in time $\sum_{l=1}^{m} h_l \cdot O(h_l^2 \log h_l) = O(\sum_{l=1}^{m} h_l^3 \log h_l)$, when given as input the basis $\mathcal{B}_{H_l}$ for each linear space H_l.

[20] Note that $\tilde{g}$ depends on $\mathcal{B}_{H_1},\ldots,\mathcal{B}_{H_m}$, but we drop this subscript.

Next, we argue that there exists a ρ-element-output $\mathbb{F}$-arithmetic circuit $\widehat{C}_g$ of size $O(\mathsf{s})$ that, on input the $\sum_{i=1}^{m} h_i$ individual bits[21] of an input in $H_1 \times \cdots \times H_m$, computes g and, moreover, the polynomial induced by $\widehat{C}_g$ has (multiplicative) degree (function) D and can be constructed in time $O(\mathsf{s})$ from C (which was the boolean circuit of size s and depth d that computes g). Indeed, letting $\widehat{C}_g$ be the straightforward arithmetization of C does the job: for example, the arithmetization of the AND and NOT gate is performed as $\mathrm{AND}(x, y) = x \cdot y$ and $\mathrm{NOT}(x) = 1 - x$; the arithmetized gates compute the intended values whenever x and y take values in $\mathbb{F}_2$, which is the case.[22]

Finally, to finish the proof, we let $\widehat{C} = \widehat{C}_g \circ \widehat{C}_{\mathrm{bin}}$, so that the overall size is $O(\sum_{i=1}^{m} h_i^2 + \mathsf{s})$. As for the multiplicative degree: letting $h_0 := 0$, for $i = 1, \ldots, m$ and $j = 1, \ldots, \rho$, the degree in the i-th variable of the j-th polynomial is

$$\frac{|H_i|}{2} \cdot \left(\max_{k - \sum_{r=0}^{i-1} h_r \in \{1,\ldots,h_i\}} \mathsf{D}[k \to j] \right) .$$

Moreover, from the constructibility of $\widehat{C}_g$ and $\widehat{C}_{\mathrm{bin}}$, we deduce that $\widehat{C}$ can be constructed in time $O(\sum_{l=1}^{m} h_l^3 \log h_l + \mathsf{s})$ and space $O(\sum_{i=1}^{m} h_i^2 + \mathsf{s})$ when given as input the basis $\mathcal{B}_{H_l}$ for each linear space H_l and C. □

[21] Actually, bits represented as elements of $\mathbb{F}_2$.

[22] More generally, any binary gate $G(x, y)$ is replaced by an appropriate multilinear polynomial in x and y, which extends the corresponding mapping $\mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2$.

References

[AF07] Masayuki Abe and Serge Fehr. Perfect NIZK with adaptive soundness. In *TCC '07: Proceedings of the 4th Theory of Cryptography Conference on Theory of Cryptography*, pages 118–136, Berlin, Heidelberg, 2007. Springer-Verlag.

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming*, ICALP '10, pages 152–163, 2010.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $0(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, 1983.

[AV77] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. In *Proceedings on 9th Annual ACM Symposium on Theory of Computing*, STOC '77, pages 30–41, 1977.

[BC12] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In *Proceedings of the 32nd Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '12, 2012.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[BCCT11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. Cryptology ePrint Archive, Report 2011/443, 2011.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. Cryptology ePrint Archive, Report 2011/95, 2012.

[BEG+91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, FOCS '91, pages 90–99, 1991.

[Ben65] Václav E. Beneš. *Mathematical theory of connecting networks and telephone traffic*. New York, Academic Press, 1965.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, STOC '91, pages 21–32, 1991.

[BG08] Boaz Barak and Oded Goldreich. Universal arguments and their applications. *SIAM Journal on Computing*, 38(5):1661–1694, 2008. Preliminary version appeared in CCC '02. We reference the version available online at `http://www.wisdom.weizmann.ac.il/~oded/PS/ua-rev3.ps`.

