Contents

1 Introduction **1**

2 Satisfiability of word equations **1**

3 Exponent of periodicity **4**
3.1 Visible blocks and their lengths 4
3.2 Arithmetic expressions 5
3.3 System of integer equations 5
3.4 Parametrised solutions 6
3.5 Solutions of system of linear Diophantine equations 6
3.6 Bound on Σ-exponent of periodicity 7

4 LZ77 and SLPs **8**
4.1 LZ77 8
4.2 SLP 9
4.3 Composition systems 9

5 Short proof for small SLP representation of length-minimal solution **9**

6 Smallest grammar problem **10**
6.1 Basic notations 11
6.2 AVL grammar 11
6.3 From LZ77 to AVL grammar 12

7 Generalising Recompression from word equations to SLPs **12**
7.1 Size 13
7.2 Pair compression 14
7.3 Faster block compression 17

8 Approximation of the smallest SLP using recompression **20**
8.1 Intuition and road map 21
8.1.1 Modifying the grammar 21
8.1.2 Block compression 21
8.1.3 Paying the representation cost: credit 22
8.1.4 Additional cost 22
8.2 Estimation of credit 22
8.3 Calculating the cost of representing letters in block compression 23
8.3.1 G-based representation 23
8.3.2 Cost of G-based representation 24
8.3.3 Comparing the G-based representation cost and texttoSLP-based representation cost 24

9 Equality testing: similar approach **26**
9.1 How to calculate assignment 27
9.2 Storing 28
9.3 Update 28
9.4 Comments 28

10 Compressed pattern matching: Combinatorial approach **29**
10.1 AP table . . . 29
10.1.1 Filling AP using LSP . . . 29
10.2 Local search procedure . . . 30

11 Quadratic word equations **30**

12 Word equations: limited number of variables **31**
12.0.1 One variable . . . 31
12.1 One-variable equations . . . 31
12.2 Representation of solutions . . . 32
12.2.1 Preserving solutions . . . 33
12.3 Specialisation of precodures . . . 33
12.4 The algorithm . . . 35

13 Free groups **38**
13.1 Free groups . . . 38
13.2 Free monoids/semigroups with involution . . . 38
13.3 Word equation in free groups . . . 39
13.4 Reduction: equations in groups to word equations . . . 39

14 Positive theory of free groups and free semigroups with recognisable constraints **39**
14.1 General comments . . . 40
14.2 Quantifier elimination . . . 40
14.3 Proof of Lemma 67 . . . 42

15 Solving equations in free groups **43**
15.1 Recognisable/Regular sets . . . 43
15.2 Regular constraints . . . 44
15.3 Main issue . . . 44
15.4 Goal . . . 44
15.5 Needed modifications . . . 45
15.5.1 Constraints . . . 45
15.5.2 Involution . . . 45
15.5.3 Pair compression . . . 45
15.5.4 Blocks and Quasiblocks compression . . . 45
15.6 Letters . . . 45

16 Representation of all solutions **47**

17 Terms and Unification **49**
17.1 Labelled trees . . . 49
17.2 What the variables represent . . . 49
17.3 Patterns . . . 49
17.4 Second order unification . . . 50

18 Linear Monadic Second Order Unification **50**

19 General second order unification **53**

20 Context Unification **55**
20.1 Introduction 55
20.1.1 Context unification 55
20.1.2 Extensions and connections to other problems 57
20.1.3 Context unification and word equations 58
20.2 Compression of trees 59
20.2.1 Patterns 59
20.2.2 Local compression of trees 59
20.3 Context unification 60
20.4 Compressions on the equation 61
20.5 Uncrossing 63
20.6 Uncrossing father-leaf subpattern 63
20.7 Uncrossing patterns 66
20.8 The algorithm 66
20.9 Analysis of the algorithm 67
20.9.1 Input signature 67
20.9.2 Exponent of periodicity 68
20.9.3 Occurrences of variables 69
20.9.4 Size bounds 71
20.9.5 Simple solutions 74

1 Introduction

In context of word equations we always consider a finite alphabet Σ and finite set of variables Ω, which is disjoint with Σ. Elements of Σ are usually denoted by small letters $a, b, c, \ldots$. Elements of Ω are usually denoted as $X, Y, Z, \ldots$.

A *word equation* is a pair (u, v), usually written as $u = v$, where $u, v \in (\Sigma \cup \Omega)^*$. A *system of word equations* is a set of word equations, usually denoted as (u_1, v_1), (u_2, v_2), ...

A *substitution* is a morphism $S : \Omega \mapsto \Sigma^+$ (we sometimes also consider Σ^*, but then we explicitly state this). It is extended to Σ as an identity (so $S(a) = a$ for $a \in \Sigma$) and to $(\Sigma \cup \Omega)^*$ as a homomorphism (so $S(\alpha\beta) = S(\alpha)S(\beta)$ for $\alpha, \beta \in (\Sigma \cup \Omega)^+$.

A substitution is a *solution* of a word equation $u = v$, when $S(u) = S(v)$ (the solution of a system of equations is defined similarly). A solution S of a word equation $u = v$ is *length-minimal* (or simply *minimal*), when for any other solution S' it holds that

$$|S(u)| \leq |S'(u)| \ .$$

A *satisfiability problem* for word equations is:
"Given a system of word equations decide, whether they have a solution."

We say that a system of word equations is *quadratic*, if every variable occurs at most twice in it. It is *cubic*, when every variable occurs at most thrice.

Constraints for system of word equations are given as additional constraints of the form $X \in C$ or $X \notin C$, where $X \in \Omega$ and C comes from some specified language class (say: regular, context-free, etc.). The meaning of the constraint $X \in C$ (or $X \notin C$) is that we require from a solution S that $S(X) \in C$ (or $S(X) \notin C$).

