Journal of Computer and System Sciences 123 (2022) 122–142

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Word equations in non-deterministic linear space [☆]

Artur Jeż

University of Wrocław, Poland

ARTICLE INFO

Article history:
Received 19 October 2020
Received in revised form 27 July 2021
Accepted 6 August 2021
Available online 1 September 2021

Keywords:
Word equations
String unification
Context-sensitive languages
Space efficient computations
Linear space

ABSTRACT

Satisfiability of word equations problem is: Given two sequences consisting of letters and variables decide whether there is a substitution for the variables that turns this equation into true equality. The exact computational complexity of this problem remains unknown, with the best lower and upper bounds being, respectively, NP and PSPACE. Recently, the novel technique of recompression was applied to this problem, simplifying the known proofs and lowering the space complexity to (non-deterministic) $\mathcal{O}(n \log n)$. In this paper we show that satisfiability of word equations is in non-deterministic linear space, thus the language of satisfiable word equations is context-sensitive. We use the known recompression-based algorithm and additionally employ Huffman coding for letters. The proof, however, uses analysis of how the fragments of the equation depend on each other as well as a new strategy for non-deterministic choices of the algorithm.

1. Introduction

Solving *word equations* has been an intriguing problem since the dawn of computer science, motivated first by its ties to Hilbert's 10th problem. Initially it was conjectured that this problem is undecidable, which was disproved in a seminal work of Makanin [15]. At first little attention was given to computational complexity of Makanin's algorithm and the problem itself; these questions were reinvestigated in the '90 [10,24,14], culminating in the EXPSPACE implementation of Makanin's algorithm by Gutiérrez [9].

The connection to compression was first observed by Rytter and Plandowski [21], who showed that a length-minimal solution of size N has a compressed representation of size at most polynomial in n and $\log N$, yet the proposed algorithm still used the bound on the size of the smallest solution following from Makanin's algorithm. Plandowski further explored this approach [19] and proposed a PSPACE algorithm [18], which is the best computational complexity class upper bound up to date; a simpler PSPACE solution also based on compression was proposed by Jeż [11]. On the other hand, this problem is only known to be NP-hard, and it is conjectured that it is in NP.

The importance of these mentioned algorithms lays also with the possibility to extend them (in nontrivial ways) to various scenarios: free groups [16,5,7], representation of all solutions [20,11,22], traces [17,6], graph groups [8], terms [13], context free groups [4], hyperbolic groups [23,1], and others.

While the computational complexity of word equations remains unknown, its exact space complexity is intriguing as well: Makanin's algorithm uses exponential space [9], Plandowski [18] gave no explicit bound on the space usage of his (non-deterministic) algorithm, a rough estimation is $\mathcal{O}(n^5)$; here n denotes the length, i.e. the number of letters and variables, of the input equation. The recent (also non-deterministic) solution of Jeż [11] uses $\mathcal{O}(n \log n)$ space. Moreover, for

☆ A conference version of this paper was presented at ICALP 2017 [12].

E-mail address: aje@cs.uni.wroc.pl.

https://doi.org/10.1016/j.jcss.2021.08.001
0022-0000/© 2021 Elsevier Inc. All rights reserved.

$\mathcal{O}(1)$ variables a linear bound on space usage was shown [11]; recall that languages recognisable in non-deterministic linear space are exactly the context-sensitive languages.

In this paper we show that satisfiability of word equations can be tested in non-deterministic linear space. As customary for linear-space algorithm, we count the space usage in bits, i.e. the (non-deterministic) algorithm uses $\mathcal{O}(m)$ bits, where the input equation uses m bits. This shows that the language of satisfiable word equations is context-sensitive (and by the famous Immerman–Szelepcsényi theorem: also the language of unsatisfiable word equations is context-sensitive). The employed algorithm is a (variant of) algorithm of Jeż [11], which additionally uses Huffman coding for letters in the equation. On the other hand, the actual proof uses a different encoding of letters, which extends the ideas used in a (much simpler) proof in case of $\mathcal{O}(1)$ variables [11, Section 5], i.e. we encode the letters in the equation using factors of the original equations on which those letters "depend". The exact notion of this "dependence" is defined and analysed. The other new ingredient is a different strategy of compression: roughly speaking, previously a strategy that minimised the length of the equation was used. Here, a more refined strategy is used: it simultaneously minimises the size of a particular bit encoding, enforces that changes in the equation (during the algorithm) are local, and limits the amount of new letters that are introduced to the equation.

The bound holds when letters and variables in the input are encoded using an arbitrary uniquely decodable code, so in particular, the Huffman coding (so the most efficient one among uniquely decodable codes); in essence, Huffman coding represents each letter of the input alphabet using strings of bits of varying lengths, depending on the frequencies of the letters, where a letter of frequency f is assigned a bit string of length roughly $-\log f$.

The paper is organised as follows: Section 2 recalls the definitions of word equations, space-complexity classes and codings. Section 3 recalls the known algorithm for word equations and its main properties. The last Section 4 gives the proof of the linear-space usage (for appropriate coding) and wraps up the whole proof; this section is the main input of this paper.

A conference version of this paper was presented at ICALP 2017 [12]. The journal version contains omitted proofs and improvements on notation and presentation (in particular the treatment of coding theory and linear space computation) as well as fixes some minor errors.

2. Notions

2.1. Codes

An alphabet is a finite set of symbols Γ and Γ^* is a set of all finite words over this alphabet, i.e. finite sequences of letters from Γ. A code $\mathcal{C}$ for and alphabet Γ assigns to each letter $a \in \Gamma$ a finite string of bits, formally $\mathcal{C} : \Gamma \to \{\mathtt{0}, \mathtt{1}\}^*$. A code is naturally extended to words from Γ^*, i.e. $\mathcal{C} : \Gamma^* \to \{\mathtt{0}, \mathtt{1}\}^*$ (by concatenation); this is often called an extended code in the literature. When $\mathcal{C}$ is known from the context, the $\mathcal{C}^*(w)$ is called the encoding of w. A code is uniquely decodable, when the $\mathcal{C}$ is an injection on Γ^*, or informally, encodings of two different words are different. A natural code for Γ assigns consecutive bitstrings of length $\lceil \log |\Gamma| \rceil$ to each symbol in Γ. A Huffman code $\mathcal{C}_{H,w}$ is a way of creating a uniquely decodable code given a word $w \in \Gamma^*$. It is a folklore fact, see for instance [2, Theorem 17.4], that Huffman code is optimal, i.e. given a word w the length $|\mathcal{C}_{H,w}(w)|$ is smallest among the lengths $|\mathcal{C}'(w)|$ over all uniquely decodable codes C' (for Γ). Note that the Huffman code $\mathcal{C}_{H,w}$ for w depends on w, so the universally quantified optimality should not be particularly surprising. Moreover, the optimality of Huffman coding is often stated in a more general meaning [3, Theorem 5.8.1], in which the word (of length n) is a random variable with some fixed distribution of letters (independently on each position) and the optimality means that the expected length of the coding is smallest possible among all uniquely decodable codes; such a statement is more general, as a word w defines a random variable (whose frequency of letters is the same as the frequency in w) and then the expected length of code for a random word of length $|w|$ is exactly the length of the encoding of w.

The Huffman coding can be efficiently constructed, the construction is recalled in proof of Lemma 6.

By $\|w\|_{\mathcal{C}}$ we denote the bit-length of encoding $\mathcal{C}^*(w)$; the code $\mathcal{C}$ shall be always clear from the context, and so we write $\|w\|$.

2.2. Word equations

A word equation is a pair (U, V), written as $U = V$, where $U, V \in (\Gamma \cup \mathcal{X})^*$ and Γ and $\mathcal{X}$ are disjoint alphabets of *letters* and *variables*, both are collectively called *symbols*. By n_X we denote the number of occurrences of X in the (current) equation; in the algorithm n_X does not change till X is removed from the equation, in which case n_X becomes 0. A *substitution* is a morphism $S : \mathcal{X} \cup \Gamma \to \Gamma'^*$, where $\Gamma' \supseteq \Gamma$ and $S(a) = a$ for every $a \in \Gamma$, a substitution naturally extends to $(\mathcal{X} \cup \Gamma)^*$. A *solution* of an equation $U = V$ is a substitution S such that $S(U) = S(V)$; given a solution S of an equation $U = V$ we call $S(U)$ the *solution word*. We allow the solution to use letters that are not present in the equation, this does not change the satisfiability: all such letters can be changed to a fixed letter from Γ (or to ϵ), and the obtained substitution is still a solution. Yet, the proofs become easier, when we allow the usage of such letters. The alphabet Γ' is usually given implicitly: as the set of letters used by the substitution. A *substring* denotes a (contiguous) sequence of letters, while a *factor*: a (contiguous) sequence of letters and variables; in both cases, usually the ones occurring in the equation. A *block* is a string a^ℓ with $\ell \geq 1$ that cannot be extended to the left nor to the right with a.

For technical reasons we insert into the equation ending markers at the beginning and end of U and V, i.e. write them as $@U@, @V@$ for some special symbol @. Those markers are ignored by the algorithm, yet they are needed for the encoding. This increases the encoding size by a constant (multiplicative) factor.

2.3. Non-deterministic computation

The model of computation is a non-deterministic Turing Machine with tape alphabet $\{0, 1\}$. A word $w \in \{0, 1\}^*$ is accepted by a machine M if M on input w for some non-deterministic choices gets to an accepting state; it is rejected if it gets to the rejecting state for all computations. A set P is recognized in non-deterministic linear space if there exists a non-deterministic Turing Machine M, which given $w \in \{0, 1\}$ accepts w if and only if $w \in P$ and it uses $\mathcal{O}(|w|)$ space.

Alternatively, we can assume that w is given to the Turing Machine using a larger alphabet Γ and the tape-alphabet is fixed $\Gamma' \supseteq \Gamma$; those definitions are equivalent.

We also allow that the input is given using a different (smaller) encoding of w, which may depend on w, this leads to a potentially more restrictive class.

The model of linear-bounded computation is flexible enough, so that we specify the actions of the algorithm using usual declarative programming pseudocode, this can be encoded using the Turing machine model, assuming that the space usage (in bits) is linear.

When proving that a set P is recognised in non-deterministic linear space it is enough to show that some computation for $w \in P$ (i.e. for some non-deterministic choices) uses at most $c|w|$ space and that no computation for $w \notin P$ accepts (i.e. we allow that it exceeds the space bound or does not terminate): we can add to the machine a counter that counts from 0 to $2^{c|w|} \times c|w| \times q$, where q is the number of states of the machine, in binary, when the counter is full then we reject. In this way the space bound is at most $2 \cdot c|w| + \log(cq|w|)$, the computation always terminates and exactly the same words are accepted.

Lemma 1. *For a set P and Turing Machine M if for each $w \in P$ there are non-deterministic choices for which M accepts using $\mathcal{O}(|w|)$ space and M does not accept any $w \notin P$ then P is recognized in non-deterministic linear space.*

In our case P is the set of satisfiable word equations and the instance is an equation, in the following the definitions are given with regard to this problem. We need some encoding of the equation into bitstrings. We assume only that the equation is given by a fixed uniquely decodable code, so without loss of generality the Huffman code, as it could be computed in linear space, see Lemma 6. This setting is more restrictive than using the natural code for the equation. Observe also that it is important, which symbols in U, V are variables, and which letters. This can be encoded using $|UV|$ bits, so increases the space consumption by a small constant.

A non-deterministic procedure is *sound*, when given an unsatisfiable word equation $U = V$ it cannot transform it to a satisfiable one, regardless of the non-deterministic choices; a procedure is *complete*, if given a satisfiable equation $U = V$ for some non-deterministic choices it returns a satisfiable equation $U' = V'$. It is easy to observe that a composition of procedures, i.e. a sequence of procedures, the next performed on the output of the previous, that are all sound (complete) yields a procedure that is sound (complete, respectively). Note that when there is no solution, then a sound procedure is guaranteed not to accept. The soundness of the presented algorithm is easy to establish, and so we will focus on the algorithm working on a satisfiable word equation. Taking into the account that it is enough to bound the space usage of some particular non-deterministic choices, we imagine our algorithm as if it knew the solution of the equation and made the appropriate non-deterministic choices and we bound the space only in the case of such non-deterministic choices. This is formalized in the following section.

3. The (known) algorithm

We use (a variant of) recompression algorithm [11], the proofs of correctness are omitted, as they are available, yet they are intuitively clear. The algorithm conceptually applies the following two operations on $S(U)$ and $S(V)$: given a string w and alphabets $\Gamma, \Gamma_\ell, \Gamma_r$, where Γ_ℓ and Γ_r are disjoint:

- the Γ block compression of w is a string w' obtained by replacing every block a^ℓ in w, where $a \in \Gamma$ and $\ell \geq 2$, with a fresh letter a_ℓ;
- the (Γ_ℓ, Γ_r) pair compression of w is a string w' obtained by replacing every occurrence of a pair $ab \in \Gamma_\ell\Gamma_r$ with a fresh letter c_{ab}.

A *fresh* letter means that it is not currently used in the equation, neither in Γ (for Γ block compression) or not in $\Gamma_\ell \cup \Gamma_r$ (for (Γ_ℓ, Γ_r) pair compression), yet each occurrence of a fixed ab is replaced with the same letter and similarly, in Γ block compression each occurrence of a block a^ℓ is replaced with the same letter. The a_ℓ and c_{ab} are just notation conventions, the actual letters in w' do not store the information how they were obtained. For brevity, we refer to Γ block compression as block compression, when Γ is clear from the context or unimportant. Similarly, (Γ_ℓ, Γ_r) pair compression is also called pair compression, when (Γ_ℓ, Γ_r) is clear from the context or unimportant. We say that a pair $ab \in \Gamma_\ell\Gamma_r$ is *covered* by a partition Γ_ℓ, Γ_r.

The intuition is that the algorithm aims at performing those compression operations on the solution word and to this end it modifies the equation a bit and then performs the compression operations on U and V (and conceptually also on the solution, i.e. on $S(X)$ for each variable X). Below we describe, how it is performed on the equation. For simplicity of description, we assume that for each variable X the substitution $S(X)$ is nonempty, i.e. $S(X) \neq \epsilon$. This can be ensured for the input equation by guessing the set of variables violating this condition and removing of such variables from the equation. Afterwards, the algorithm non-deterministically guesses when $S(X)$ becomes ϵ and removes it.