[BGV12] Zvika Brakerski, Craig Gentry, and Vaikuntanathan Vinod. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science*, ITCS '12, pages 309–325, 2012.

[Bha05] Arnab Bhattacharyya. Implementing probabilistically checkable proofs of proximity. Technical Report MIT-CSAIL-TR-2005-051, MIT, 2005. Available at `http://dspace.mit.edu/handle/1721.1/30562`.

[BHZ87] Ravi B. Boppana, Johan Håstad, and Stathis Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[BOGKW88] Michael Ben-Or, Shafi Goldwasser, Joe Kilian, and Avi Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 113–131, 1988.

[BSCGT12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Towards practical PCPs, 2012. Electronic Colloquium on Computational Complexity.

[BSGH+04] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, STOC '04, pages 1–10, 2004. Full version available at `http://people.seas.harvard.edu/~salil/research/shortPCP.pdf`.

[BSGH+05] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Short PCPs verifiable in polylogarithmic time. In *Proceedings of the 20th Annual IEEE Conference on Computational Complexity*, CCC '05, pages 120–134, 2005.

[BSGH+06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust PCPs of proximity, shorter PCPs, and applications to coding. *SIAM Journal on Computing*, 36(4):889–974, 2006. Preliminary versions of this paper have appeared in Proceedings of the 36th ACM Symposium on Theory of Computing and in Electronic Colloquium on Computational Complexity.

[BSS08] Eli Ben-Sasson and Madhu Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008. Preliminary version appeared in STOC '05.

[BSSVW03] Eli Ben-Sasson, Madhu Sudan, Salil Vadhan, and Avi Wigderson. Randomness-efficient low degree tests and short pcps via epsilon-biased sets. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, STOC '03, pages 612–621, 2003.

[CKLR11] Kai-Min Chung, Yael Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In *Proceedings of the 31st Annual International Cryptology Conference*, CRYPTO '11, pages 151–168, 2011.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Proceedings of the 30th Annual International Cryptology Conference*, CRYPTO '10, pages 483–501, 2010.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science*, ITCS '12, pages 90–112, 2012.

[CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, 1972.

[CRR11] Ran Canetti, Ben Riva, and Guy N. Rothblum. Two 1-round protocols for delegation of computation. Cryptology ePrint Archive, Report 2011/518, 2011.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Proceedings of the 1st Symposium on Innovations in Computer Science*, ICS '10, pages 310–331, 2010.

[CT12] Alessandro Chiesa and Eran Tromer. Proof-carrying data: Secure computation on untrusted platforms (high-level description). *The Next Wave: The National Security Agency's review of emerging technologies*, 19(2):40–46, 2012.

[CTY10] Graham Cormode, Justin Thaler, and Ke Yi. Verifying computations with streaming interactive proofs. ECCC, 2010. Available at `http://eccc.hpi-web.de/report/2010/159/`.

[DCL08] Giovanni Di Crescenzo and Helger Lipmaa. Succinct NP proofs from an extractability assumption. In *Proceedings of the 4th Conference on Computability in Europe*, CiE '08, pages 175–185, 2008.

[DFH12] Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In *Proceedings of the 9th International Conference on Theory of Cryptography*, TCC '12, pages 54–74, 2012.

[DT83] Patrick W. Dymond and Martin Tompa. Speedups of deterministic machines by synchronous parallel machines. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, STOC '83, pages 336–343, 1983.

[Fic93] Faith E. Fich. The complexity of computation on the parallel random access machine. In *[Rei93]*, pages 843–899, 1993.

[FvDD12] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Compilation techniques for efficient encrypted computation. Cryptology ePrint Archive, Report 2012/266, 2012.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual International Cryptology Conference*, CRYPTO '10, pages 465–482, 2010.

[GGPR12] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. Cryptology ePrint Archive, Report 2012/215, 2012.

[GH98] Oded Goldreich and Johan Håstad. On the complexity of interactive proofs with bounded communication. *Information Processing Letters*, 67(4):205–214, 1998.