2 Satisfiability of word equations

Algorithm 1 Compression of a word w

```
1: while |w| > 1 do
2:     L ← list of letters in w
3:     for a ∈ L do
4:         compress blocks of a                    ▷ Replace a^ℓ with a_ℓ
5:     P ← list pairs in w
6:     for ab ∈ P do
7:         replace all occurrences of ab in w by a fresh letter c
```

We say that a nondeterministic procedure *is sound*, when given a unsatisfiable word equation $U = V$ it cannot transform it to a satisfiable one, regardless of the nondeterministic choices; such a procedure *is complete*, if given a satisfiable equation $U = V$ for some nondeterministic choices it returns a satisfiable equation $U' = V'$. Observe, that a composition of sound (complete) procedures is sound (complete, respectively)

Lemma 1. *The following operations are sound:*

1. *replacing all occurrences of a variable X with wXv for arbitrary $w, v \in \Gamma^*$;*
2. *replacing all occurrences of a word $w \in \Gamma^+$ (in U and V) with a fresh letter c;*
3. *replacing occurrences of a variable X with a word w.*

Proof. In the first case, if S' is a solution of $U' = V'$ then S defined as $S(X) = wS'(X)v$ and $S(Y) = S'(Y)$ otherwise is a solution of $U = V$.

In the second case, if S' is a solution of $U' = V'$ then S obtained from S' by replacing each c with w is a solution of $U = V$.

Lastly, in the third case, if S' is a solution of $U' = V'$ then we can obtain S from S' by defining the substitution $S(X) = w$ and $S(Y) = S'(Y)$ in other cases. □

Definition 2. Given an equation $u = v$ and a substitution S and a substring $w \in \Sigma^+$ of $S(U)$ (or $S(V)$) we say that this occurrence of w is

- *explicit*, if it comes from substring w of u (or v, respectively)
- *implicit*, it it comes from $S(X)$ for an occurrence of a variable X
- *crossing* otherwise.

A string w is *crossing* (with respect to a solution S) if it has a crossing occurrence and *non-crossing* (with respect to a solution S) otherwise.

We say that a pair of ab is a *crossing pair* (with respect to a solution S), if ab has a crossing occurrence. Otherwise, a pair is *non-crossing*. Unless explicitly stated, we consider crossing/non-crossing pairs ab in which $a \neq b$. Similarly, a letter $a \in \Sigma$ has a *crossing block*, if there is a maximal block of a which has a crossing occurrence. This is equivalent to a (simpler) condition that aa is a crossing pair.

Algorithm 2 PairCompNCr(a, b) Pair compression for a non-crossing pair

let $c \in \Sigma$ be an unused letter
replace each explicit ab in U and V by c

Algorithm 3 BlockCompNCr(a) Block compression for a letter a with no crossing block

for each explicit a occurring in U or V **do**
    for each ℓ that is a visible length of an a block in U or V **do**
        let $a_\ell \in \Sigma$ be an unused letter
        replace every explicit a's maximal ℓ-block occurring in U or V by a_ℓ

Lemma 3. PairCompNCr(a, b) *is sound and when ab is a non-crossing pair in an equation $U = V$ (with respect to some solution S) then it is complete and implements the pair compression of ab for S.*

Similarly, BlockCompNCr(a) *is sound and when a has no crossing blocks in $U = V$ (with respect to some solution S) it is complete; to be more precise, if S is a solution of $U = V$ such that ab is non-crossing with respect to S, then the new equation $U' = V'$ has a solution S' such that $S'(U')$ is obtained by compression of pair ab in $S(U)$.*

Proof. From Lemma 1 it follows that both PairCompNCr(a, b) and BlockCompNCr(a) are sound.

Suppose that $U = V$ has a solution S such that ab is a noncrossing pair with respect to S. Define S': $S'(X)$ is equal to $S(X)$ with each ab replaced with c (where c is a new letter). Consider $S(U)$ and $S'(U')$. Then $S'(U')$ is obtained from $S(U)$ by replacing each ab: the explicit occurrences of ab are replaced by PairCompNCr(a, b), the implicit ones are replaced by the definition of S' and by the assumption there are no crossing occurrences. The same applies to $S(V)$ and $S'(V')$. Hence $S'(U') = S'(V')$, which concludes the proof in this case.

The proof for the block compression follows in the same way. □

Algorithm 4 Pop(a, b)

for $X \in \Omega$ **do**
    let b be the first letter of $S(X)$ ▷ Guess
    if the first letter of $S(X)$ is b **then**
        replace each X in U and V by bX ▷ Implicitly change $S(X) = bw$ to $S(X) = w$
        if $S(X) = \epsilon$ **then** ▷ Guess
            remove X from U and V
    ... ▷ Perform a symmetric action for the last letter and a

Lemma 4. *The* Pop(a, b) *is sound and complete.*

Furthermore, if S is a solution of $U = V$ then for some nondeterministic choices the obtained $U' = V'$ has a solution S' such that $S'(U') = S(U)$ and ab is non-crossing (with regards to S').

Lemma 5. CutPrefSuff *is sound. It is complete, to be more precise: For a solution S of $U = V$ let for each X ℓ_X, r_X be the lengths of a-prefix and suffix of $S(X)$. Then when*

Algorithm 5 CutPrefSuff(a) Cutting a-prefixes and a-suffixes

for $X \in \Omega$ **do**
  let ℓ_X and r_X be the lengths of the a-prefix and suffix of $S(X)$ ▷ Guess
    ▷ If $S(X) = a_X^{\ell_X}$ then $r_X = 0$
  replace each X in U and V by $a^{\ell_X} X a^{r_X}$ ▷ ℓ_X and r_X are stored as bitvectors,
    ▷ implicitly change $S(X) = a^{\ell_X} w a^{r_X}$ to $S(X) = w$
  if $S(X) = \epsilon$ **then** ▷ Guess
    remove X from U and V

CutPrefSuff *pops* a^{ℓ_X} *to the left and* a^{r_X} *to the right, the returned equation* $U' = V'$ *has a solution* S' *such that* $S(U) = S'(U')$ *and* a *has no crossing blocks with respect to* S'.

Algorithm 6 WordEqSat Checking the satisfiability of a word equation

while $|U| > 1$ or $|V| > 1$ **do**
  $L \leftarrow$ list of letters without crossing blocks
  $L' \leftarrow$ list of letters with crossing blocks
  for $a \in L$ **do**
    BlockCompNCr(a)
  for $a \in L'$ **do**
    CutPrefSuff(a)
    BlockCompNCr(a)
  $P \leftarrow$ list pairs
  while there is non-crossing $ab \in P$ **do**
    take some non-crossing $ab \in P$
    PairCompNCr(a, b)
  for $ab \in P$ **do**
    Pop(a, b)
    PairCompNCr(a, b)
Solve the problem naively ▷ With sides of length 1, the problem is trivial

Lemma 6. *For appropriate nondeterministic choices, the equations stored by (successful) computation of* WordEqSat *are of length* $\mathcal{O}(n^2)$*, the additional computation performed by* WordEqSat *use* $\mathcal{O}(n^2)$ *space.*

3 Exponent of periodicity

In this section it is more conveniant to use s for a solution. By n we denote the length of the eqaution and by n_v the number of occurrences of variables in this equation.