BlockComp: For the equation $U = V$ and the alphabet Γ of letters in this equation (except the ending markers) for each variable X we first guess the first and last letter of $S(X)$ as well as the lengths ℓ, r of the longest prefix consisting only of a for some $a \in \Gamma$, called an a-prefix, and a b-suffix (defined similarly for some $b \in \Gamma$) of $S(X)$. Then we replace X with $a^\ell X b^r$ (or $a^\ell b^r$ or a^ℓ when $S(X) = a^\ell b^r$ or $S(X) = a^\ell$); this operation is called *popping an a-prefix and a b-suffix*. Then we perform the Γ-block compression on the equation (this is well-defined, as we can treat variables as symbols from outside Γ).

Algorithm 1 BlockComp(Γ).

```
Require: $\Gamma$ is the set of letters in $U = V$
 1: for $X \in \mathcal{X}$ do
 2:     let $a, b$ be the first and last letter of $S(X)$
 3:     guess $\ell \geq 1, r \geq 0$                  ▷ $S(X) = a^\ell w b^r$, where $w$ does not begin with $a$ nor end with $b$
 4:                                                   ▷ If $S(X) = a^\ell$ then $r = 0$
 5:     replace each $X$ in $U$ and $V$ by $a^\ell X b^r$       ▷ $S(X) = a^\ell w b^r$ changes to $S(X) = w$
 6:     if $S(X) = \epsilon$ then remove $X$ from $U$ and $V$   ▷ Guess
 7: for each letter $a \in \Gamma$ and each $\ell \geq 2$ do
 8:     replace every block $a^\ell$ in $U$ and $V$ by a fresh letter $a_\ell$
```

PairComp: For the alphabet Γ, which will always be the alphabet of letters in the equation right before the preceding block compression (again: except the ending markers), we partition Γ into Γ_ℓ and Γ_r and then for each variable X guess whether $S(X)$ begins with a letter $b \in \Gamma_r$ and if so, replace X with bX or b, when $S(X) = b$, and then do a symmetric action for the last letter and Γ_ℓ; this operation is later referred to as *popping letters*. Then we perform the (Γ_ℓ, Γ_r) pair compression on the equation. Note that the choice of the partition of Γ to Γ_ℓ and Γ_r affects the space computation and the construction of appropriate partitions is one of the main technical input of the paper, details are given in Section 4.2.

Algorithm 2 PairComp(Γ_ℓ, Γ_r), $\Gamma = \Gamma_\ell \cup \Gamma_r$.

```
Require: $\Gamma_\ell$, $\Gamma_r$ are disjoint
 1: for $X \in \mathcal{X}$ do
 2:     let $b$ be the first letter of $S(X)$                       ▷ Guess
 3:     if $b \in \Gamma_r$ then
 4:         replace each $X$ in $U$ and $V$ by $bX$           ▷ Implicitly change $S(X) = bw$ to $S(X) = w$
 5:         if $S(X) = \epsilon$ then
 6:             remove $X$ from $U$ and $V$ and proceed to next variable in $\mathcal{X}$   ▷ Guess
 7:     let $a$ be the last letter of $S(X)$
 8:     if $a \in \Gamma_\ell$ then
 9:         replace each $X$ in $U$ and $V$ by $Xa$           ▷ Implicitly change $S(X) = wa$ to $S(X) = w$
10:         if $S(X) = \epsilon$ then
11:             remove $X$ from $U$ and $V$                  ▷ Guess
12: for $ab \in \Gamma_\ell\Gamma_r$ do
13:     replace each substring $ab$ in $U$ and $V$ by a fresh letter $c$
```

LinWordEqSat works in *phases*, until an equation with both sides of length 1 is obtained: in a single phase it establishes the alphabet Γ of letters in the equation, performs the Γ block compression and then repeats: guess the partition of Γ to Γ_ℓ and Γ_r and perform the (Γ_ℓ, Γ_r) pair compression, until each pair $ab \in \Gamma^2$ was covered by some partition. When both sides are reduced to length 1, then we reject when those are two different letters, and otherwise we accept.

Algorithm 3 LinWordEqSat.

```
 1: while $|U| > 1$ or $|V| > 1$ do
 2:     $\Gamma \leftarrow$ letters in $U = V$
 3:     BlockComp($\Gamma$)
 4:     while some pair in $\Gamma^2$ was not covered do
 5:         partition $\Gamma$ to $\Gamma_\ell$ and $\Gamma_r$          ▷ Guess
 6:         PairComp($\Gamma_\ell, \Gamma_r$)
 7: if $U$, $V$ are different letters then
 8:     reject
 9: else
10:     accept
```

Correctness Informally, the non-deterministic choices fall into two categories: there is the *partition choice*, i.e. given Γ we partition it into Γ_ℓ, Γ_r, and there are the other choices. The former affects the space usage, i.e. proper partition choice will guarantee small space usage (this is shown in Section 4), the other are done to guarantee completeness (of the non-deterministic procedure). Moreover, for appropriate choices of the second type the procedure transforms the solutions in the sense described in the below Lemma 2.

Lemma 2 *([11, Lemma 2.8 and Lemma 2.10]).* BlockComp *is sound. It is complete, moreover, for any solution S of an equation $U = V$, such that $S(U) \in \Gamma^*$ where Γ is the set of letters in the equation, for some non-deterministic choices the returned equation $U' = V'$ has a solution S' such that $S'(U')$ is the Γ block compression of $S(U)$ and $S'(X)$ is obtained from $S(X)$ by removing the a-prefix and b-suffix, where a is the first letter of $S(X)$ and b the last, and then performing the Γ block compression.*

When Γ_ℓ and Γ_r are disjoint, the PairComp(Γ_ℓ, Γ_r) *is sound. It is complete, moreover, for any solution S of an equation $U = V$ and disjoint sets Γ_ℓ and Γ_r for some non-deterministic choices the returned equation $U' = V'$ has a solution S' such that $S'(U')$ is the (Γ_ℓ, Γ_r) pair compression of $S(U)$ and $S'(X)$ is obtained from $S(X)$ by removing the first letter of $S(X)$, if it is in Γ_r, and the last, if it is in Γ_ℓ, and then performing the (Γ_ℓ, Γ_r) pair compression.*

The non-deterministic choices in Lemma 2 are called *corresponding* (to S, in case of block compression; to S and partition Γ_ℓ, Γ_r in case of pair compression). Those choices are intuitively clear: all of them are described in Algorithms 1–3 'as if' the solution were given explicitly (for instance, we replace X with $a^\ell X b^r$ when $S(X)$ has an a-prefix a^ℓ and a b-suffix b^r). Similarly, the solution S' from Lemma 2 is called a *solution corresponding to* S after (Γ_ℓ, Γ_r) pair compression (Γ block compression, respectively); we also talk about a *solution corresponding to* S, when the compression operation is clear from the context and extend this notion to a solution corresponding to S after a series of compressions (in one phase), note that this implicitly takes into account the appropriate partitions. What is important later on is how $S'(X)$ is obtained from $S(X)$: it is modified as if the subprocedures knew the first/last letter of $S(X)$ and popped appropriate letters from the variables and then compressed pairs/blocks.

Note also that Lemma 2 treats completeness more carefully than soundness: we do not claim that if the new equation $U' = V'$ has a solution S' then there is a solution S of the original equation $U = V$ such that S' corresponds to S. In fact this is the case [11], but this is not needed for the purpose of this paper.

Lastly, observe that Lemma 2 treats the alphabet Γ in block compression and in pair compression differently. This is because Γ is computed as the set of letters right before block compression, so we may assume that this is indeed this set and considering only solution over this alphabet simplifies the algorithm. On the other hand, as pair compressions are after the block compression, we no longer can assume that Γ is still the set of letters in the equation and need to deal with the more general case.

Lemma 2 yields the soundness and completeness of LinWordEqSat. For the termination we observe that iterating the compression operations shortens the string by a constant fraction, thus the length of a solution word shortens by a constant fraction in each phase.

Lemma 3. *Let w be a string over an alphabet Γ and w' a string obtained from w by a Γ block compression followed by a sequence of (Γ_ℓ, Γ_r) pair compressions (where Γ_ℓ, Γ_r is a partition of Γ) such that each pair $ab \in \Gamma^2$ is covered by some partition. Then $|w'| \le \frac{2|w|+1}{3}$.*

Proof. First observe that the claim holds when $|w| \le 2$: if $|w| = 1$ then it trivially holds: $\frac{2|w|+1}{3} = |w|$. When $w = 2$ then we will perform some compression on w and so $|w'| = 1 \le \frac{2|w|+1}{3} = \frac{5}{3}$.

Consider two consecutive letters a, b in w. At least one of those letters is compressed during the procedure:

- *if $a = b$:* In this case they are compressed during the Γ block compression.
- *$a \ne b$:* At some point the pair ab is covered by some (Γ_ℓ, Γ_r) pair compression. If any of the letters a, b was already compressed, then we are done. Otherwise, this occurrence of ab is now compressed.

Hence, each uncompressed letter in w (except perhaps the last letter) can be associated with the two letters to the right that are compressed. This means that (in a phase) at least $\frac{2}{3}(|w| - 1)$ letters are compressed and so $|w'| \le |w| - \frac{1}{3}(|w| - 1)$, as claimed. □

Theorem 1. LinWordEqSat *is sound, complete and terminates (for appropriate non-deterministic choices) for satisfiable equations. It runs in linear space.*

There is a non-deterministic algorithm that decides word equations in linear-space. It always terminates and never exceed the linear-space bound.

The proof of the first claim (for LinWordEqSat) is given in Section 4.3, the second is obtained by unwrapping the definitions of soundness and completeness and the application of Lemma 1, which guarantee that weak termination conditions and weak space usage guarantees can be turned into strong ones.

In the following, we will also need one more technical property of block compression.

Lemma 4. *Consider a solution S at a beginning of a phase and the corresponding solution S' of $U' = V'$ after the block compression and some pair compressions. Then $S'(U')$ has no two consecutive letters $aa \in \Gamma$.*

This is true after block compression and afterwards we only replace substrings of letters from Γ with letters outside Γ, so the claim clearly holds. Note that this does not hold for arbitrary letters, i.e. outside Γ.

Compressing blocks in small space Storing, even concisely, the lengths of popped prefixes and suffixes in Γ block compression makes attaining linear space difficult. This was already observed [11] and a linear-space implementation of BlockComp is known [11]. It performs a different set of operations, yet the effect is the same as for BlockComp. Instead of explicitly naming the lengths of blocks, we treat them as integer parameters; then we declare, which maximal blocks are of the same length (those lengths depend linearly on the parameters); verifying the validity of such a guess is done by writing a system of (linear) Diophantine equations that formalise those equalities and checking its satisfiability. This procedure is described in detail in [11, Section 4]. In the end, it can be implemented in linear space.

Lemma 5 *([11, Lemma 4.7]).* BlockComp *can be implemented in space linear in the size of an encoding (using a uniquely decodable code) of the equation.*

Encoding the equation At each step of the algorithm we encode letters (though not variables) in the equation using Huffman coding. The variables are encoded using the original coding from the input equation. To distinguish between the two codes, we prefix each code for a letter with `0` and for a variable with `1`. This increases the space usage at most twice when compared to the usage without this extra bit, so we disregard it, as we aim for linear space (with an arbitrary constant). Also, the space usage for variables is at most $2\|U_0V_0\|$, where $U_0 = V_0$ is the input equation and $\|U_0V_0\|$ is the size of the encoding of the input equation. Thus, we disregard later on the space used for variables and focus on the space usage of the letters.

Using the Huffman coding of the letters may mean that when going from $U = V$ to $U' = V'$ the encoding of letters changes and in fact using the former encoding in the latter equation may lead to super-linear space (imagine that we pop from each variable a letter that has a very long code). Using standard methods, changing the encoding during a transition from $U = V$ to $U' = V'$ can be done in space linear in the input encoding size plus the output encoding size, which may use a different code.

Lemma 6. *Given a string (encoded using some uniquely decodable code), its Huffman coding can be computed in linear space.*

Each subprocedure of LinWordEqSat *that transforms an equation $U = V$ to $U' = V'$ can be implemented in space $\mathcal{O}(\|U = V\|_1 + \|U' = V'\|_2)$, where $\|\cdot\|_1$ and $\|\cdot\|_2$ are the Huffman codings for letters associated with the equations $U = V$ and $U' = V'$, respectively.*

Proof. Observe that as the code is uniquely decodable, there is exactly one partition of the input bitstring into codewords. It can be found in linear space by simple testing all the possibilities (or it can be guessed using non-determinism).

In the following, we will use some 'fresh symbols'. To guarantee that they have short codes, we prefix each code used in the string by, say, `0` and each fresh symbol by `1`. This will keep the space linear. In the end this ad-hoc encoding is replaced with a Huffman so all those extra bits are no longer used.

A standard implementation of the Huffman coding [2, Theorem 17.4] firstly calculates for each symbol in the string the number of its occurrences, this can be done in linear space, as a symbol plus number of its occurrences takes space linear in the space taken by all those occurrences. Then it iteratively builds an edge-labelled (with $\{0, 1\}$) tree with leaves corresponding to original letters. The labels on the path from the root to a leaf a give a code for a. The algorithm takes two letters with the smallest number of occurrences, creates a new node (which is treated further on as a leaf), attaches the two nodes to the new node and labels the new edges with `0` and `1`. This is iterated till one node is obtained. The tree uses linear space in total, and otherwise the used space only decreases. It is easy to see that the whole computation can be done in linear space.

For block compression, Lemma 5 already states that it can be performed in space linear in the encoding size of the old equation, the Huffman coding of the new one can be then computed in linear space.

For pair compression, we first prefix all old symbols with `0` and all introduced ones are prefixed by `1`, this increases the total space consumption at most twice. The following issues should be addressed

- how to encode letters popped from variables: it could be that a letter with long code is introduced in many copies;
- how to encode the letters obtained after the compression.