[GH11a] Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *Proceedings of the IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS' 11, pages 107–109, 2011.

[GH11b] Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Proceedings of the 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '11, pages 129–148, 2011.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC '08: Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 113–122, New York, NY, USA, 2008. ACM.

[GL03] William F. Gilreath and Phillip A. Laplante. *Computer Architecture*. Kluwer Academic Publishers, 2003.

[GLR11] Shafi Goldwasser, Huijia Lin, and Aviad Rubinstein. Delegation of computation without rejection problem from designated verifier CS-proofs. Cryptology ePrint Archive, Report 2011/456, 2011.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43:431–473, May 1996.

[GOS06a] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, CRYPTO '06, pages 97–111, 2006.

[GOS06b] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for NP. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, EUROCRYPT '06, pages 339–358, 2006.

[Gro09] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '09, pages 192–208, 2009.

[Gro10a] Jens Groth. Short non-interactive zero-knowledge proofs. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 341–358, 2010.

[Gro10b] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[GS06] Oded Goldreich and Madhu Sudan. Locally testable codes and pcps of almost-linear length. *Journal of the ACM*, 53:558–655, July 2006. Preliminary version in STOC '02.

[GSS12a] Craig Gentry, Halevi Shai, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *Proceedings of the 15th International Conference on Practice and Theory in Public Key Cryptography*, PKC '12, pages 1–16, 2012.

[GSS12b] Craig Gentry, Halevi Shai, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *Proceedings of the 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '12, pages 465–482, 2012.

[GVW02] Oded Goldreich, Salil Vadhan, and Avi Wigderson. On interactive proofs with a laconic prover. *Computational Complexity*, 11(1/2):1–53, 2002.

[GW12] Rosario Gennaro and Daniel Wichs. Fully homomorphic message authenticators. Cryptology ePrint Archive, Report 2012/290, 2012.

[Har04] Prahladh Harsha. *Robust PCPs of Proximity and Shorter PCPs*. PhD thesis, MIT, EECS, September 2004.

[HPV77] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. *Journal of the ACM*, 24(2):332–337, 1977.

[HS00] Prahladh Harsha and Madhu Sudan. Small PCPs with low query complexity. *Computational Complexity*, 9(3–4):157–201, Dec 2000. Preliminary version in STACS '91.

[Jon88] Douglas W. Jones. The ultimate RISC. *ACM SIGARCH Computer Architecture News*, 16:48–55, June 1988.

[Jon93] Neil D. Jones. Constant time factors do matter. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, STOC '93, pages 602–611, 1993.

[Kan05] Rajgopal Kannan. The KR-Beneš network: A control-optimal rearrangeable permutation network. *IEEE Transactions on Computers*, 54(5):534–544, 2005.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, 1992.

[Kol53] Andrey N. Kolmogorov. To the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 8(4):175–176, 1953.

[Kop10] Swastik Kopparty. Private communication, 2010.

[KRR12] Yael Kalai, Ran Raz, and Ron Rothblum. Where delegation meets Einstein. Isaac Newton Institute for Mathematical Sciences, Formal and Computational Cryptographic Proofs, 2012.

[KU58] Andrey N. Kolmogorov and Vladimir A. Uspenskiĭ. To the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 13(4):3–28, 1958. In Russian. English translation in in AMS Translations, ser. 2, vol. 21 (1963), 217Ð-245.

[KvLP88] Jyrki Katajainen, Jan van Leeuwen, and Martti Penttonen. Fast simulation of Turing machines by random access machines. *SIAM Journal on Computing*, 17(1):77–88, 1988.

[Lei92] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*, TCC '12, pages 169–189, 2012.

[LL92] Michael C. Loui and David R. Luginbuhl. Optimal on-line simulations of tree machines by random access machines. *SIAM Journal on Computing*, 21(5):959–971, 1992.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, Cambridge, UK, second edition edition, 1997.