Definition 7. For a word w the *exponent of periodicity* $\mathrm{per}(w)$ is the maximal k such that u^k is a substring of w, for some $u \in \Sigma^+$; Σ-*exponent of periodicity* $\mathrm{per}_\Sigma(w)$ restricts the choice of u to Σ.

The notion of exponent of periodicity is naturally transferred from strings to equations: For an equation $u = v$, define the exponent of periodicity as

$$\mathrm{per}(u = v) = \max_s \left[\mathrm{per}(s(u))\right] \ ,$$

where the maximum is taken over all length-minimal solutions s of $u = v$; define the Σ-*exponent of periodicity* of $u = v$ in a similar way.

3.1 Visible blocks and their lengths

Definition 8. A maximal a-block in $s(u)$ or $s(v)$ is *visible* for a solution s, if it contains an explicit letter or non-empty a-prefix or a-suffix of some occurrence of $s(X)$.

A *visible length* is a lenght of a visible (maximal) a-block.

Lemma 9. *There are at most $n + 2n_v$ different visible blocks.*

When s is length-minimal, each maximal a-block has a visible length.

Proof. One letter and one prefix/suffix belongs to at most one maximal a-block.

If such a block has a length that is not visible, then all maximal blocks of this length can be deleted (requires some further argument, but works). □

By e_1, e_2, ..., e_k we denote the lengths of visible a-blocks. Note that some of those values may be equal. By ℓ_X, r_X we denote the length of the a-prefix and a-suffix of $s(X)$, if $s(X) \in a^+$ then we set $r_X = 0$. W generally disregard those values, that are equal 0: simply remove them.

3.2 Arithmetic expressions

Definition 10. Arithmetic expressions may use positive constants and variables $\{L_X, R_X\}_{X\in\mathcal{X}}$. They are usually denoted by E_1, E_2, ..., E_k.

For a given word equation of length n a set of such expressions is a system of small arithmetic expressions when

- it uses variables $\{L_X, R_X\}_{X\in\mathcal{X}}$, where $\mathcal{X}$ is a set of variables used in the word eqution
- the sum of constants in those expressions is at most n
- variable L_X (R_X) is used in this set at most as many times as the variable X in the word equation

For an expression E depending on variables $\{L_X, R_X\}_{X\in\mathcal{X}}$ by $E[\{L_X, R_X\}_{X\in\mathcal{X}}]$ we denote the value that is obtained by substituting ℓ_X and r_X (that are natural numbers) for variables L_X and R_X, for each variable X.

Lemma 11. *For a given word equation with variables $\mathcal{X}$ and lengths of visible blocks e_1, e_2, ..., e_k there is a small set of arithmetic expressions E_1, E_2, ..., E_k in variables $\{L_X, R_X\}_{X\in\mathcal{X}}$ such that $e_i = E_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$, where ℓ_X and r_X are the lengths of the a-prefix and a-suffix of $s(X)$.*

Proof. Consider the maximal visible block and think of the a-prefixes and suffixes. □

3.3 System of integer equations

Define a system of equations: partition the expressions $e_1, e_2, \ldots, e_k$ into groups of the same value. For each such group $\{e_{i_1}, e_{i_2}, \ldots, e_{i_j}\}$ add equations

$$e_{i_1} = e_{i_2}, e_{i_2} = e_{i_3}, \ldots, e_{i_{j-1}} = e_{i_j}$$

And add to it inequalities $L_X > 0$ ($R_X > 0$), when ℓ_X exists, i.e. it is non-zero (the same for r_X). Call this system of equations $\mathcal{D}$.

Lemma 12. *$\{\ell_X, r_X\}_{X\in\mathcal{X}}$ is a solution of $\mathcal{D}$.*

For every solution of this system $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ it holds that

$$e_i = e_j \text{ implies } E_i[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}] = E_j[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$$

Proof. Straight from the definition: $E_i = E_j$ is added when $e_i = e_j$ and $E_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}] = e_i$.

Again, when $e_i = e_j$ then $E_i = E_j$ is a consequence of $\mathcal{D}$. □

3.4 Parametrised solutions

Consider a $s(X)$. Each maximal block of a in $s(X)$ that is not visible is of visible length. Assign to each such block one of e_i in an arbitrary way. To a visible block assign its length.

Define a *parametrised solution* $S(X)$: it is $s(X)$ in which we replace each block assigned with e_i with a^{E_i}; moreover, we replace the a-prefix and a-suffix with a^{L_X} and a^{R_X}.

Lemma 13. *$S(u)$ and $S(v)$ are obtained by replacing each maximal a-block of length e_i (or assigned length e_i) with a^{E_i}.*

Proof. Easy induction. □

We define $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}](X)$ in a natural way: each a^E we tun into $a^{E[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]}$ and each a^{L_X}, a^{R_X} into $a^{\ell'_X}$, $a^{r'_X}$.

Lemma 14. *Each $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is a well-defined substitution.*

$S[\{\ell_X, r_X\}_{X\in\mathcal{X}}] = s$.

Proof. First: obvious.

Second: as $E_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}] = e_i$. □

Theorem 15. *If $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ is a solution of $\mathcal{D}$ then $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is a solution of the word equation.*

Moreover this solution is obtained by replacing an a-maximal block of length e_i with $E_i[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$.

Proof. Both follow from the second observation in Lemma 12. □

3.5 Solutions of system of linear Diophantine equations

Consider a system of m linear Diophantine equations in r variables $x_1, \ldots, x_r$, written as

$$\sum_{j=1}^{r} n_{i,j} x_j = n_i \qquad \text{for } i = 1, \ldots, m \tag{1a}$$

together with inequalities guaranteeing that each x_i is positive

$$x_j \geq 1 \qquad \text{for } j = 1, \ldots, r \ . \tag{1b}$$

In the following, we are interested only in *natural* solutions, i.e. the ones in which each component is a natural number; observe that inequality (1b) guarantees that each of the component is greater than zero. We introduce a partial ordering on such solutions:

$$(q_1, \ldots, q_r) \geq (q'_1, \ldots, q'_r) \quad \text{if and only if} \quad q_j \geq q'_j \text{ for each } j = 1, \ldots, r.$$

A solution $(q_1, \ldots, q_r)$ is a *minimal* if it satisfies (1) and there is no solution smaller than it. (Note, that there may be incomparable minimal solutions.)