All other letters existed in the old equation, so their total space usage is taken into the account. For letters popped in (Γ_ℓ, Γ_r) pair compression from variable X to the left and right we use fresh symbols $X\#0$ and $X\#1$, which are encoded as X plus $\mathcal{O}(1)$, we also give a list of all such letters that are equal, and for each group we also give the letter from the equation to which they are equal. This uses as much space as the input equation times (at most) a constant factor. Finally, we can replace all codes that represent the same letter with the shortest among them; this clearly decreases the space usage. For compression itself, when ab is compressed, we encode it as (ab), where '(' and ')' are fresh symbols (though the same are used for each such encoding) and a, b are encoded as before. Clearly, the space usage grows by at most a constant factor. We finally compute the Huffman coding of the new equation and translate the old encoding to the new. □

4. Space consumption

In order to bound space consumption, we will analyse an encoding of letters that depends on the current equation. We use the term 'encoding' even though it may assign different codes to different occurrences of the same letter, but two different letters never have the same code; this is fine as alternatively we can replace all different codes for one letter with the shortest one, obtaining an encoding that is not-longer. Since we are interested in showing a linear space bound, we do not care about the multiplicative $\mathcal{O}(1)$ factors in the space consumption and can ensure that our code is a prefix code (i.e. no codeword is a prefix of another) and that there $\mathcal{O}(1)$ codewords of length $\mathcal{O}(1)$ that are not used in the equation: say, for the encoding presented later on we replace each `0` with `00` and `1` with `01` and additionally terminate each code with `11`; then the additional codewords can begin with `10` and terminate with `11` as well. We show that such an encoding uses linear space, which also shows that the Huffman encoding of the letters in the equation uses linear space, as Huffman code uses the smallest space among the prefix codes [3, Theorem 5.8.1], [2, Theorem 17.4]. In particular, our specially designed coding is used only for the purpose of the proof, it is not used by the algorithm.

The idea of the encoding is: for each letter in the current equation we establish an interval I of indices in the original equation (viewed as a string $U_0V_0[1 \,..\, |U_0V_0|]$) on which it 'depends' (this is formalised in Section 4.1) and encode this letter as $U_0V_0[I]\#i$, when it is the ith in the sequence of letters assigned I and $U_0V_0[I]$ is the factor of the original equation restricted to indices in I. We prove that letters given the same encoding are indeed the same. This is nontrivial, as it may be that for two different intervals I, I' we have $U_0V_0[I] = U_0V_0[I']$ and the encoding include only factors $U_0V_0[I]$ or $U_0V_0[I]$ and does not include the endpoints of the intervals I, I'. For the space bound, we separately count the space used by all $U_0V_0[I]$s and separately the space used for numbers (i.e. the ones after the # sign in the encoding). We show that using a random partition each time we can guarantee that on average a single symbol α from the original equation is used in constant number of Is (as α may have different encodings, we also need to weight this using the $\|\alpha\|$). For the numbers in the encodings (i.e. the ones after the #), the argument is similar: for a given I the average amount of different letters encoded as $I\#i$ is $\mathcal{O}(1)$ and so the total space consumption is linear (the actual calculations are a bit more tricky here). The dependency is formalised in Section 4.1, while Section 4.2 first gives the high-level intuition and then upper-bound on the used space. The encoding "$I\#i$" is still too abstract to be actually used (it uses a distinct symbol #, there is no way to determine where i ends, etc.) so the concrete encoding is a bit longer and more complex. This is given in detail in Lemma 11, however, the actual low-level encoding does not influence the whole argument and it is easier to argue on the more abstract level.

4.1. Dependency intervals

The input equation is denoted by $U_0 = V_0$, the $U = V$ and $U' = V'$ are used for the current equation and equation after performing some operation. We treat the input equation as a single string U_0V_0 and consider its *indices*, i.e. numbers from 1 to $|U_0V_0|$, denoted by letters i, i', j and intervals of such indices, denoted by letter I, I' or $[i \,..\, j]$. The $U_0V_0[I]$ and $U_0V_0[i \,..\, j]$ denotes the factor of U_0V_0 restricted to indices in I or in $[i \,..\, j]$, note that this factor may include variables. We use a partial order $\leq$ on intervals: $[i \,..\, j] \leq [i' \,..\, j']$ if $i \leq i'$ and $j \leq j'$. Note that this is *not* the inclusion of sets, which we will also use.

In the current equation, i.e. the one stored by LinWordEqSat, we do not consider indices but rather *positions* and denote them by letters p, q. We do not think of them as numbers but rather as pointers: when $U = V$ is transformed by some operation to $U' = V'$ but the letter/variable at position p was not affected by this transformation, we still say that this letter/variable is at position p. On the other hand, the affected letters are on positions that were not present in $U = V$. In the same spirit, and using a similar notation, we will consider positions also in $S(U) = S(V)$. We still use the left-to-right ordering on positions, use $p - 1$ and $p + 1$ to denote the previous and next position. For a position p of a letter in $U = V$ we often consider the corresponding position in $S(U) = S(V)$. We denote such a position by $S(p)$, so that the notation resembles the move from $U = V$ to $S(U) = S(V)$. For instance, $S(p + 1)$ and $S(p) + 1$ denote the same position, assuming that $p, p + 1$ are both positions of a letter. We also consider intervals of positions, yet they are used rarely so that they are not confused with intervals of indices, on which we focus mostly. Given an equation $U = V$ and an interval of positions P by $UV[P]$ we denote the factor of letters and variables at positions in P, again, this notation is used rarely. In the input equation the index and position is the same.

With each position p in the (current) equations (including the endmarkers and the variables) we associate an interval of indices $\text{depint}(p)$ in U_0V_0, called *dependency interval* or *depint* for short. As intervals of indices are used mostly in context of dependency intervals, we call them all depints for short. If the depint is a single index $\{i\}$, we denote it i. The idea is that knowing $U_0V_0[\text{depint}(p)]$ and the non-deterministic choices of the algorithm we can uniquely determine, what is the letter at position p, in particular, if $U_0V_0[\text{depint}(p)] = U_0V_0[\text{depint}(p')]$ then the letters at those positions p and p' are the same. Note that $U_0V_0[\text{depint}(p)]$ may include both variables and letters. We use the notions of $\subseteq$ and $\supseteq$ for the depints with a usual meaning; we take unions of them, denoted by $\cup$, but only when the result is an interval (the fact that the union is an interval is always shown, whenever the union is used). We say that I and I' are similar, denoted as $I \sim I$, if $U_0V_0[I] = U_0V_0[I']$, note that the $U_0V_0[I]$, $U_0V_0[I']$ here are factors of the input equation.

To get some rough intuition: if we have several occurrences of XwY in the equation, then as long as X and Y are not removed, the word between them changes in the same way and so the letters between X and Y have depints that are factors of XwY. When Y is removed, we have to resort to the next variable, say Z and gradually enlarge the depint to $XwYvZ$.

Given an interval I of indices in U_0V_0 by $\text{Pos}_=(I)$ we denote positions in the current equation whose depint is I, i.e. $\text{Pos}_=(I) = \{p \mid \text{depint}(p) = I\}$. In the analysis it is also convenient to look at positions whose depint is a superset of I: $\text{Pos}_\supseteq(I) = \{p \mid \text{depint}(p) \supseteq I\}$, this is usually used for $I = \{i\}$. Both notation are designed to convey their meaning: $\text{Pos}_=(I)$: *positions*, whose depint "$= I$" and $\text{Pos}_\supseteq(I)$: *positions*, whose depint "$\supseteq I$." Note that we also consider intervals of position in $S(U) = S(V)$, in such a case we will use the notation $S(\text{Pos}_=(I))$ and $S(\text{Pos}_\supseteq(I))$ with the obvious meaning (sets of positions in $S(U) = S(V)$ corresponding to $\text{Pos}_=(I)$ and $\text{Pos}_\supseteq(I)$ in $U = V$, respectively). Note that $S(\text{Pos}_= I)$ and $S(\text{Pos}_\supseteq I)$ may include positions that come from substitution for a variable, in which case the "corresponding positions" in $U = V$ and $S(U) = S(V)$ are not in a bijection.

We shall ensure the following properties

(I1) Given a depint I, the $\text{Pos}_=(I)$ is a (perhaps empty) interval of positions, similarly $\text{Pos}_\supseteq(I)$.
(I2) Given depints I, I' such that $\text{Pos}_=(I) \neq \emptyset \neq \text{Pos}_=(I')$ either $I \leq I'$ or $I \geq I'$.
(I3) If $I \sim I'$ then the factors induced by the intervals of positions $\text{Pos}_=(I)$ and $\text{Pos}_=(I')$ are the same, i.e. $UV[\text{Pos}_=(I)] = UV[\text{Pos}_=(I')]$.

Encoding of letters Letters in $\text{Pos}_=(I)$ are encoded from left-to-right as $U_0V_0[I]\#1$, $U_0V_0[I]\#2$, etc. Note, that there is no a priori bound on the size of used numbers, i.e. the ones following #. Furthermore, if $I' \sim I$ then encoding $I\#i$ and $I'\#i$ is the same, this is fine, as by (I3) these symbols are equal.

Assigning depints to letters The original positions in the equation have depints equal to themselves, i.e. for position p in U_0V_0 we have $\text{depint}(p) = \{p\}$. Note that the ending markers also have their depints. When X at position p pops a letter into position p' then $\text{depint}(p') \leftarrow \text{depint}(p)$ (which is the position of this occurrence of X in the input equation). When we pop an a-prefix or b-suffix, then each letter in the popped prefix or suffix is assigned the depint of the variable that popped them (this can be equally seen as iterative popping and assigning this depint each time according to the above rule). Before we perform the (Γ_ℓ, Γ_r) pair compression then in parallel for each position p such that $UV[p] \in \Gamma_\ell$ we assign $\text{depint}(p) \leftarrow \text{depint}(p) \cup \text{depint}(p+1)$, here $p+1$ may be a position of a variable or of an endmarker, we say that we *update the depint* in this case. Then we perform a symmetric action for positions whose letters are in Γ_r (so for $p-1$). A simple argument, see Lemma 7, shows that the order of operation (Γ_ℓ or Γ_r first) does not matter and the new depint is obtained by taking a union with the appropriate neighbouring one. In particular, when we compress a pair then both compressed letters have the same depint, we assign the new position this depint, which is the union of depints of positions from before the update of depints. This is intuitively clear: if ab is compressed to c then this c depends on the union of intervals on which a and b depended. However, note that if we do not compress a letter from Γ_ℓ on position p then its depint should also be updated: it was not compressed because of the letter on position $p+1$.

Lemma 7. *The depints assigned before pair compression are the same, regardless of whether the Γ_ℓ or Γ_r letters are considered first.*
If $UV[p \mathinner{..} p+1] \in \Gamma_\ell\Gamma_r$ then right before the compression these two letters have the same depint.

Proof. It is enough to show that for three consecutive letters abc the depint of b is going to be the same, regardless of whether we consider Γ_ℓ or Γ_r first (note that the endmarkers are never compressed, so a letter $b \in \Gamma_\ell \cup \Gamma_r$ always has a symbol to the right and left). If $b \notin \Gamma_\ell \cup \Gamma_r$ then there is nothing to prove, as the depint remains the same; the case $b \in \Gamma_\ell$ and $b \in \Gamma_r$ are symmetric (note that it is always true that $\Gamma_\ell \cap \Gamma_r = \emptyset$), so we consider only the former.

Let the depints of b, c (formally: of their positions) be I, I'. If we consider Γ_ℓ first, then in the first step b gets the depint $I \cup I'$ and in the second step nothing changes for b. If we consider first Γ_r and $c \notin \Gamma_r$ then after the first step the depints of b, c are still I, I' and in the second step b gets depint $I \cup I'$. If $c \in \Gamma_r$ then in the first step it gets the depint $I \cup I'$ and b still has depint I. Then in the second step b gets depint $I \cup (I \cup I') = I \cup I'$. □

For Γ block compression, we perform in parallel the following operation for each block (perhaps of length 1) of a letter in Γ: given a block a^ℓ (for $\ell \geq 1$) at positions $p, p+1, \ldots, p+\ell-1$ we set the depints of those positions to $\bigcup_{i=-1}^{\ell} \text{depint}(p+i)$; note that $p-1$ and $p+\ell$ (so the positions directly to the left and right of the block a^ℓ) are included, and if the letters at their positions are in Γ then their depints are also updated at the same time. Observe that:

- when we compress a block then all letters within the compressed block have the same depint;
- we update the depint even if $\ell = 1$, in which case there is no compression.

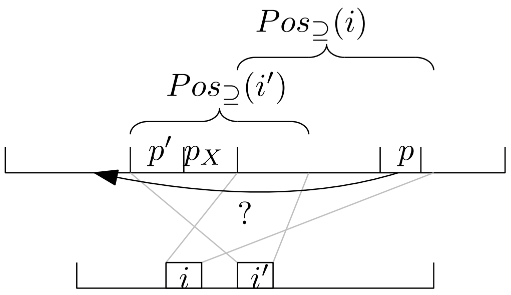

Fig. 1. Illustration to Claim 1, first case: p' is popped to the left of p_X. $\mathrm{Pos}_{\supseteq}(i)$ and $\mathrm{Pos}_{\supseteq}(i')$ are shown. The position p, which should be to the left of p_X and right of p' is shown.

The intuition for including the depints of letters to the right and left of the block is that the block depends on them as well, as they show where the block begins and ends.

For future reference, observe that by the way the depints are extended:

- the depints of the endmarkers are never changed;
- $\mathrm{Pos}_{\supseteq}(i)$ is always nonempty.

The first claim clearly follows from the definition, for the second, if i is an index of a letter, then initially it has exactly one position, for a given position its depint can only increase and if a position with depint that includes i is compressed, then the resulting letter also has i in its depint. If i is a position of a variable, then the variable can be removed only after popping the letter, but the popped letter will have i in its depint and the analysis as above applies.