[Mat08] Todd Mateer. *Fast Fourier Transform algorithms with applications*. PhD thesis, Clemson University, 2008.

[Mei12] Or Meir. Combinatorial pcps with short proofs. In *Proceedings of the 26th Annual IEEE Conference on Computational Complexity*, CCC '12, 2012.

[Mer89] Ralph C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference*, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[NRS94] Ashish V. Naik, Kenneth W. Regan, and D. Sivakumar. On quasilinear-time complexity theory. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '94, pages 325–349, 1994.

[NS82] David Nassimi and Sartaj Sahni. Parallel algorithms to set up the Beneš permutation network. *IEEE Transactions on Computers*, 31(2):148–154, 1982.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, STOC '89, pages 33–43, 1989.

[Ofm65] Yuri P. Ofman. A universal automaton. *Transactions of the Moscow Mathematical Society*, 14:200–215, 1965.

[OTW71] D. C. Opferman and N. T. Tsao-Wu. On a class of rearrangeable switching networks - part i: Control algorithm. *Bell System Technical Journal*, 50(5):1579–1600, 1971.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, USA, 1994.

[Pau78] Wolfgang J. Paul. *Komplexitätstheorie*. Teubner, Stuttgart, Germany, 1978.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *Journal of the ACM*, 26:361–381, April 1979.

[Pip73] Nicholas Pippenger. The complexity theory of switching networks. Technical Report 487, MIT Research Lab of Electronics, 1973.

[Pip77] Nicholas Pippenger. Superconcentrators. *SIAM Journal on Computing*, 6(2):298–304, 1977.

[PR79] W. J. Paul and R. Reischuk. On time versus space II. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, FOCS '79, pages 298–306, 1979.

[PS94] Alexander Polishchuk and Daniel A. Spielman. Nearly-linear size holographic proofs. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, STOC '94, pages 194–203, 1994.

[Rei93] John H. Reif. *Synthesis of parallel algorithms*. Morgan Kaufman, San Mateo, CA, USA, 1993.

[Rob86] J. M. Robson. Fast probabilistic RAM simulation of single tape turing machine computations. *Information and Control*, 63(1-2):67–87, 1986.

[Rob91] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, May 1991.

[Rob92] J. M. Robson. Deterministic simulation of a single tape turing machine by a random access machine in sub-linear time. *Information and Computation*, 99(1):109–121, 1992.

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, STOC '90, pages 387–394, 1990.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25:136–145, January 1978.

[Sch80] Arnold Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[SH86] Richard E. Stearns and Harry B. III Hunt. On the complexity of the satisfiability problem and the structure of NP. Technical Report 82-21, State University of New York at Albany, Computer Science Department, 1986.

[Sho88] Victor Shoup. New algorithms for finding irreducible polynomials over finite fields. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '88, pages 283–290, 1988.

[Sho99] Victor Shoup. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation*, ISSAC '99, pages 53–58, New York, NY, USA, 1999. ACM.

[Shp96] Igor Shparlinski. On finding primitive roots in finite fields. *Theoretical Computer Science*, 157(2):273–275, 1996.

[Spi95] Daniel Spielman. *Computationally Efficient Error-Correcting Codes and Holographic Proofs*. PhD thesis, MIT, Mathamatics Department, May 1995.

[Tho78] C. D. Thompson. Generalized connection networks for parallel processor intercommunication. *IEEE Transactions on Computers*, 27(12):1119–1125, 1978.

[VL88] Ramarathnam Venkatesan and Leonid Levin. Random instances of a graph coloring problem are hard. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 217–222, 1988.

[vzGG96] Joachim von zur Gathen and Jürgen Gerhard. Arithmetic and factorization of polynomial over f_2. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ISSAC '96, pages 1–9, New York, NY, USA, 1996. ACM.

[vzGG03] Joachim von zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[Wak68] Abraham Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[Wee05] Hoeteck Wee. On round-efficient argument systems. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, ICALP '05, pages 140–152, 2005.