It is known, that each component of the minimal solution is at most exponential:

Lemma 16 (cf. [26, Corollary 4.4])**.** *For a system of linear Diophantine equations* (1) *let* $w = r + \sum_{i=1}^{m} |n_i|$ *and* $c = \sum_{i=1}^{m} \sum_{j=1}^{r} |n_{i,j}|$*. If* $(q_1, \ldots, q_r)$ *is its minimal solution, then* $q_j \leq (w + r)e^{c/e}$*.*

The proof is a slight extension of the original proof of Kościelski and Pacholski, which takes in to the account also the inequalities. For completeness, we recall its proof, as given in [26].

proof, cf. [26]. The proof follows by estimation based on work of [63] and independently by [28]

Claim 1 ([63]; [28])**.** *Consider a (vector) equations and inequalities* $Ax = B$*,* $Cx \geq D$ *with integer entries in* A*,* B*,* C *and* D*. Let* M *be the upper bound on the absolute values of the determinants of square submatrices of the matrix* $\begin{pmatrix} A \\ C \end{pmatrix}$*,* r *be the number of variables and* w *the sum of absolute values of elements in* B *and* D*. Then for each minimal natural solution* $(q_1, \ldots, q_r)$ *of* (1)*, for each* $1 \leq i \leq r$ *we have* $q_i \leq (w + r)M$*.* □

So it remains to estimate M from Claim 1, we recall the argument of [26].

Recall the Hadamard inequality: for any matrix $N = (n_{i,j})_{i,j=1}^{k}$ we have

$$\det^2(N) \leq \prod_{j=1}^{k} \sum_{i=1}^{k} n_{i,j}^2 \ .$$

Therefore

$$\begin{aligned}
\det(N) &\le \left(\prod_{j=1}^{k}\sum_{i=1}^{k} n_{i,j}^2\right)^{1/2} && \text{Hadamard inequality}\\
&\le \left(\prod_{j=1}^{k}\left(\sum_{i=1}^{k} |n_{i,j}|\right)^2\right)^{1/2} && \text{trivial}\\
&= \prod_{j=1}^{k}\sum_{i=1}^{k} |n_{i,j}| && \text{simplification}\\
&\le \left(\frac{\sum_{j=1}^{k}\left(\sum_{i=1}^{k}|n_{i,j}|\right)}{k}\right)^{k} && \text{inequality between means}\\
&\le \left(\frac{c}{k}\right)^{k} && \text{by definition } \sum_{j=1}^{k}\sum_{i=1}^{k}|n_{i,j}| = c\\
&\le e^{c/e} && \text{calculus: sup at } k = c/e.
\end{aligned}$$

Taking N to be any submatrix of $(n_{i,j})$ yields that $M \le e^{c/e}$ and consequently $q_i \le (w+r)e^{c/e}$, as claimed.

3.6 Bound on Σ-exponent of periodicity

We can now infer the upper-bound on the Σ-exponent of periodicity of the length-minimal solution of the word equation.

As a first step, let us estimate the values w, r, M from Lemma 16 in case of system of equations $\mathcal{D}$

Lemma 17. *For a system of equations $\mathcal{D}$ Lemma 16 yields a bound of*

$$\mathcal{O}(ne^{4n_v/e})$$

on coordinates of some of its solutions.

Proof. Concerning the sum of coefficients on the left hand side, for inequalities we have 1 per variable, so $2n_v$ in total, for equalities we have $2n_v$, as each expression can be used twice and they have sum of constants n. Thus $w = 2n_v + 2n$. For r: we have $2n_v$ variables.

For M, note that the matrix C in our case is an identity, it is enough to consider the bound on the values of determinants of square submatrices of A. Those are coefficients by the variables, so by definition of small set of arithmetic expressions, this sum is at most $4n_v$.

Thus the bound is $\mathcal{O}(ne^{4n_v/e})$. □

Lemma 18 (cf. [26])**.** *Consider a solution s of a word equation $u = v$, and a system $\mathcal{D}$ created for it. Consider all solutions $\{\ell'_X, r'_X\}_{X\in\Omega}$ of this system and the corresponding solutions $S[\{\ell'_X, r'_X\}_{X\in\Omega}]$. For a length-minimal s' among them the lengths of longest a-block in $s'(u)$ is $\mathcal{O}(n^2e^{8n_v/e})$).*

Proof. We know that all $S[\{\ell_X, r_X\}_{X\in\Omega}]$ are solutions. Let, as in the statement, s' be a length minimal among them, let it correspond to a solution $\{\ell'_X, r'_X\}_{X\in\Omega}$ of $\mathcal{D}$. Let

$\{\ell''_X, r''_X\}_{X\in\Omega}$ be a solution of $\mathcal{D}$ for which Lemma 17 gives a $\mathcal{O}(ne^{4n_v/e})$ on each ℓ''_X and r''_X; let s'' be the corresponding solution of the word equation. Then the total number of as in $s'(u)$ is not larger than the total number of as in $s''(u)$. But the latter is $\sum_i E_i[\{\ell''_X, r''_X\}_{X\in\Omega}]$ which is $\mathcal{O}(n_v n e^{4n_v/e})$ by Lemma 17. □

As a short corollary we obtain:

Theorem 19 (cf. [26]). *The Σ-exponent of periodicity of a word equation $U = V$ with n_v occurrences of variables is $\mathcal{O}(\mathrm{poly}(n)e^{4n_v/e})$).*

Proof. We can estimate the lengths of maximal a-blocks for each a separately by Lemma 18. □

4 LZ77 and SLPs

4.1 LZ77

An LZ77 *factorisation* or *parse* of a word w is a representation $w = f_1 f_2 \cdots f_\ell$, where each f_i is either a single letter (called *free letter* in the following) or $f_i = w[j\,..\,j + |f_i| - 1]$ for some $j \le |f_1 \cdots f_{i-1}|$, in such a case f is called a *factor* or *phrase* and $w[j\,..\,j + |f_i| - 1]$ is called the *definition* of this factor. The *size* of the LZ77 factorisation $f_1 f_2 \cdots f_\ell$ is ℓ. There are several simple and efficient linear-time algorithms for computing the smallest LZ77 factorisation of a word [1, 5, 9, 20, 25, 45, 21] and all of them rely on linear-time algorithm for computing the suffix array [24]. It is easy to show (exercise) that the greedy algorithm returns such smallest factorisation.