The updates of depints before the block compression and pair compression may change depints of positions that were outside $\mathrm{Pos}_{\supseteq}(I)$ so that afterwards they are inside $\mathrm{Pos}_{\supseteq}(I)$. In such case, we say that $\mathrm{Pos}_{\supseteq}(I)$ *extends* to the neighbouring positions. Note that the same operation may extend $\mathrm{Pos}_{\supseteq}(I)$ and $\mathrm{Pos}_{\supseteq}(I')$ to the same position. In the following, we mostly focus on $\mathrm{Pos}_{\supseteq}(i)$.

Depints defined in this way indeed satisfy the conditions (I1–I3), this is shown in the below Lemma 8.

Lemma 8. *(I1–I3) hold during* LinWordEqSat.

Proof. We first show (I1) for $\mathrm{Pos}_{\supseteq}(i)$. The proof is by induction; this is true at the beginning, as $\mathrm{Pos}_{\supseteq}(i) = \{i\}$. When we update the depints, a position adjacent to a position in $\mathrm{Pos}_{\supseteq}(i)$ can become part of $\mathrm{Pos}_{\supseteq}(i)$ (this can be iterated when the depints are updated before the block compression), which is fine. During the compression, we compress symbols on positions with the same depints, and afterwards all those positions are replaced with one position with the same depint. So inside $\mathrm{Pos}_{\supseteq}(i)$ we replace a sequence of consecutive positions with a single one, so $\mathrm{Pos}_{\supseteq}(i)$ still remains an interval. When we pop a letter from variable at position p to position p', which then is neighbouring to p, then $\mathrm{depint}(p') = \mathrm{depint}(p)$ and so both p, p' are in $\mathrm{Pos}_{\supseteq}(i)$ and by inductive assumption $\mathrm{Pos}_{\supseteq}(i)$ was an interval, so either we insert a new position into it, so it is still an interval, or we create a new position to the left or right of it, so $\mathrm{Pos}_{\supseteq}(i)$ is still an interval. The same argument applies when we pop the whole a-prefix or suffix in Γ block compression. Lastly, when we remove a variable then we remove its position altogether, so an interval of positions remains an interval.

Now (I1) for $\mathrm{Pos}_{\supseteq}([i\,..\,j])$ for an arbitrary depint $[i\,..\,j]$ follows: $\mathrm{Pos}_{\supseteq}([i\,..\,j]) = \bigcap_{k=i}^{j} \mathrm{Pos}_{\supseteq}(k)$ and as each $\mathrm{Pos}_{\supseteq}(k)$ is an interval, also $\mathrm{Pos}_{\supseteq}([i\,..\,j])$ is an interval.

In order to show (I2) we show some technical claims.

Claim 1. *If $i \le i'$ then $Pos_{\supseteq}(i) \le Pos_{\supseteq}(i')$.*

We show Claim 1 by induction on the number of operations performed by the algorithm. Clearly this holds at the beginning, as then $\mathrm{Pos}_{\supseteq}(i) = \{i\}$ and $\mathrm{Pos}_{\supseteq}(i') = \{i'\}$. Consider the moment, in which the condition $\mathrm{Pos}_{\supseteq}(i) \le \mathrm{Pos}_{\supseteq}(i')$ is first violated, by symmetry it is enough to consider the case in which the first position in $\mathrm{Pos}_{\supseteq}(i')$, say p', is smaller than the first in $\mathrm{Pos}_{\supseteq}(i)$. If p' has been just popped, say from a variable at position p_X, then it cannot be popped to the right, as $\mathrm{depint}(p_X) = \mathrm{depint}(p')$ implies $p_X \in \mathrm{Pos}_{\supseteq}(i')$ and this is a contradiction with the choice of p': the left-most position in $\mathrm{Pos}_{\supseteq}(i')$. So p' was popped to the left from p_X, see Fig. 1. As already observed, $p_X \in \mathrm{Pos}_{\supseteq}(i')$ and before popping $\mathrm{Pos}_{\supseteq}(i) \le \mathrm{Pos}_{\supseteq}(i')$ held. So either $p_X \in \mathrm{Pos}_{\supseteq}(i)$ and then $\mathrm{depint}(p') = \mathrm{depint}(p_X)$ implies $p' \in \mathrm{Pos}_{\supseteq}(i)$, which contradicts the choice of p'; or there is $p \in \mathrm{Pos}_{\supseteq}(i)$ such that $p_X > p$. Then $p' > p \in \mathrm{Pos}_{\supseteq}(i)$, contradiction.

The other possibility is that the $\mathrm{depint}(p')$ was changed so that it got into $\mathrm{Pos}_{\supseteq}(i')$. In such a case $\mathrm{depint}(p')$ becomes a union of depints of some interval of positions $[p' - \ell \cdots p' + r]$ for some $\ell, r \ge 0$ and afterwards p' is in $\mathrm{Pos}_{\supseteq}(i')$ if and only if one of those positions $p' + o$ (here o can be negative, but $o \ne 0$) was in $\mathrm{Pos}_{\supseteq}(i')$ before taking this union. By induction

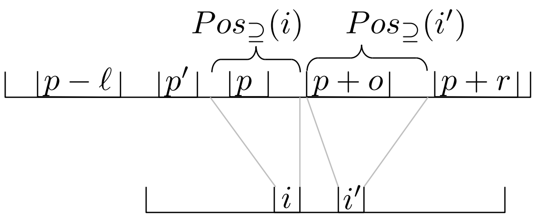

Fig. 2. Illustration to Claim 1, second case, letters at positions $p-\ell+1,\dots,p+k-1$ have their depints updated, as there is a block on those positions.

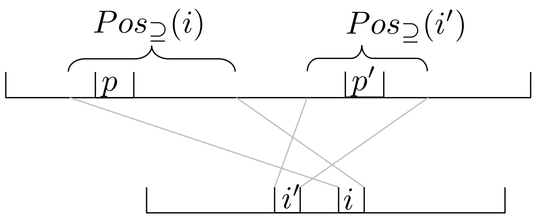

Fig. 3. Illustration to Claim 2.

assumption before the update of depints we had $\mathrm{Pos}_\supseteq(i) \le \mathrm{Pos}_\supseteq(i')$ and so there was $p \in \mathrm{Pos}_\supseteq(i)$ such that $p \le p'+o$. During the update, the depint(p) can only increase, so afterwards p is still in $\mathrm{Pos}_\supseteq(i)$. If $p \le p'$ then we are done, in particular, if $o < 0$ then we are done, so in the following we consider $o > 0$, see Fig. 2. Then $p' < p \le p'+o \le p+r$ and so depint(p') is increased as well by depint(p), so $p' \in \mathrm{Pos}_\supseteq(i)$. As p' is left-most in $\mathrm{Pos}_\supseteq(i')$, we have that left-most position in $\mathrm{Pos}_\supseteq(i)$ is less or equal to p'.

The last possibility is that a compression (pair compression or block compression) is applied. However, an interval of positions $p, p+1, \dots, p+k$ that have the same depint are replaced with one position with the same depint. This does not affect the relative order of $\mathrm{Pos}_\supseteq(i)$ and $\mathrm{Pos}_\supseteq(i')$: if one of the affected positions was in, say $\mathrm{Pos}_\supseteq(i)$, then all of them were in $\mathrm{Pos}_\supseteq(i)$. If they were in only one of $\mathrm{Pos}_\supseteq(i)$, $\mathrm{Pos}_\supseteq(i')$ then the replacement with one position does not affect $\mathrm{Pos}_\supseteq(i) \le \mathrm{Pos}_\supseteq(i')$, if they were in both then the replacement affects $\mathrm{Pos}_\supseteq(i)$, $\mathrm{Pos}_\supseteq(i')$ in the same way and if they were in neither then nothing changes. △

The second claim needed to show (I2) is

Claim 2.

$$p < p' \implies \mathrm{depint}(p) \le \mathrm{depint}(p')$$

Let $\mathrm{depint}(p) = [i\,..\,j]$ and $\mathrm{depint}(p') = [i'\,..\,j']$. If $\mathrm{depint}(p) \le \mathrm{depint}(p')$ does not hold, then either $i > i'$ or $j > j'$. We consider the former, the proof for the latter is symmetric. Hence suppose that $i > i'$, see Fig. 3. In particular, $i' \notin \mathrm{depint}(p)$ and so from the definition $p \notin \mathrm{Pos}_\supseteq(i')$. Consider

$$\mathrm{Pos}_\supseteq(i') \not\ni p \le p' \in \mathrm{Pos}_\supseteq(i') \ .$$

By (I1) for $\mathrm{Pos}_\supseteq$, the $\mathrm{Pos}_\supseteq(i')$ is an interval. Hence we get that p is smaller than all positions in $\mathrm{Pos}_\supseteq(i')$. However, by Claim 1 the $i > i'$ implies $\mathrm{Pos}_\supseteq(i) \ge \mathrm{Pos}_\supseteq(i')$ and by assumption $p \in \mathrm{Pos}_\supseteq(i)$, which means that it cannot be smaller than each position in $\mathrm{Pos}_\supseteq(i')$, contradiction (Fig. 3). △

Now (I2) follows: let $I \ne I'$ be such that $\mathrm{Pos}_=(I) \ne \emptyset \ne \mathrm{Pos}_=(I')$. Let $p \in \mathrm{Pos}_=(I)$ and $p' \in \mathrm{Pos}_=(I')$. As $p \in \mathrm{Pos}_=(I)$ if and only if $\mathrm{depint}(i) = I$ and similarly for p' and I' we get that $p \ne p'$, by symmetry let $p < p'$. Then from Claim 2 we get $I = \mathrm{depint}(p) \le \mathrm{depint}(p') = I'$, as desired.

For the purpose of the proof of (I3), define $\mathrm{Pos}_\subseteq(I) = \{p \mid \mathrm{depint}(p) \subseteq I\}$ (a dual notion to $\mathrm{Pos}_\supseteq(I)$). Moreover, given two similar depints $I \sim I'$ we say that they are *translated by* k when $I = [i\,..\,j]$ and $I' = [i+k\,..\,j+k]$, this is denoted as $I' = I + k$.

Claim 3. *$Pos_\subseteq(I)$ is an interval of positions. Given two similar depints $I \sim I'$ it holds that $UV[Pos_\subseteq(I)] = UV[Pos_\subseteq(I')]$. Let m be such that $I' = I + m$. Then every position in $Pos_\subseteq(I')$ has the depint of the corresponding position in $Pos_\subseteq(I)$ translated by m.*

Before we show Claim 3, observe that (I3) follows from Claim 3: clearly $\mathrm{Pos}_=(I) \subseteq \mathrm{Pos}_\subseteq(I)$ (and $\mathrm{Pos}_=(I') \subseteq \mathrm{Pos}_\subseteq(I')$), by Claim 3 we have $UV[\mathrm{Pos}_\subseteq(I)] = UV[\mathrm{Pos}_\subseteq(I')]$ and the corresponding positions in them have their depints translated by the same m. The $\mathrm{Pos}_=(I)$ are exactly the positions with depint equal to I and so the corresponding positions have depint

$I + m = I'$. Moreover, each other position in position $\text{Pos}_{\subseteq}(I)$ and $\text{Pos}_{\subseteq}(I')$ has strictly smaller depints, so they are not in $\text{Pos}_{=}(I)$ nor in $\text{Pos}_{=}(I')$, which shows that $UV[\text{Pos}_{=}(I)] = UV[\text{Pos}_{=}(I')]$.

Claim 3 also implies (I1) for $\text{Pos}_{=}(I)$, as $\text{Pos}_{=}(I) = \text{Pos}_{\supseteq}(I) \cap \text{Pos}_{\subseteq}(I)$ and both are intervals, so also $\text{Pos}_{=}(I)$ is an interval.

It remains to show Claim 3, we do it by induction on the number of operations performed by the algorithm. At the beginning we have $\text{Pos}_{\subseteq}(I) = \text{Pos}_{=}(I) = I$ and similarly $\text{Pos}_{\subseteq}(I') = \text{Pos}_{=}(I') = I'$. Then $I \sim I'$ by definition means that $U_0V_0[I] = U_0V_0[I']$ and as at the beginning $UV = U_0V_0$, we get $UV[I] = UV[I']$. Let m be such that $I + m = I'$. Then clearly if index i has $\text{depint}(i) = \{i\}$ then index $i + m$ has $\text{depint}(i + m) = \{i + m\}$.

Moving to the proof, if a variable X at position p_X pops a letter to position p, then $\text{depint}(p) = \text{depint}(p_X)$. If $p_X \notin \text{Pos}_{\subseteq}(I)$ then also $p \notin \text{Pos}_{\subseteq}(I)$. If $p_X \in \text{Pos}_{\subseteq}(I)$ then also $p \in \text{Pos}_{\subseteq}(I)$ and as p_X, p are next to each other, then $\text{Pos}_{\subseteq}(I)$ is still an interval. Let $I' \sim I$ and $I' = I + m$. Then by inductive assumption at the corresponding position $p'_X \in \text{Pos}_{\subseteq}I'$ there is X and by the algorithm it pops the same letters (to the same side). Those positions are at corresponding places, have the same letter, so $UV[\text{Pos}_{\subseteq}(I)] = UV[\text{Pos}_{\subseteq}(I')]$ still holds. Also, those positions have depints $\text{depint}(p_X)$ and $\text{depint}(p'_X) = \text{depint}(p_X) + m$ by inductive assumption, so they are still translated by the same m.

The other possible operation is the removal of the variable, say X, from the equation. If X is within $\text{Pos}_{\subseteq}(I)$ then after the removal of all occurrences of X the $\text{Pos}_{\subseteq}(I)$ is still an interval (we are removing the whole positions, not positions from $\text{Pos}_{\subseteq}(I)$). Moreover, when $I \sim I'$ then by the inductive assumption $UV[\text{Pos}_{\subseteq}(I)] = UV[\text{Pos}_{\subseteq}(I')]$ and so X occurs at the corresponding position of $UV[\text{Pos}_{\subseteq}(I)]$ and $UV[\text{Pos}_{\subseteq}(I')]$, so after the removal still $UV[\text{Pos}_{\subseteq}(I)] = UV[\text{Pos}_{\subseteq}(I')]$. Lastly, the depints of remaining positions do not change, in particular, the corresponding positions in $UV[\text{Pos}_{\subseteq}(I)]$ and $UV[\text{Pos}_{\subseteq}(I')]$ have the depints translated by m.