An LZ77 factorisation is called non-self-referencing, if for a factor f_i with definition $w[j\,..\,j + |f_i| - 1]$ we have that $j + |f_i| - 1 \le |f_1 f_2 \cdots f_i|$ and self-referencing otherwise.

The implicit assumption for the LZ77 is that the length of the text fits in $\mathcal{O}(1)$ memory cells.

Practical notes For practical purposes, a single letter is represented as itself and a factor is represented as an offset plus length. In practical implementation there is an upper bound on the offset. If there are several possible definitions, we take the last one (large offsets cause problems for access other than RAM).

4.2 SLP

Straight Line Programme (*SLP*) is a CFG in the Chomsky normal form that generates a unique string. Without loss of generality we assume that nonterminals of an SLP are $X_1, \ldots, X_g$, each rule is either of the form $X_i \to a$ or $X_i \to X_j X_k$, where $j, k < i$. The *size* of the SLP is the number of its nonterminals (here: g).

The problem of finding smallest SLP generating the input word w is NP-hard [61] and the size of the smallest grammar for the input word cannot be approximated within factor $\frac{8569}{8568}$ [4]. On the other hand, several algorithms with an approximation ratio $\mathcal{O}(1 + \log(N/g))$, where g is the size of the smallest grammar generating w, are known [4, 50, 52, 22].

Relation between SLP and LZ77

Lemma 20 ([50]). *Let ℓ and g be the sizes of the LZ77 and smallest SLP for a word. Then $\ell \leq g$.*

This bound is relatively easy to obtain: any SLP (of size k) defines a specific LZ77 factorisation (of size at most k), in particular, there is a factorisation of size g.

4.3 Composition systems

In many proofs it is easier to use the 'substring' approach of LZ77 rather than the SLP. Thus the *composition systems* are SLPs that additionally allow a usage of substrings of nonterminals, i.e. we can use $A[b : e]$ in a rule, its semantics is 'substring of a string generated by a from position$b$ to e'. It is easy to show that composition system can be transformed into an SLP with a polynomial size increase. (We shall show it later).

5 Short proof for small SLP representation of length-minimal solution

This is based on [48].

Definition 21. A *cut* (in an equation) is a position between two explicit letters in an equation or between two variables or between a variable and an equation. We generalise this notion to cut for a solution.

A substring in $S(u)$ or $S(v)$ *overlaps* a cut α, if α is within this word or at its beginning or end.

Note that there are $|u| + |v| + 2$ cuts in a word equation $u = v$.

Definition 22. For a function $f : \Omega \mapsto \mathbb{N}$ a solution S is an f-solution, if $|S(X)| = f(X)$ for each variable X.

Definition 23. Given a solution S we say that two position in $S(uv)$ are in $\mathcal{R}'$ relation, if:

- they are corresponding positions of $S(u)$ and $S(v)$ or
- they are corresponding positions of different occurrences of some $S(X)$

Define $\mathcal{R}$ as a transitive closure of $\mathcal{R}'$

Lemma 24. *Consider a solution S and the $\mathcal{R}$ relation, let f be such that S is an f-solution. Then*

1. *if there is a equivalence class corresponding containing no constant then S is not length-minimal. Moreover, the symbols at positions in this class can be filled with the same arbitrary string, in particular by ϵ.*
2. *For any two positions $i\mathcal{R}j$ an f-solution S' we have $S'(uv)[i] = S'(uv)[j]$.*
3. *There is an f-solution if and only if no equivalence class contains two positions corresponding to different constants.*

Proof. Rather obvious. □

Lemma 25. *Suppose that S is a length-minimal solution and w is a substring in $S(u)$. Then there is a substring w in $S(u)$ or $S(v)$ which overlaps with a cut.*

spoiler, as this is an exercise. Look at $\mathcal{R}$. What happens, when no letter in w touches a cut? And what when it does? □

In the following,we denote cuts by Greek letters. For a cut α let $(\alpha)_k$ be the word that extends 2^{k-1} to the left and right from α (truncate it, when this exceeds the $S(U)$ or $S(v)$).

Consider $(\alpha)_{k+1}$ and express it as

$$(\alpha)_{k+1} = w_k(\alpha)_k w'_k$$

where $|w_k| = |w'_k| = 2^{k-1}$.

By Lemma 25, we get that w_k and w'_k are substrings of some $(\beta)_k$ and $(\gamma)_k$ (note that they are of length 2^{k-1}, so when w overlaps α, it is within $(\alpha)_k$). Thus

$$(\alpha)_{k+1} = (\beta)_k[i\,..\,j](\alpha)_k(\gamma)_k[i'\,..\,j']$$

Treating $(\alpha)_{k+1}$ as nonterminals, we obtain a composition systems for those cuts. Now, for $k = \log N$ the $(\alpha)_{k+1}$ is actually the whole $S(u)$. Thus, we have a composition system of size $\mathcal{O}(n \log N)$ for the smallest solution (and so also the same size for each variable).

6 Smallest grammar problem

We now investigate the question of constructing a small SLP for a given string.

The first two algorithms with an approximation ratio $\mathcal{O}(\log(N/g))$ were developed simultaneously by Rytter [50] and Charikar et al. [4]. They followed a similar approach, we first present Rytter's approach as it is a bit easier to explain.

Rytter's algorithm [50] applies the LZ77 compression to the input string and then transforms the obtained LZ77 representation to an $\mathcal{O}(\ell \log(N/\ell))$ size grammar, where ℓ is the size of the LZ77 representation. It is easy to show that $\ell \leq g$ and as $f(x) = x \log(N/x)$ is increasing, the bound $\mathcal{O}(g \log(N/g))$ on the size of the grammar follows (and so a bound $\mathcal{O}(\log(N/g))$ on the approximation ratio). The crucial part of the construction is the requirement that the derivation tree of the intermediate constructed grammar satisfies the AVL condition. While enforcing this requirement is in fact easier than in the case of the AVL search trees (as the internal nodes do not store any data), it remains involved and non-trivial. Note that the final grammar for the input text is also AVL-balanced, which makes is suitable for later processing.

Charikar et al. [4] followed more or less the same path, with a different condition imposed on the grammar: it is required that the derivation tree is length-balanced, i.e. for a rule $X \to YZ$ the lengths of words generated by Y and Z are within a certain multiplicative constant factor from each other. For such trees efficient implementation of merging, splitting etc. operations were given (i.e. constructed from scratch) by the authors and so the same running time as in the case of the AVL grammars was obtained. Since all the operations are defined from scratch, the obtained algorithm is also quite involved and the analysis is even more non-trivial.

6.1 Basic notations

For a nonterminal A in an SLP G we use $\mathrm{val}(A)$ to denote the unique word generated by A, $\mathrm{val}(G)$ is the word generated by the starting nonterminal of G. For an SLP G by $\mathrm{Tree}(G)$ we denote the derivation tree of G, whose leaves are labelled with letters of $\mathrm{val}(G)$ We sometimes label the inner nodes by the corresponding nonterminals.

Denote by $\mathrm{height}(G)$ the height of $\mathrm{Tree}(G)$ and by $\mathrm{height}(A)$ the height of the parse tree with the root labelled by a nonterminal A.

6.2 AVL grammar

This section is essentially taken verbatim from [50].

AVL-grammars correspond to AVL-trees. We use the standard AVL-trees, for each node v the balance of v, denoted $\mathrm{balance}(v)$ is the difference between the height of the left and right subtrees of the subtree of T rooted at v. T is AVL-balanced i $|\,\mathrm{balance}(v)| \leq 1$ for each node v. We say that a grammar G is AVL-balanced if $\mathrm{Tree}(G)$ is AVL-balanced.

Lemma 26. *If the grammar G is AVL-balanced then $height(G) = O(\log n)$, where $n = |\,\mathrm{val}(G)|$.*

In each nonterminal A we keep also $\mathrm{balance}(A)$, no such information is stored in terminals (as it is not defined there).

Lemma 27. *Assume A, B are two nonterminals of AVL-balanced grammars. Then we can construct in $\mathcal{O}(|\,\mathrm{height}(A) - \mathrm{height}(B)|)$ time a AVL-balanced grammar G such that $\mathrm{val}(G) = \mathrm{val}(A) \cdot \mathrm{val}(B)$. This introduces $\mathcal{O}(|height(A) - height(B)|)$ new nonterminals.*

Proof. This is a simpler variant of standard AVL construction.

Note that the bound on the number of nonterminals follows from the bound on the running time.

Let $T_A = \mathrm{Tree}(A)$ and $T_B = \mathrm{Tree}(B)$. Assume that $height(T_A) \geq height(T_B)$, other case is symmetric. We follow the rightmost branch of T_A, the heights of nodes decrease each time at most by 2 in each step. We stop at a node v such that $\mathrm{height}(v) - \mathrm{height}(T_B) \in \{0, 1\}$. Create a new node v', attach it to father of v and attach to it v and root of T_B The resulting tree can be unbalanced (by at most 2) on the rightmost branch. Using standard AVL rotation going up we perform at most $\mathcal{O}(|height(A) - height(B))$ of them.

Note that the real parse-tree could be even of an exponential size, but we operate only on right-most path and consider only $\mathcal{O}(|height(A) - height(B))$ nodes on it, they can be recovered from the SLP in constant time per nonterminal. Only occurrences of nonterminals on the rightmost path (and their children) can be affected by the rotations. □

6.3 From LZ77 to AVL grammar

Suppose we have an LZ-factorization $w = f_1 f_2 \cdots f_k$. For each prefix $f_1 f_2 \cdots f_i$ we will construct an AVL balanced grammar of size $\mathcal{O}(i \log N)$.

Suppose that we have already constructed AVL-balanced for $f_1 f_2 \cdots f_{i-1}$. If f_i is a letter a than we set $G \leftarrow Cancat(G, a)$, which introduces $\mathcal{O}(\log N)$ new nonterminals. Otherwise we locate the segment corresponding to f_i in the $f_1 f_2 \cdots f_{i-1}$. As G is balanced we can find an $\mathcal{O}(\log N)$ nonterminals $S_1,\ S_2,\ \ldots S_{t(i)}$ of G such that

$f_i = \mathrm{val}(S_1)\,\mathrm{val}(S_2)\cdots\mathrm{val}(S_{t(i)})$. Denote by $u_1, u_2, \ldots$ the parents of those nodes. The sequence S_1, S_2, $\ldots S_t$ is a *grammar decomposition* of f_i. We concatenate the parts of the grammar corresponding to this nonterminals with G, using *Concat*.

Observe that S_1, S_2, $\ldots S_t$ can be divided into two parts: S_1, S_2, $\ldots S_{t_1}$ and S_{t_1+1}, S_{t_1+2}, $\ldots S_t$ such that heights of S_1, S_2, $\ldots S_{t_1}$ are non-decreasing and heights of S_{t_1+1}, S_{t_1+2}, $\ldots S_t$ are non-increasing.

By easy induction it can be shown that the grammar G_i for S_1, S_2, $\ldots$, S_i ($i \leq t_1$) has height at most $\mathrm{height}(u_i)$. The induction basis is G_1: the grammar for S_1, for which this is obvious (as S_1 is a child of u_1). Otherwise, when we concatenate G_{i-1} and S_i then by standard analysis of the AVL trees it follows that the height of the grammar is at most $\max(\mathrm{height}(G_{i-1}), \mathrm{height}(S_i) + 1$, and $\mathrm{height}(G_{i-1}) + 1 \leq \mathrm{height}(u_{i-1}) + 1 \leq \mathrm{height}(u_i)$ and $\mathrm{height}(S_i) + 1 \leq \mathrm{height}(u_i)$.

Similar analysis can be made also for the non-increasing part of the sequence.

Thus for each merge the number of rotations is at most $\mathrm{height}(S_{i+1} - \mathrm{height}(S_i))$ and this telescopically sums to $\mathcal{O}(\log N)$

The argument can be improved to $\mathcal{O}(\log(n/g))$ using appropriate modifications.

7 Generalising Recompression from word equations to SLPs

We employ the following naming conventions for SLPs: its nonterminals are ordered (without loss of generality: $X_1, X_2, \ldots, X_m$), each nonterminal has exactly one production and if X_j occurs in the production for X_i then $j < i$; we will use symbols $\mathcal{A}$, $\mathcal{B}$, etc. to denote an SLP.