Another operation is updating the depint right before the pair compression, say $\text{depint}(p) \leftarrow \text{depint}(p) \cup \text{depint}(p^\bullet)$ where $p^\bullet = p + 1$ or $p^\bullet = p - 1$. Then p ceases to be in $\text{Pos}_{\subseteq}(I)$ if and only if $p^\bullet \notin \text{Pos}_{\subseteq}(I)$ and so $\text{Pos}_{\subseteq}(I)$ loses its first or last position, so it is still an interval. Observe that we perform exactly the same operation on the corresponding position p' and it is removed from $\text{Pos}_{\subseteq}(I')$ in exactly the same case, when p is removed from $\text{Pos}_{\subseteq}(I)$, as by the inductive assumption they have the same letter. In particular: if one is first/last position in $\text{Pos}_{\subseteq}(i)$ or $\text{Pos}_{\subseteq}(i')$, then the other one is on the first/last position of the other set.

If $p^\bullet \in \text{Pos}_{\subseteq}(I)$ then by induction assumption the corresponding position $p'^\bullet \in \text{Pos}_{\subseteq}(i')$, $\text{depint}(p') = \text{depint}(p) + m$ and $\text{depint}(p'^\bullet) = \text{depint}(p^\bullet) + m$, hence also $\text{depint}(p') \cup \text{depint}(p'^\bullet) = (\text{depint}(p) \cup \text{depint}(p^\bullet)) + m$, as desired.

The last operation is updating the depints right before the block compression. Consider a position $p \in \text{Pos}_{\subseteq}(I)$ and a block at positions $[p - k \, .. \, p + \ell]$, then each of those positions gets the same depint: the union $\bigcup_{i=-k-1}^{\ell+1} \text{depint}(p + i)$. If each position $p - k - 1, \dots, p + \ell + 1$ was in $\text{Pos}_{\subseteq}(I)$ then afterwards each $p - k, \dots, p + \ell$ is still in $\text{Pos}_{\subseteq}(I)$ and if at least one of $p - k - 1, \dots, p + \ell + 1$ was outside $\text{Pos}_{\subseteq}(I)$ then afterwards all $p - k, \dots, p + \ell$ are outside $\text{Pos}_{\subseteq}(I)$. In particular, $\text{Pos}_{\subseteq}(I)$ is shortened (on the left) by exactly the positions corresponding to the a-prefix, where a is the first letter of $UV[\text{Pos}_{\subseteq}(I)]$, or nothing, when the first symbol of $UV[\text{Pos}_{\subseteq}(I)]$ is an endmarker or a variable; and similarly it is shortened on the right by the b-suffix, where b is the last letter of $UV[\text{Pos}_{\subseteq}(I)]$, or nothing, when the last symbol of $UV[\text{Pos}_{\subseteq}(I)]$ is an endmarker or a variable.

Consider the similar depints $I, I' = I + m$ and let p' be the position corresponding to p in I'. Observe that by the induction assumption p is a part of the a-prefix or b-suffix of $UV[\text{Pos}_{\subseteq}(I)]$ if and only if p' is a part of the a-prefix or b-suffix of $UV[\text{Pos}_{\subseteq}(I')]$. Hence, p is removed from $\text{Pos}_{\subseteq}(I)$ if and only if p' is removed from $\text{Pos}_{\subseteq}(I')$. Moreover, when it is not removed, then the two blocks, in which p and p' are, span by the same number of letters to the left and to the right, as they are both within $UV[\text{Pos}_{\subseteq}(I)]$ and $UV[\text{Pos}_{\subseteq}(I')]$, respectively, and by inductive assumption the corresponding positions have the same letters. In particular, $UV[\text{Pos}_{\subseteq}(I)] = UV[\text{Pos}_{\subseteq}(I')]$.

It is left to compare the new depints of p and p', assuming that they are still in $\text{Pos}_{\subseteq}(I)$ and $\text{Pos}_{\subseteq}(I')$, respectively. The new depint of p is equal to $\bigcup_{i=-k-1}^{\ell+1} \text{depint}(p + i)$ while the new depint of p' is $\bigcup_{i=-k-1}^{\ell+1} \text{depint}(p' + i)$. As by induction assumption we have $\text{depint}(p' + i) = \text{depint}(p + i) + m$ for each $i \in \{-k - 1, \dots, \ell + 1\}$ (here we use the fact that the positions p, p' are not within the a-prefix nor the b-suffix of $UV[\text{Pos}_{\subseteq}(I)]$ and $UV[\text{Pos}_{\subseteq}(I')]$, respectively). Thus also the new depints are translated by m. This ends the proof of Claim 3 and of the whole Lemma. □

4.2. Pair compression strategy

Recall that by Lemma 2 all subprocedures of LinWordEqSat are sound, so it will not accept an unsatisfiable word equations. At the same time, by Lemma 1 in order to show that the word equations are in non-deterministic linear space it is enough to show a linear space bound for *some* non-deterministic choices of the algorithm. Thus in the following we assume that LinWordEqSat always makes the non-deterministic choices according to the solution, see comment after Lemma 2, which introduces this notion. For such choices the space consumption of a particular non-deterministic execution depends only on the choices of the partitions during pair compression, called in the following a *strategy*. We describe a strategy leading to a linear-space usage, which will be enough to show Theorem 1.

Idea. Imagine we ensured that during one phase each variable popped $\mathcal{O}(1)$ letters and each $\text{Pos}_{\supseteq}(i)$ extended by $\mathcal{O}(1)$ letters. Then $|\text{Pos}_{\supseteq}(i)| = \mathcal{O}(1)$: we introduced $\mathcal{O}(1)$ positions to $\text{Pos}_{=}(i)$, say at most k, and by Lemma 3 among positions in $\text{Pos}_{\supseteq}(i)$ at the beginning of the phase at least 2/3 took part in compression, so their number dropped by 1/3; thus

$|\mathrm{Pos}_{\supseteq}(i)| \leq 3k$. As a result, $|\mathrm{Pos}_{=}(I)| \leq 3k$ for each depint I: as $\mathrm{Pos}_{=}(I) \subseteq \mathrm{Pos}_{\supseteq}(i)$ for each $i \in I$. This would yield that the whole space used for the encoding is linear: recall that each letter is encoded as $U_0V_0[I]\#m$ for some depint I and number m. As observed, each number m used in $U_0V_0[I]\#m$ is at most $3k = \mathcal{O}(1)$, so for each $U_0V_0[I]\#m$ the encoding of m uses at most constant factor larger amount of additional space than the encoding of $U_0V_0[I]$. As each $U_0V_0[I]$ is used $|\mathrm{Pos}_{=}(I)|$ times in the encoding, it is enough to estimate

$$\sum_{I:\text{depint}} \|U_0V_0[I]\| \cdot |\mathrm{Pos}_{=}(I)| \ .$$

Substituting the definition and some simple calculations yields, see the proof of Lemma 11, that

$$\sum_{I:\text{depint}} \|U_0V_0[I]\| \cdot |\mathrm{Pos}_{=}(I)| = \sum_{i:\text{index}} \|U_0V_0[i]\| \cdot |\mathrm{Pos}_{\supseteq}(i)|$$

and the right-hand side is at most linear in terms of the input equation: $|\mathrm{Pos}_{\supseteq}(i)| = \mathcal{O}(1)$ and $\sum_{i:\text{index}} \|U_0V_0[i]\|$ is the encoding size of the input equation.

Unfortunately, we cannot ensure that each variable pops $\mathcal{O}(1)$ letters nor that each $\mathrm{Pos}_{\supseteq}(i)$ extends by $\mathcal{O}(1)$ positions. We can make this true *in expectation*: Given a phase, we call a letter *new*, if it was introduced during this phase. New letters cannot be popped nor can $\mathrm{Pos}_{\supseteq}(i)$ be extended by positions with new letters. Thus they are used to prevent extending $\mathrm{Pos}_{\supseteq}(i)$ and prevent popping from variables: it is enough to ensure that the first/last letter of a variable is new and that a letter on the position to the left/right of $\mathrm{Pos}_{\supseteq}(I)$ is new. Now, given a random partition there is a $1/4$ probability that a fixed pair is compressed (and the resulting letter is new), which in a sense means that in expectation a new letter will appear within $\mathcal{O}(1)$ letters from each end of $S(X)$ and within $\mathcal{O}(1)$ letters to the left and right of $\mathrm{Pos}_{\supseteq}(I)$. It remains to formalise this approach and show that the expectation translates to only $\mathcal{O}(1)$ times worse worst-case performance.

Strategy Given a solution S of an equation, we say that a variable X is *left blocked* if

- $S(X)$ has at most one letter or was removed from the equation *or*
- the first or second letter in $S(X)$ is new,

otherwise a variable is *left unblocked*; define *right blocked* and *right unblocked* variables similarly. The idea is that the unblocked variables are the problematic ones, therefore by convention the removed variables are blocked.

Let i be an index and let p_i be the left-most position in $\mathrm{Pos}_{\supseteq}(i)$ (recall that $\mathrm{Pos}_{\supseteq}(i)$ is always non-empty), consider the corresponding position $S(p_i)$ in $S(U)$ or $S(V)$ (if p_i is a position of a variable, then consider the leftmost-position from the corresponding substitution for the variable; note that this happens only when this variable has not popped anything to the left). Then i is *left blocked* if on $S(p_i) - 1$ or $S(p_i) - 2$ there is a new letter or the left endmarker (note that saying that "there is an endmarker" on some position is just a unified way of treating new letters and equation's end/beginning). Otherwise, i is *left unblocked*; define *right blocked* and *right unblocked* indices similarly. Note that an index can be blocked from left and unblocked from right or blocked from both sides, etc. Note that the positions $S(p_i) - 1$ and $S(p_i) - 2$ may be inside a substitution for a variable, even if p_i is a position of a letter.

Lemma 9. *Consider a solution $S = S_0$ and consecutive solutions $S_1, S_2, \ldots$ corresponding to it during a phase.*

If a variable X becomes left blocked for some S_k, then it is left blocked for each S_ℓ for $\ell \geq k$ and it pops to the left at most 1 *letter after it became left blocked.*

If an index i becomes left blocked for some S_k then it is left blocked for each S_ℓ for $\ell \geq k$ and $\mathrm{Pos}_{\supseteq}(i)$ extends to at most one letter to the left after i became left blocked.

For both claims the symmetric statements with "left" replaced with "right", hold.

Proof. First observe that by convention the removed variable is blocked and the removed variables trivially satisfy all the claims of the lemma.

Recall that the notion that $S_1, S_2, \ldots$ correspond to S_0 means that each consecutive $S_{k+1}(X)$ is obtained from $S_k(X)$ by changes appropriate to the compression operation: for Γ compression — by removing the a-prefix and b-suffix of $S_k(X)$ followed by Γ-block compression; for Γ_ℓ, Γ_r pair compression — by removing the first letter, when it is in Γ_r, and last, when it is in Γ_ℓ, of $S_k(X)$, followed by the Γ_ℓ, Γ_r pair compression.

If X becomes left blocked because $S_k(X)$ has one letter, then it will stay left blocked and can pop at most one letter further on.

If it becomes left blocked because first or second letter of $S_k(X)$ is new then this new letter cannot be popped, as we pop only letters from Γ, so this letter will remain on first or second position within $S_\ell(X)$ for $\ell \geq k$ (so in this phase) and so X remains left blocked. In particular, if this letter is first (second) in $S_k(X)$, then X cannot pop left a letter (can pop at most one letter); a similar argument applies on the right side.

Similarly, $\mathrm{Pos}_{\supseteq}(i)$ can extend only to positions with letters from Γ (this does not include endmarkers), let p_i be the leftmost position in $\mathrm{Pos}_{\supseteq}(i)$, consider the corresponding position $S(p_i)$ (if p_i is a position of a letter; if p_i is a position of a variable: the left-most position in the substitution for the variable at position p_i) If a letter at position $S(p_i)-2$ or $S(p_i)-1$ is new or is an endmarker, then $\mathrm{Pos}_{\supseteq}(i)$ cannot extend to this position and so $\mathrm{Pos}_{\supseteq}(i)$ will extend by at most one position and will remain left-blocked. A symmetric argument applies for right blocked depints. $\square$

The idea of the strategy is as follows: we consider the increase of size of encoding of depints in the equation. We consider separately the increases in sizes of:

1. encoding of depints (without the following numbers) of letters popped from variables;
2. encodings of depints (without the following numbers) due to updates of depints (which is viewed as extending $\mathrm{Pos}_{\supseteq}(i)$ to new positions);
3. encoding of numbers in the encoding of letters for letters popped from variables
4. encoding of numbers in the encoding of letters after the extension of depints.

Note that when the left-most position in $\mathrm{Pos}_{\supseteq}(i)$ is a variable X, then letters popped (to the left) from X get into $\mathrm{Pos}_{\supseteq}(i)$ and $\mathrm{Pos}_{\supseteq}(i)$ can extend to the left, to the letters that were to the left of the corresponding $S(X)$. Similarly, even when i is blocked, still letters with depint equal to i can be popped. Thus we consider separately extending of $\mathrm{Pos}_{\supseteq}(i)$ and the popped letters.

We can upper bound each of these increases as follows (a formal proof is given later on), with i-th sum upper-bounding the i-th increase above.

$$\sum_{X\in\mathcal{X}\text{: left unblocked}} n_X\cdot\|X\| + \sum_{X\in\mathcal{X}\text{: right unblocked}} n_X\cdot\|X\| \tag{1}$$

$$\sum_{i\text{: left unblocked index}} \|U_0V_0[i]\| + \sum_{i\text{: right unblocked index}} \|U_0V_0[i]\| \tag{2}$$

$$\sum_{X\in\mathcal{X}\text{: left unblocked}} n_X + \sum_{X\in\mathcal{X}\text{: right unblocked}} n_X \tag{3}$$

$$\sum_{i\text{: left unblocked index}} 1 + \sum_{i\text{: right unblocked index}} 1 \tag{4}$$

In all cases $\|\alpha\|$, where α is a variable or a letter in the original equation $U_0=V_0$, denotes the encoding size of α in the input equation. If all letters and variables in the input equation are encoded using natural encoding, as bit sequences of the same length, then (1) is equivalent to (3) and (2) to (4), in both cases with a multiplier $\log\lceil|\Gamma|\rceil$.