We can treat SLP as a system of word equations (in variables $X_1, \ldots, X_m$): production $X_i \to \alpha_i$ corresponds to an equation $X_i = \alpha_i$; observe that such an equality is meaningful as $\mathrm{val}(X) = \mathrm{val}(\alpha)$ (where val is naturally extended to strings of letters and nonterminals), moreover, this is the unique solution of this equation. Thus the recompression technique can be applied to SLPs as well (so far we used recompression only to one equation but it easily generalises also to a system of equations). In particular, the space bound still applies, which shows the SLPs considered during the algorithm have polynomial size.

It remains to deal with the effectiveness: the recompression for word equations is highly non-deterministic, while algorithms for SLPs should, if possible, be deterministic (in fact we usually want them to be efficient, i.e. we want as small polynomial degree as possible).

Fortunately, it is easy to see that the non-determinism vanishes in case of SLPs—it is needed to:

1. establish, whether $\mathrm{val}(X_i) = \epsilon$;
2. establish the first (and last) letter of $\mathrm{val}(X_i)$;
3. establish the length of a-prefix and suffix of $\mathrm{val}(X_i)$.

Each of those questions can be easily answered, assuming that we already know the answers for X_j for $j < i$: let $X_i \to \alpha_i$, then we first remove from α_i all nonterminals X_j, for which $\mathrm{val}(X_j) = \epsilon$, and then

1. $\mathrm{val}(X_i) = \epsilon$ if and only if $\alpha_i = \epsilon$;

2. the first letter of $\mathrm{val}(X_i)$ is the first letter of α_i or the first letter of $\mathrm{val}(X_j)$, if the first symbol of α_i is X_j;

3. the length of the a-prefix depends only on the letters a in α_i and the lengths of a-prefixes in nonterminals in α_i.

All those conditions can be verified in linear time, thus the recompression for SLPs runs in polynomial (in SLP's size) time (so polynomial in total).

The first problem that we will consider in detail, is the equivalence of two SLPs, i.e. whether they define the same word.

7.1 Size

Our implementation will use parallel block compression (so for all letters in parallel) and pair compression for a all pairs from a partition.

To make the computation efficient, we run the algorithm in alternating phases:

- in the first it will use a partition so that the size of the encoded word is decreased by a constant fraction

- in second it will use a partition so that the number of letters in the SLP is decreased by a constant fraction.

In both cases the appropriate partition can be calculated in linear time.

In the former case this is an exercise, this requires that $N = |\mathrm{val}(\mathcal{X})|$ fits in $\mathcal{O}(1)$ memory cells. In the latter case the proof is exactly the same as in the case of word equations (as an SLP is a system of word equations).

This shows that

Lemma 28. *The number of phases of the algorithm is* $\mathcal{O}(\log N)$.

We would like to also show a linear bound on the size of the SLP, but to this end we need to know, how many letters are introduced into the SLP due to uncrossing.

Thus it remains ot show that each phase takes linear time.

7.2 Pair compression

We first uncross the partition, this can be done bottom-up; we give the code for one SLP, but it is run on both.

Lemma 29. *If* Σ_ℓ *and* Σ_r *are disjoint then after* $\mathsf{Pop}(\Sigma_\ell, \Sigma_r)$ *no pair in* $\Sigma_\ell\Sigma_r$ *is crossing. Furthermore,* $\mathrm{val}(\mathcal{X})$ *has not changed.*

Pop *runs in time* $\mathcal{O}(n+m)$ *and introduces at most* $4(n+m)$ *letters to* G.

Proof. Let β_i be the string popped from α_i to the left, i.e. a letter from Σ_r or ϵ and similarly γ_i the string popped to the right. As before, by X_i, α_i, etc. we denote the nonterminals, rules, etc. in the input and X'_i, α'_i, etc. the ones in the output. We use term i-th rule (and i-th string) to denote the contemporary value of the rule for X_i and the derived string. As a first step of the proof we show by a simple induction on i that:

1. When we have processed i-th rule then for $j > i$ the j-th string is equal to $\mathrm{val}(X_i)$, i.e. as in the input.

Algorithm 7 $\mathsf{Pop}(\Sigma_\ell, \Sigma_r)$: Popping letters from Σ_ℓ and Σ_r

```
1: for i ← 1 .. n − 1 do
2:     let X_i → α_i and b the first letter of α_i
3:     if b ∈ Σ_r then                                   ▷ Left-popping
4:         remove leading b from α_i
5:         replace X_i in G's rules by bX_i
6:         if α_i = ϵ then
7:             remove X_i from rules of G                ▷ X_i is empty
8:         else
9:             let a be the last letter of α_i
10:            if a ∈ Σ_ℓ then                           ▷ Right-popping
11:                remove ending a from α_i
12:                replace X_i in G's rules by X_i a
13:                if α_i = ϵ then
14:                    remove X_i from rules of G        ▷ X_i is empty
```

2. $\beta_i \neq \epsilon$ if and only if $\mathrm{val}(X_i)$ begins with a letter from Σ_r; additionally, this letter is β_i.

3. $\gamma_i \neq \epsilon$ if and only if $\mathrm{val}(X_i)$ ends with a letter from Σ_ℓ; additionally, this letter is γ_i.

4. $\mathrm{val}(X_i) = \beta_i \,\mathrm{val}(X_i')\gamma_i$.

To see 1 observe that if we remove β_i from the front of i-th rule, we also replace X_i with $\beta_i X_i$, the same applies to popping letters to the right. Also, X_i is removed from the rules if and only if i-th string is ϵ. To see 2 note that by induction assumption on 1 before considering i-th rule, the i-th string is $\mathrm{val}(X_i)$. So if i-th rule begins with a letter, it is the first letter of $\mathrm{val}(X_i)$ and we are done (as it is popped). The remaining case is that the i-th rule begins with a nonterminal, say X_j'. But this means that we did not pop a letter to the left from j-th rule when it was considered. So by 4 the $\mathrm{val}(X_j')$ and $\mathrm{val}(X_j)$ begin with the same letter, which is the same as the first letter of $\mathrm{val}(X_i)$, so it is in Σ_r, contradiction with 2 for X_j. The same analysis applies also to 3. Lastly, for 4 note that i-th string does not change when we consider $j \neq i$. So the only changes to i-th rule are done when the algorithm considers it and the claim is obvious.