The strategy iterates steps (1)–(4). In a step i it chooses a partition so that the corresponding i-th sum decreases by a maximum amount (we show that this is at least half), unless this sum is already 0. Thus, the upper-bound on each of the size-increase halves every 4 choice of partition, so in total the sum of all increases for all partition choices (in one phase) is $\mathcal{O}(1)$ times the sum of values of (1)–(4) before any pair compression is performed. By easy calculation, this is $\mathcal{O}(1)$ times the size of the encoding of the input equation.

Note that the sum in (1) has a natural interpretation: it upper-bounds the number of letters that are popped from unblocked variables, taking into account the size of their encoding; by Lemma 9 for one variable it can happen only once that it pops a letter to the left (right) after becoming left (right, respectively) blocked, so we can estimate this separately. The sum in (2) has a similar interpretation: it is the maximal amount of position to which unblocked $\mathrm{Pos}_{\supseteq}(i)$ can extend, again taking into account the size of their encoding; in terms of depints: the maximal number of positions whose encoding will start using $U_0V_0[i]$; again, Lemma 9 shows that $\mathrm{Pos}_{\supseteq}(i)$ can extend at most once to the left (right, respectively) after i becomes left (right, respectively) blocked. The sums in (3) and (4) are connected to the appropriate values in a more complex way, and they upper-bound them, which is shown later on.

Lemma 10. *During the pair compression* LinWordEqSat *can always choose a partition that at least halves the value of a chosen non-zero sum among* (1)–(4)*, the other sums then do not increase.*

Proof. First observe that for any partition all sums in (1)–(4) do not increase in a phase: all those sums depend only on whether a variable (or index) is left/right blocked, and by Lemma 9 if a variable (an index) is left/right blocked, then it will remain left/right blocked in this phase.

Consider (1) and take a random partition, in the sense that each letter $a\in\Gamma$ goes to the Γ_ℓ with probability 1/2 and to Γ_r with probability 1/2. Let us fix a variable X and its side, say left. Consider, what happens with $n_X\cdot\|X\|$ in the sum from (1) corresponding to left unblocked variables? If X is left blocked then, by Lemma 9, it will stay left blocked and so the contribution is and will be 0. If it is left unblocked, then its two first letters a, b are not new, so they are in Γ. If $S(X)$ has only those two letters or the third leftmost letter is new, then with probability 1/2 the a will be in Γ_r and it will be popped and X will become left blocked (as $S(X)$ has only one letter or there is new letter at the first or the second position in $S(X)$). The remaining case is that the three leftmost letters in $S(X)$ are not new, let them be $a, b, c\in\Gamma$. By Lemma 4 we have $a \neq b \neq c$, as there are no blocks of letters from Γ in $S(U)$ and $a, b, c \in \Gamma$. With probability 1/4 $ab \in \Gamma_\ell \Gamma_r$ and with probability 1/4 $bc \in \Gamma_\ell \Gamma_r$. Those events are disjoint (as in one $b \in \Gamma_r$ and in the other $b \in \Gamma_\ell$) and so their union happens with probability 1/2. In both cases X will become left blocked, as a new letter is its first or second in $S(X)$. In all uninvestigated cases the contribution of $n_X \cdot \|X\|$ cannot raise. Also, the analysis for the sum of right-unblocked X is symmetrical. This shows the claim in this case.

The case of (3) is shown in the same way as (1).

For the sum in (2), the analysis for an index i that is left unblocked (in $S(U) = S(V)$) is similar, but this time we consider the positions to the left of $\mathrm{Pos}_\supseteq(i)$ and $\mathrm{Pos}_\supseteq(i)$ can extend to them (instead of letters being popped from variables in case of (1)) and some of them may be compressed to one. The only subtle difference is that only letters in the equation can be in $\mathrm{Pos}_\supseteq(i)$, so if to the left of $\mathrm{Pos}_\supseteq(i)$ there is a variable, we have to consider $S(\mathrm{Pos}_\supseteq(i))$.

Moving to the details, let p_i be the left-most position in $\mathrm{Pos}_\supseteq(i)$, consider $S(p_i)$, when p_i is a position of a letter, if p_i is a position of a variable then consider the left-most position in the substitution for a variable at position p_i. As in the case above, if i is left-blocked then it will remain left-blocked, and the corresponding sum in (2) is 0. Hence we are done with the case when there is a new letter or an endmarker at position $S(p_i) - 1$ or $S(p_i) - 2$. If there is a new letter or endmarker at the position $S(p_i) - 3$ then consider the position $S(p_i) - 1$. With probability 1/2 the letter at position $S(p_i) - 1$ is assigned to Γ_ℓ, we consider this case. If $S(p_i) - 1$ is a position within a substitution for a variable, then its letter is popped. Afterwards, regardless of whether $S(p_i) - 1$ was a position of a variable or not, the depint of the corresponding position (in the equation) $p_i - 1$ is updated: $\mathrm{depint}(p_i - 1) \leftarrow \mathrm{depint}(p_i - 1) \cup \mathrm{depint}(p_i)$ and so $p_i - 1$ is now in $\mathrm{Pos}_\supseteq(i)$. Hence i is left-blocked, as there is a new letter or endmarker two position to the left from $\mathrm{Pos}_\supseteq(i)$; in other word: i becomes left-blocked with probability at least 1/2. The last case is when at positions $S(p_i) - 3, S(p_i) - 2, S(p_i) - 1$ there are letters from Γ. Then as in the case above we can show that with probability at least 1/2 one of pairs at position $(S(p_i) - 3, S(p_i) - 2)$ or $(S(p_i) - 2, S(p_i) - 1)$ is compressed and so i becomes left-blocked.

The case of (4) is shown in the same way as (2). □

Space consumption We now give a linear space bound on the size of encoding of equation. This formalises the intuition from the beginning of Section 4.2. As a first step, we show some upper-bound. Define:

$$\begin{aligned} H_d(U, V) &= \sum_{i:\text{index}} \|U_0 V_0[i]\| \cdot |\mathrm{Pos}_\supseteq(i)| \\ H_n(U, V) &= 3 \sum_{i:\text{index}} |\mathrm{Pos}_\supseteq(i)| \cdot \log(|\mathrm{Pos}_\supseteq(i)| + 1) \\ H(U, V) &= H_d(U, V) + H_n(U, V) \ . \end{aligned}$$

H_d corresponds to encoding size of depints and H_n: to the numbers (following depints) in the encoding.

Lemma 11. *Given the equation (U, V) it holds that $\|(U, V)\| \le 3\|(U_0, V_0)\|_0 + 3H(U, V)$, where $\|(U_0, V_0)\|_0$ is the size of the encoding of the input equation and $\|(U, V)\|$ is the encoding size using the coding described in Section 4.1.*

Proof. Recall that the idea is that we encode the variables "as in the input equation" and a letter at position p as "$\|U_0 V_0[\mathrm{depint}(p)]\|_0 \# q$", where p is the q-th position in $\mathrm{Pos}_=(\mathrm{depint}(p))$. However, we have to ensure that the codes for different symbols are different: in principle the "codes" above could be the same for a letter and a variable. This is obtained by standard methods and described in detail in the following.

The variables are encoded as in the input equation, with each bit prefixed with `00`, i.e. with bit `0` represented as `000` and bit `1` as `001`. Thus variables use at most $3\|(U_0, V_0)\|_0$ bits and so it is enough to show that our encoding uses at most $3H(U, V)$ for letters.

A letter at position p that is a q-th position in $\mathrm{Pos}_=(\mathrm{depint}(p))$ is encoded as $\|U_0 V_0[\mathrm{depint}(p)]\|_0$ in which each bit is prefixed with `01`, i.e. `0` is represented as `010` and `1` as `011`; followed by the bit representation of number q, in which the bits are prefixed with `10`. Note that we do not encode #, which was needed only as a separator. Such code is uniquely decodable: we read the bits in triples, the first two bits determine, what type of symbol we encode and in particular we can establish the beginnings and ends of codes of symbols. Moreover, two different symbols are assigned different codes: variables and symbols have different codes, codes for different variables are different (as they were different in the input equation) and different letters are given different codes: if $\|U_0 V_0[\mathrm{depint}(p)]\|_0 \# q$ and $\|U_0 V_0[\mathrm{depint}(p')]\|_0 \# q'$ are encoded in the same way then $\mathrm{depint}(p) = \mathrm{depint}(p')$ and $q = q'$ hence by (I3) the encoded letter is the same.

It remains to estimate the space used for encoding the letters. We first do it for the space used by $\|U_0 V_0[\mathrm{depint}(p)]\|_0$ (without prefixing of the bits) and then the one used by q.

Let $\mathrm{depint}(p) = [k\,..\,\ell]$, then $\|U_0 V_0[\mathrm{depint}(p)]\|_0 = \sum_{i=k}^{\ell} \|U_0 V_0[i]\|_0$ and the space usage is obtained by taking a sum over all positions (with letters) in the equation (U, V). When we change the order of grouping and first group by $U_0 V_0[i]$, then summed over $i = 1 \ldots |U_0 V_0|$ we obtain $\sum_{i:\text{ index}} \|U_0 V_0[i]\| \cdot |\mathrm{Pos}_\supseteq(k)| = H_d(U, V)$. In numbers:

$$\begin{aligned}
\sum_{p\text{: position in } U=V} \|U_0V_0[\text{depint}(p)]\|_0 &= \sum_{p\text{: position in } U=V}\ \sum_{i\in\text{depint}(p)} \|U_0V_0[i]\|_0\\
&= \sum_{\substack{(p,i):p \text{ is a position in } U=V,\\ i\in\text{depint}(p)}} \|U_0V_0[i]\|_0\\
&= \sum_{(p,i):p\in\text{Pos}_{\supseteq}(i)} \|U_0V_0[i]\|_0\\
&= \sum_{i\text{: index}} \|U_0V_0[i]\|_0 \cdot |\text{Pos}_{\supseteq}(i)|\\
&= H_d(U,V)\ .
\end{aligned}$$

Taking into the account the prefixing of the buts yields the desired $3H_d(U,V)$.

Let us now move to the space usage of numbers in the encoding (so the ones following the depints). Given a depint I each letter in $\text{Pos}_=(I)$ is assigned a number from 1 to $|\text{Pos}_=(I)|$, which is encoded on $\lceil\log(|\text{Pos}_=(I)|+1)\rceil$ bits. So a number for the position p uses $\lceil\log(|\text{Pos}_=(\text{depint}(p))|+1)\rceil$ bits and so the space usage for all positions is:

$$\sum_{p\text{: position}} \lceil\log(|\text{Pos}_=(\text{depint}(p))|+1)\rceil = \sum_{I\text{: depint}} |\text{Pos}_=(I)|\lceil\log(|\text{Pos}_=(I)|+1)\rceil\ .$$

For each depint I we choose an index $i_I \in I$ such that one i is chosen at most twice over all depints, this is done as follows: We know that depints are linearly ordered by $\leq$, see (I2). Fix two consecutive depints in this order $I \leq I'$, let $I=[i\,..\,j]$, $I'=[i'\,..\,j']$. If $i<i'$ then we choose $i_I=i$ and if $i=i'$ then we choose $i_I=j$, so i_I is one of two ends of I. If I' is the last depint, then if $i=i'$ then we choose $i_{I'}=j'$ and otherwise $i_{I'}=i'$.

Suppose that some fixed i is chosen for two depints $I<I'$ as the beginning. But this cannot be, as by the choice of $i=i_I$ we have that all following depints do not include i. So suppose that i was chosen twice for $I<I'$ as the end. But this cannot be, as by the choice of $i=i_I$ we have that all following depints include $i+1$ or some larger index.

As $i_I\in I$ we have $\text{Pos}_{\supseteq}(i_I)\supseteq\text{Pos}_=(I)$ and so $|\text{Pos}_{\supseteq}(i_I)|\geq|\text{Pos}_=(I)|$. Hence

$$\begin{aligned}
\sum_{I\text{: depint}} |\text{Pos}_=(I)|\lceil\log(|\text{Pos}_=(I)|+1)\rceil &\leq \sum_{I\text{: depint}} |\text{Pos}_{\supseteq}(i_I)|\lceil\log(|\text{Pos}_{\supseteq}(i_I)|+1)\rceil\\
&\leq 2\sum_{i\text{: index}} |\text{Pos}_{\supseteq}(i)|\lceil\log(|\text{Pos}_{\supseteq}(i)|+1)\rceil
\end{aligned}$$

It can be verified by simple calculation that $\lceil\log(x+1)\rceil \leq \frac{3}{2}\log(x+1)$ for natural x and so

$$\begin{aligned}
\sum_{I\text{: depint}} |\text{Pos}_=(I)|\lceil\log(|\text{Pos}_=(I)|+1)\rceil &\leq 3\sum_{i\text{: index}} |\text{Pos}_{\supseteq}(i)|\log(|\text{Pos}_{\supseteq}(i)|+1)\\
&= H_n(U,V)\ ,
\end{aligned}$$

again, taking into the account the prefixing of the bits yields the desired $3H_n(U,V)$. □

Instead of showing a linear bound on $\|(U,V)\|$ we give a linear bound on $H(U,V)$. Recall that (U_0,V_0) denotes the input equation.

Lemma 12. *Consider an equation $U=V$, its solution S, a phase of* LinWordEqSat *which makes the non-deterministic choices according to S and partitions according to the strategy. Let the returned equation be (U',V'). Then*

$$H(U',V') \leq \frac{5}{6}H(U,V)+\alpha\|(U_0,V_0)\|$$

and for every intermediate equation (U'',V'') we have

$$H(U'',V'') \leq \beta H(U,V)+\gamma\|(U_0,V_0)\|$$

for some constants α,β,γ.

Proof. We separately estimate the H_d and H_n. Recall that

$$H_d(U,V) = \sum_{i\text{:index}} \|U_0V_0[i]\| \cdot |\text{Pos}_{\supseteq}(i)|$$

Clearly for a fixed index i the $\|U_0V_0[i]\|$ does not change, what changes is $|\mathrm{Pos}_{\supseteq}(i)|$, i.e. the number of positions that have i in their depint. This can change in the following way:

Hd1 When i is a position of a variable, i.e. $U_0V_0[i]$ is a variable, then this variable can pop letters, which will have depint equal to $\{i\}$.
Hd2 Due to updates of depints new positions get to $\mathrm{Pos}_{\supseteq}(i)$, i.e. they now have $\{i\}$ in their depints.
Hd3 $|\mathrm{Pos}_{\supseteq}(i)|$ can decrease due to compression: when we compress two (or more) letters whose positions are in $\mathrm{Pos}_{\supseteq}(i)$, we replace them by one and so the size of $\mathrm{Pos}_{\supseteq}(i)$ decreases.
Hd4 When i is a position of a variable, $\mathrm{Pos}_{\supseteq}(i)$ can decrease when this variable is removed. We will disregard this case: decrease of $\mathrm{Pos}_{\supseteq}(i)$ cannot increase the encoding size.

We upper-bound the values corresponding to (Hd1)–(Hd2) and lower bound the one corresponding to (Hd3). In other words, if $U' = V'$ is an equation after the phase, we use as an estimation:

$$\begin{aligned}H_d(U',V') \le H_d(U,V) &+ \underbrace{\sum_{X\in\mathcal{X}} \|X\| \cdot |\{p \mid p \textit{ is popped by a variable } X\}|}_{(\text{Hd1})}\\ &+ \underbrace{\sum_{i:\,\text{index}} \|U_0V_0[i]\| \cdot |\{p \mid p \textit{ has its depint updated by } i\}|}_{(\text{Hd2})}\\ &- \underbrace{\sum_{i:\,\text{index}} \|U_0V_0[i]\| \cdot |\{p \mid i \in \mathrm{depint}(p),\ p \textit{ was compressed}\}|}_{(\text{Hd3})}\ .\end{aligned}$$

Note that in the last case we need to take into the account that some positions are created after the compression, i.e. compression of a pair removes 1 position and not 2.

Consider first (Hd1), i.e. positions of letters popped from a variable. For each variable we pop perhaps several letters to the left and right before block compression, but those letters are immediately replaced with single letters, so we count each as 1; also, when this side of a variable becomes blocked, it can pop at most one letter (Lemma 9). Otherwise, a side of a variable pops at most 1 letter per pair compression, in which it is unblocked from this side. Note that the depint is the same as for the variable, so the encoding size is $\|X\|$. So in total the encoding size (without the numbers) of popped letters is at most:

$$\underbrace{\sum_{X\in\mathcal{X}} 2n_X \cdot \|X\|}_{\text{block compression}} + \underbrace{\sum_{X\in\mathcal{X}} 2n_X \cdot \|X\|}_{\text{after } X \text{ becomes blocked}} + \sum_{P:\,\text{partition}} \left(\sum_{\substack{X\in\mathcal{X}\\ \text{left unblocked in } P}} n_X \cdot \|X\| + \sum_{\substack{X\in\mathcal{X}\\ \text{right unblocked in } P}} n_X \cdot \|X\| \right). \tag{5}$$

Consider the summands of the third sum, so the ones depending on the partition. The one for the first partition is equal to $\sum_X 2n_X \cdot \|X\|$, as no side of the variable is blocked. By the strategy point (1), we choose a partition that at least halves this value every 4th pair compression, existence of such a partition is guaranteed by Lemma 10, the same Lemma also guarantees that this sum cannot increase at other choices of the partition. Thus (5), i.e. the increase during a phase due to (Hd1), is at most

$$\begin{aligned}4\sum_X n_X \cdot \|X\| + 8\sum_X n_X \cdot \|X\| \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) &= 20\sum_X n_X \cdot \|X\|\\ &\le 20\|(U_0, V_0)\|\ .\end{aligned}$$

We now similarly estimate how many positions got into $\mathrm{Pos}_{\supseteq}(i)$ due to expansion of $\mathrm{Pos}_{\supseteq}(i)$, i.e. the increase due to (Hd2): $\mathrm{Pos}_{\supseteq}(i)$ can extend to two letters during the block compression (to be more precise: to positions that are inside a block and to neighbouring blocks to the left/right of the block, but positions in a block are replaced with a single position and one of them was in $\mathrm{Pos}_{\supseteq}(i)$, so there is no increase in the middle of a block), to one position at each side after i becomes blocked (Lemma 9) and by one position for each partition P in which this side of i is not blocked. So the increase in the encoding size is at most

$$\underbrace{\sum_{i\text{: index}} 2\|U_0V_0[i]\|}_{\text{block compression}} + \underbrace{\sum_{i\text{: index}} 2\|U_0V_0[i]\|}_{\text{after blocked}} + \sum_{P\text{: partition}} \Bigg(\sum_{\substack{i\text{: index}\\ \text{left unblocked in } P}} \|U_0V_0[i]\| + \sum_{\substack{i\text{: index}\\ \text{right unblocked in } P}} \|U_0V_0[i]\| \Bigg) \ . \quad (6)$$

As in (5) consider the summands of the third sum, so the ones depending on the partition. The first summand is $\sum_{i\text{: index}} 2\|U_0V_0[i]\| = 2\|(U_0, V_0)\|$ and this value at least halves every 4th partition, by strategy point (2) (this is possible by Lemma 10). Thus similar calculations show that (6) is at most $20\|(U_0, V_0)\|$.

Lastly, we lower bound the decrease due to (Hd3):

Claim 4. *Given $Pos_{\supseteq}(i)$ at the beginning of a phase, at least $\frac{|Pos_{\supseteq}(i)|}{3} - 1$ of those positions are removed due to the compression during the phase.*

- If $U_0V_0[i]$ is a letter, then $\mathrm{Pos}_{\supseteq}(i)$ are all positions of letters and Lemma 3 yields that $\mathrm{Pos}_{\supseteq}(i)$ loses at least $\frac{|\mathrm{Pos}_{\supseteq}(i)|-1}{3} \geq \frac{|\mathrm{Pos}_{\supseteq}(i)|}{3} - 1$ positions.
- If $U_0V_0[i]$ is an ending marker, then the marker itself is unchanged and the remaining positions in $\mathrm{Pos}_{\supseteq}(i)$ are letter-positions and Lemma 3 applies to them, so $\mathrm{Pos}_{\supseteq}(i)$ loses at least $\frac{|\mathrm{Pos}_{\supseteq}(i)|-2}{3} > \frac{|\mathrm{Pos}_{\supseteq}(i)|}{3} - 1$ positions.
- If $U_0V_0[i]$ is a variable then $\mathrm{Pos}_{\supseteq}(i)$ may include the position of a variable. If it does not, then the analysis is the same as in the first case. If it includes the position of the variable then Lemma 3 applies to strings of letters to the left and right of the variable, say of length ℓ, r, where $\ell + r = |\mathrm{Pos}_{\supseteq}(i)| - 1$. Then due to compressions $\mathrm{Pos}_{\supseteq}(i)$ loses at least $\frac{\ell-1}{3} + \frac{r-1}{3} = \frac{|\mathrm{Pos}_{\supseteq}(i)|}{3} - 1$ positions,

which shows the claim. △

Thus:

$$\begin{aligned} H_d(U', V') &\leq H_d(U, V) + \underbrace{40\|(U_0, V_0)\|}_{(\text{Hd1}) \text{ and } (\text{Hd2})} - \underbrace{\sum_{i\text{: index}} \|U_0V_0[i]\| \cdot \left(\frac{1}{3}|\mathrm{Pos}_{\supseteq}(i)| - 1\right)}_{(\text{Hd3})} \\ &= \frac{2}{3}H_d(U, V) + 40\|(U_0, V_0)\| + \sum_{i\text{: index}} \|U_0V_0[i]\| \\ &= \frac{2}{3}H_d(U, V) + 41\|(U_0, V_0)\| \ . \end{aligned}$$

We also estimate the maximal value of H_d during the phase, as for intermediate equations we cannot guarantee that the compression reduced the length of all letters. We already showed that in a phase we increase H_d by at most $40\|(U_0, V_0)\|$. This yields a bound of

$$H_d(U'', V'') \leq H_d(U, V) + 40\|(U_0, V_0)\| \ .$$

We move to the proof for H_n. To shorten the notation, let $h(x) = x\log(x+1)$. Fix a phase, for an index i let b_i, d_i, e_i denote, respectively:

- $b_i = |\mathrm{Pos}_{\supseteq}(i)|$ at the beginning of the phase (so b_i stands for *b*eginning), note that by definition
$$\begin{aligned} H_n(U, V) &= 3\sum_{i\text{:index}} |\mathrm{Pos}_{\supseteq}(i)| \cdot \log(|\mathrm{Pos}_{\supseteq}(i)| + 1) \\ &= 3\sum_{i\text{:index}} h(b_i) \ . \end{aligned}$$
- d_i is the number of positions of letters popped from a variable with depint i (so d_i stands for *d*epint, $d_i = 0$ when no variable has depint $\{i\}$).
- e_i is the number of positions to which $\mathrm{Pos}_{\supseteq}(i)$ extended due to update of depints (so e_i stands for *e*xtended).

First we estimate $\sum_{i\text{: index}} h(d_i)$ and $\sum_{i\text{: index}} h(e_i)$ and then use those estimations to calculate the bound on $H_n(U', V')$. We first inspect the case of d_i; let $P_1, P_2, \ldots$ denote the consecutive partitions in phase. We show that

$$\sum_{i\text{: index}} h(d_i) \leq \sum_{X \in \mathcal{X}} 25n_X + \sum_{m \geq 1} m \cdot \Bigg(\sum_{\substack{X \in \mathcal{X}\\ \text{left unblocked in } P_m}} n_X + \sum_{\substack{X \in \mathcal{X}\\ \text{right unblocked in } P_m}} n_X \Bigg). \quad (7)$$

To see that (7) holds, consider one occurrence of X. Suppose it popped $q_X \geq 0$ letters. Then it was not blocked on the left (right) side for $q_{X,\ell}$ ($q_{X,r}$, respectively) partitions, where $q_{X,\ell} + q_{X,r} \geq q_X - 4$, as from each side we can pop once for block compression (formally, a sequence is popped, but it is immediately replaced with a single letter), once after the side becomes blocked, see Lemma 9, and at most once for each pair compression in which the side is not blocked. Then on the right-hand side of (7) the contribution from one occurrence of X is at least

$$\begin{aligned}
25 + \sum_{m=1}^{q_{X,\ell}} m + \sum_{m=1}^{q_{X,r}} m &= 25 + \frac{q_{X,\ell}(q_{X,\ell}+1) + q_{X,r}(q_{X,r}+1)}{2} \\
&= 25 + \frac{q_{X,\ell}^2 + q_{X,r}^2}{2} + \frac{q_{X,r} + q_{X,\ell}}{2} \\
&\geq 25 + \frac{(q_{X,\ell} + q_{X,r})^2}{4} + \frac{q_{X,r} + q_{X,\ell}}{2}
\end{aligned}$$

If $q_X \leq 3$ then $q_X \log(q_X + 1) \leq 6$ and so clearly the above sum is greater than $q_X \log(q_X + 1)$. Otherwise $q_{X,\ell} + q_{X,r} \geq q_X - 4 \geq 0$ and so

$$\begin{aligned}
25 + \sum_{m=1}^{q_{X,\ell}} m + \sum_{m=1}^{q_{X,r}} m &\geq 25 + \frac{(q_X - 4)^2}{4} + \frac{q_X - 4}{2} \\
&\geq q_X \log(q_X + 1) \ ,
\end{aligned}$$

where the last inequality can be checked by simple numerical calculation. Lastly, in (7) each d_i is equal to q_X, when $U_0V_0[i] = X$, or to 0, otherwise.

The sum in brackets on the right-hand side of (7) initially is at most $2|U_0V_0| \leq 2\|(U_0, V_0)\|$ and by strategy choice (3) it is at least halved every 4th step (this is possible by Lemma 10). So this sum is at most:

$$\begin{aligned}
\sum_{k \geq 0} \underbrace{(16k+10)}_{\text{steps } 4k+1,\ldots,4k+4} \cdot \underbrace{2 \cdot \|(U_0, V_0)\|}_{\text{initial size}} \cdot \left(\frac{1}{2}\right)^k &= 32\|(U_0, V_0)\| \cdot \underbrace{\sum_{k \geq 0} k \cdot \left(\frac{1}{2}\right)^k}_{=2} + 20\|(U_0, V_0)\| \cdot \underbrace{\sum_{k \geq 0} \left(\frac{1}{2}\right)^k}_{=2} \\
&= 104\|(U_0, V_0)\|
\end{aligned}$$

and consequently

$$\sum_{i\text{: index}} h(d_i) \leq 129\|(U_0, V_0)\| \ . \tag{8}$$

The analysis for e_i is similar: fix an index i and suppose that it extended to q_i positions. Then, similarly, it was not blocked on the left (right) side for $q_{i,\ell}$ ($q_{i,r}$, respectively) partitions, where $q_{i,\ell} + q_{i,r} \geq q_i - 4$: $\mathrm{Pos}_{\supseteq}(i)$ can extend once for block compression to each side (formally, to the positions of the whole block to the left or right of $\mathrm{Pos}_{\supseteq}(i)$, but those positions are immediately replaced with a single position), once after the side becomes blocked, see Lemma 9, and at most once for each pair compression in which the side is not blocked. Hence the calculations are the same as in the previous case, which leads to

$$\sum_{i\text{: index}} h(e_i) \leq 129\|(U_0, V_0)\| \ . \tag{9}$$

We now estimate, how many positions were removed from $\mathrm{Pos}_{\supseteq}(i)$ due to compression, recall that b_i is the size of $\mathrm{Pos}_{\supseteq}(i)$ at the beginning of the phase. By Claim 4 at least $\frac{b_i}{3} - 1$ positions were removed during the phase due to compression. Thus

$$H_n(U', V') \leq 3 \sum_{i\text{: index}} h\left(\frac{2}{3}b_i + 1 + d_i + e_i\right). \tag{10}$$

Consider two subcases: if $\frac{2}{3}b_i + 1 + d_i + e_i \leq \frac{5}{6}b_i$ (which implies $b_i \geq 6$), then the summand can be estimated as $h(\frac{5}{6}b_i) \leq \frac{5}{6}h(b_i)$ and we can upper bound the sum over those cases by $\frac{5}{6}\sum_{i\text{: index}} h(b_i)$. If $\frac{2}{3}b_i + 1 + d_i + e_i > \frac{5}{6}b_i$ then $1 + d_i + e_i > \frac{1}{6}b_i$ and so $\frac{2}{3}b_i + 1 + d_i + e_i < 5(1 + d_i + e_i)$. Thus (10) is upper-bounded by:

$$H_n(U', V') < 3 \cdot \left(\frac{5}{6} \sum_{i\text{: index}} h(b_i) + \sum_{i\text{: index}} h(5(1 + d_i + e_i))\right).$$

In the following, we estimate the second sum. As h is convex, we get $h(x+y+z) \le (h(3x)+h(3y)+h(3z))/3$ by Jensen's inequality and so

$$\sum_{i\text{: index}} h(5(1+d_i+e_i)) \le \frac{1}{3} \sum_{i\text{: index}} (h(15)+h(15d_i)+h(15e_i)) \enspace .$$

Consider $h(15x)$ for natural x. If $x=0$ then $h(15x)=h(x)=0$, when $x=1$ then $h(15x)=h(15)=60=60h(x)$, and otherwise

$$\begin{aligned} h(15x) &= 15x\log(15x+1) \\ &\le 15x\log(15(x+1)) \\ &< 15x(4+\log(x+1)) \\ &\le 60x\log(x+1) \\ &= 60h(x) \enspace . \end{aligned}$$

And so

$$\begin{aligned} \frac{1}{3}\sum_{i\text{: index}} (h(15)+h(15d_i)+h(15e_i)) \le \sum_{i\text{: index}} &(20+20h(d_i)+20h(e_i)) \\ &\le 20\|(U_0,V_0)\| + 2580\|(U_0,V_0)\| + 2580\|(U_0,V_0)\| \\ &= 5180\|(U_0,V_0)\| \enspace , \end{aligned}$$

and so plugging into the initial estimations we get

$$\begin{aligned} H_n(U',V') &\le 3\cdot\frac{5}{6}\sum_{i\text{: index}} h(b_i) + 3\cdot 5180\|(U_0,V_0)\| \\ &= \frac{5}{6}H_n(U,V) + 15540\|(U_0,V_0)\| \enspace . \end{aligned}$$

Taking the upper bounds on $H_d(U',V')$ and $H_n(U',V')$ together yields the claimed estimation upper bound on $H(U',V')$:

$$\begin{aligned} H(U',V') &= H_d(U',V') + H_n(U',V') \\ &\le \underbrace{\frac{2}{3}H_d(U,V) + 41\|(U_0,V_0)\|}_{\ge H_d(U',V')} + \underbrace{\frac{5}{6}H_n(U,V) + 15540\|(U_0,V_0)\|}_{\ge H_n(U',V')} \\ &\le \frac{5}{6}(H_d(U,V)+H_n(U,V)) + 15581\|(U_0,V_0)\| \\ &= \frac{5}{6}H(U,V) + 15581\|(U_0,V_0)\| \enspace , \end{aligned}$$

as claimed.

We should upper-bound the maximal H_n value during the phase, i.e. on an arbitrary equation $U''=V''$ during a phase, as inside a phase we cannot guarantee that letters get compressed, i.e. we use a trivial upper-bound

$$H_n(U'',V'') \le 3\sum_{i\text{: index}} h\left(b_i+d_i+e_i\right) \enspace .$$

Using a similar calculation as in the case of (10) and properties of h we obtain:

$$3\sum_{i\text{: index}} h\left(b_i+d_i+e_i\right) \le \sum_{i\text{: index}} (h(3b_i)+h(3d_i)+h(3e_i)) \enspace .$$

For $x=0$ we have $h(3x)=h(x)=0$ and for $x\ge 1$:

$$\begin{aligned} h(3x) &= 3x\log(3x+1) \\ &\le 3x\log(3(x+1)) \\ &< 3x(2+\log(x+1)) \\ &\le 9x\log(x+1) \\ &= 9h(x) \enspace . \end{aligned}$$

This allows estimating the value of H_n on arbitrary equation $U'' = V''$ during the phase:

$$\begin{aligned}
H_n(U'', V'') &\le 3 \sum_{i\text{: index}} h(b_i + d_i + e_i) \\
&\le H_n(U, V) + \sum_{i\text{: index}} (h(3b_i) + h(3d_i) + h(3e_i)) \\
&\le 9 \sum_{i\text{: index}} (h(b_i) + h(d_i) + h(e_i)) \\
&= 9 \underbrace{\sum_{i\text{: index}} h(b_i)}_{=H_n(U,V)/3} + 9 \underbrace{\sum_{i\text{: index}} h(d_i)}_{\le 129\|(U_0,V_0)\|} + 9 \underbrace{\sum_{i\text{: index}} h(e_i)}_{\le 129\|(U_0,V_0)\|} \\
&\le 3H_n(U, V) + 2322\|(U_0, V_0)\| \enspace .
\end{aligned}$$

Taking the bounds on H_d and H_n together:

$$\begin{aligned}
H(U'', V'') &= H_d(U'', V'') + H_n(U'', V'') \\
&\le H_d(U, V) + 40\|(U_0, V_0)\| + 3H_n(U, V) + 2322\|(U_0, V_0)\| \\
&\le 3H(U, V) + 2362\|(U_0, V_0)\| \enspace ,
\end{aligned}$$

as claimed. □

4.3. Proof of Theorem 1

We give the proof for the first claim of Theorem 1, which concerns LinWordEqSat, discussion about the second claim is given directly after Theorem 1. By Lemma 11 it is enough to give a linear (in terms of $\|(U_0, V_0)\|_0$) bound on $H(U, V)$ for an equation $U = V$.

By Lemma 2 all our subprocedures are sound, so we never accept an unsatisfiable equation. Thus by Lemma 1 it is enough to show that for some non-deterministic choices the algorithm uses linear space. Consider an equation $U = V$ at the beginning of the phase. Let Γ be the set of letters in this equation. If it has a solution $S^\bullet$, then it also has a solution S over Γ such that $|S(X)| = |S^\bullet(X)|$ for each variable: we can replace letters outside Γ with a fixed letter from Γ. During the phase we will make non-deterministic choices corresponding to this S, see discussion after Lemma 2 for the definition.

Let the equation obtained at the end of the phase be $U' = V'$ and the solution corresponding to S be S'. Then $|S'(U')| \le \frac{2|S(U)|+1}{3}$ by Lemma 3 and the equation $U' = V'$ at the beginning of the next phase has solution S'; if it has some letters that are not used in the equation, then we switch to another solution S'' such that $|S''(U')| = |S'(U')|$ and it has only letters that occur in the equation; we can do it in the same way as we switched from $S^\bullet$ to S for $U = V$. Hence we terminate after $\mathcal{O}(\log N)$ phases, where N is the length of some solution of the input equation. In fact, it is known that there always is a solution which is at mostly doubly exponential [18], so we terminate after an exponential number of phases.

Let the algorithm (non-deterministically) choose the partitions according to the strategy. We show by induction that for an equation (U, V) at the beginning of a phase $H(U, V) \le 6\alpha\|(U_0, V_0)\|$, where $\alpha \ge 1$ is the constant from Lemma 12. Initially $H_n(U_0, V_0) = 3\|(U_0, V_0)\|$ and $H_d(U_0, V_0) = \|(U_0, V_0)\|$, as for each index $\text{depint}(i) = \{i\}$; hence $H(U_0, V_0) = 4\|(U_0, V_0)\|$ and the claim holds. Then by Lemma 12 the inequality holds also at the end of each phase, as $\frac{5}{6} \cdot 6\alpha\|(U_0, V_0)\| + \|(U_0, V_0)\| = 6\alpha\|(U_0, V_0)\|$. For intermediate equation (U'', V'') inside a phase, by Lemma 12

$$H(U'', V'') \le \beta \cdot H(U, V) + \gamma\|(U_0, V_0)\| \enspace .$$

Here $U = V$ is the equation at the beginning of the phase. Using the proved bound $H(U, V) \le 6\alpha\|(U_0, V_0)\|$ yields

$$\begin{aligned}
H(U'', V'') &\le \beta \cdot (6\alpha\|(U_0, V_0)\|) + \gamma\|(U_0, V_0)\| \\
&= (6\alpha\beta + \gamma)\|(U_0, V_0)\| \enspace ,
\end{aligned}$$

where β, γ are the constants from Lemma 12.

To upper-bound the space consumption, we also need to estimate other stored information: we also store the alphabet from the beginning of the phase (this is linear in the size of the equation at the beginning of the phase) and the mapping of this alphabet to the current symbols (linear in the equation at the beginning of the phase plus the size of the current equation). The terminating condition that some pair of letters in Γ^2 was not covered is guessed non-deterministically, i.e. whenever the algorithm needs to decide, whether each pair in Γ^2 was covered, it makes a non-deterministic guess. For appropriate guesses, we will indeed cover all letters in Γ^2 and the algorithm will use space as analysed above; incorrect guess (too early termination of the phase, too late termination) may lead to larger space consumption, but due to Lemma 1 we do not need to analyse the space consumption in such cases. The pair compression and block compression can be performed in linear space, see Lemma 6. Note that this includes the change of Huffman coding.

CRediT authorship contribution statement

Artur Jeż: Conceptualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

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

Acknowledgment

This work was supported under National Science Centre, Poland project number 2014/15/B/ST6/00615.

I would like to thank anonymous reviewers, who helped in improving the presentation as well as found some errors in the previous versions of this paper.

References

[1] L. Ciobanu, M. Elder, Solutions sets to systems of equations in hyperbolic groups are EDT0L in PSPACE, in: C. Baier, I. Chatzigiannakis, P. Flocchini, S. Leonardi (Eds.), 46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9–12, 2019, Patras, Greece, in: LIPIcs, vol. 132, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 110.
[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press and McGraw-Hill Book Company, 1989.
[3] T.M. Cover, J.A. Thomas, Elements of Information Theory, Wiley, 2001.
[4] V. Diekert, M. Elder, Solutions to twisted word equations and equations in virtually free groups, Int. J. Algebra Comput. 30 (2020) 731–819, https://doi.org/10.1142/s0218196720500198.
[5] V. Diekert, C. Gutiérrez, C. Hagenah, The existential theory of equations with rational constraints in free groups is PSPACE-complete, Inf. Comput. 202 (2005) 105–140, https://doi.org/10.1016/j.ic.2005.04.002.
[6] V. Diekert, A. Jeż, M. Kufleitner, Solutions of word equations over partially commutative structures, in: I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, D. Sangiorgi (Eds.), ICALP, in: LIPIcs, vol. 55, Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2016, 127.
[7] V. Diekert, A. Jeż, W. Plandowski, Finding all solutions of equations in free groups and monoids with involution, Inf. Comput. 251 (2016) 263–286, https://doi.org/10.1016/j.ic.2016.09.009.
[8] V. Diekert, M. Lohrey, Word equations over graph products, Int. J. Algebra Comput. 18 (2008) 493–533.
[9] C. Gutiérrez, Satisfiability of word equations with constants is in exponential space, in: FOCS, 1998, pp. 112–119.
[10] J. Jaffar, Minimal and complete word unification, J. ACM 37 (1990) 47–85.
[11] A. Jeż, Recompression: a simple and powerful technique for word equations, J. ACM 63 (2016) 4, https://doi.org/10.1145/2743014.
[12] A. Jeż, Word equations in nondeterministic linear space, in: I. Chatzigiannakis, P. Indyk, F. Kuhn, A. Muscholl (Eds.), ICALP, in: LIPIcs, vol. 80, Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2017, 95.
[13] A. Jeż, Deciding context unification, J. ACM 66 (2019) 39, https://doi.org/10.1145/3356904.
[14] A. Kościelski, L. Pacholski, Complexity of Makanin's algorithm, J. ACM 43 (1996) 670–684, https://doi.org/10.1145/234533.234543.
[15] G. Makanin, The problem of solvability of equations in a free semigroup, Mat. Sb. 2 (1977) 147–236 (in Russian).
[16] G. Makanin, Equations in a free group, Izv. Akad. Nauk SSSR, Ser. Mat. 46 (1983) 1199–1273; English transl. in Math. USSR, Izv. 21 (1983).
[17] Y. Matiyasevich, Some decision problems for traces, in: S. Adian, A. Nerode (Eds.), LFCS, in: LNCS, vol. 1234, Springer, 1997, pp. 248–257, invited lecture.
[18] W. Plandowski, Satisfiability of word equations with constants is in NEXPTIME, in: STOC, ACM, 1999, pp. 721–725.
[19] W. Plandowski, Satisfiability of word equations with constants is in PSPACE, J. ACM 51 (2004) 483–496, https://doi.org/10.1145/990308.990312.
[20] W. Plandowski, On PSPACE generation of a solution set of a word equation and its applications, Theor. Comput. Sci. 792 (2019) 20–61, https://doi.org/10.1016/j.tcs.2018.10.023.
[21] W. Plandowski, W. Rytter, Application of Lempel-Ziv encodings to the solution of word equations, in: K.G. Larsen, S. Skyum, G. Winskel (Eds.), ICALP, in: LNCS, vol. 1443, Springer, 1998, pp. 731–742.
[22] A.A. Razborov, On Systems of Equations in Free Groups, Ph.D. thesis, Steklov Institute of Mathematics, 1987 (in Russian).
[23] E. Rips, Z. Sela, Canonical representatives and equations in hyperbolic groups, Invent. Math. 120 (1995) 489–512.
[24] K.U. Schulz, Makanin's algorithm for word equations—two improvements and a generalization, in: K.U. Schulz (Ed.), IWWERT, in: LNCS, vol. 572, Springer, 1990, pp. 85–150.