Returning to the main claim, we want to show that it is impossible that after Pop there is a crossing pair, i.e. that one of (CP 1)–(CP 3) holds. Suppose for the sake of contradiction that (CP 1) holds, i.e. that for some $j < i$ the aX_j' occurs in the rule for X_i', where $\mathrm{val}(X_j')$ begins with $b \in \Sigma_r$. By 4 we know that $\beta_j \,\mathrm{val}(X_j')\gamma_j = \mathrm{val}(X_j)$. So $\mathrm{val}(X_j)$ begins with a letter from Σ_r: if β_j is a letter than it is from Σ_r by 1 and if not than we know that $\mathrm{val}(X_j')$ begins with a letter from Σ_r. Then by 2 we popped a letter from X_j and so β_j is a letter. But then the letter to the left of X_j' is this β_j, and we assume that it is from Σ_ℓ, contradiction, as $\Sigma_\ell \cap \Sigma_r = \emptyset$.

The other cases are shown in the same way.

Concerning the running time note that we do not need to read the whole G: it is enough to read the first and last letter in each rule. To perform the replacement, for each nonterminal X_i we keep a list of pointers to its occurrences, so that X_i can be replaced with aX_ib in $\mathcal{O}(1)$ time, and there are at most $2(n+m)$ occurrences of nonterminals in G.

Note that at most 2 letters are popped from each nonterminal and so at most $4(n+m)$ are introduced to G. □

After $\mathsf{Pop}(\Sigma_\ell, \Sigma_r)$ the pairs $ab \in \Sigma_\ell\Sigma_r$ are no longer crossing, we can compress them. Since such pairs do not overlap, this can be done in parallel in time $\mathcal{O}(|\mathcal{X}|)$, using RadixSort for grouping.

Concerning the block compression, we first uncross all blocks, in a bottom up way.

Algorithm 8 CutPrefSuff: removing crossing blocks.

```
for i ← 1..m+n, except m and n+m do
    let X_i → α_i be the production for X_i and a its first letter
    calculate and remove the a-prefix a^{ℓ_i} of α_i
    replace each X_i in rule's bodies by a^{ℓ_i} X_i
    if val(X_i) = ε then
        remove X_i from the rules' bodies
    else
        let b be the last letter of α_i
        calculate and remove the b-suffix b^{r_i} of α_i
        replace each X_i in rule's bodies by X_i b^{r_i}
        if val(X_i) = ε then
            remove X_i from the rules' bodies
```

Lemma 30. *After* CutPrefSuff *there are no crossing blocks. This algorithm and following block compression can be performed in time* $\mathcal{O}(|G| + (m+n)\log(m+n))$. *Together they introduce at most* 4 *new letters to each rule.*

Proof. We first show the first claim of the lemma, i.e. that after CutPrefSuff there are no letters with crossing blocks. This follows from the following observations:

1. When X_i is processed by CutPrefSuff, $\mathrm{val}(X_j)$ for $j \neq i$ is not changed.
2. When CutPrefSuff considers X_i with a rule $X_i \to \alpha_i$ such that $\mathrm{val}(X_i) = a^\ell w b^r$, where w does not start with a and does not end with b and $\ell, r > 0$, then α_i has an explicit a^ℓ prefix and explicit b^r suffix. If $\mathrm{val}(X_i) = a^\ell$ then $\alpha_i = a^\ell$.
3. When CutPrefSuff replaces X_i with $a^{\ell_i} X_i b^{r_i}$ then afterwards the only letter to the right of X_i in the rules is b, similarly, the only letter to the left is a.
4. After CutPrefSuff considered X_i, and X_i is to the right (left) of a then a is not the first (last, respectively) letter of $\mathrm{val}(X_i)$.

As in Lemma 29, all properties follow by a simple induction on the number i of the considered nonterminal.

We infer from these observations that after CutPrefSuff there are no crossing blocks in G. Suppose for the sake of contradiction, that there are; let a be the letter that has a crossing block. We consider only the case of (CP 1), i.e. when there is X_j such that aX_j occurs in some rule and $\mathrm{val}(X_j)$ begins with a. Note that by observation 2 when CutPrefSuff considered X_j then it replaced it with $b^\ell X_j c^r$ for some letters b and c. By observation 3 the letter to the left of X_j in the rule for X_i is not changed by CutPrefSuff

afterwards (except that it can be popped when considering some other nonterminal) hence $b = a$. Lastly, by observation 4 the first letter of $\mathrm{val}(X_j)$ is not $b = a$, contradiction.

CutPrefSuff is performed in $\mathcal{O}(|G|)$ time: we represent block a^ℓ as a pair (a, ℓ), then the length of the a-prefix (b-suffix) is calculated simply by reading the rule until a different letter is read (note that the lengths of the blocks fit in one machine word). Since there are at most 4 symbols introduced by CutPrefSuff to the rule, this takes $\mathcal{O}(|G| + n + m)$ time. The replacement of X_i by $a^{\ell_i} X_i b^{r_i}$ is done at most twice inside one rule and so takes in total $\mathcal{O}(n + m)$ time.

Note that right after CutPrefSuff it might be that there are neighbouring blocks of the same letter in the rules of G. However, we can easily replace such neighbouring blocks by one block of appropriate length in one reading of G, in time $\mathcal{O}(|G|)$.

Concerning the compression of the blocks of letters: we read the description of $\mathcal{X}$. Whenever we spot a maximal block a^ℓ for some letter a, we add a triple (a, ℓ, p) to the list, where p is the pointer to this occurrence of the block in $\mathcal{X}$. Notice, that as there are no crossing blocks, the nonterminals (and end or rules) count for termination of maximal blocks.

After reading the whole $\mathcal{X}$ we sort these pairs lexicographically, $\mathcal{O}(n \log n)$ time. Now, for a fixed letter a, we use the pointers to localise a's blocks in the rules and we replace each of its maximal block of length $\ell > 1$ by a fresh letter. Since the blocks of a are sorted according to their length, all blocks of the same length are consecutive on the list, and replacing them by the same letter is easily done.

Since we already know that there are no letters with crossing block, this procedure implements the block compression. □

Now we can infer the linear bound on the kept SLP.

Lemma 31. *During the recompression for SLPs, $|\mathcal{X}| = \mathcal{O}(n)$.*

The proof is straightforward: we show that the size of the words that were in a rule at the beginning of the phase shorten by a constant factor (in this phase). On the other hand, only Pop and CutPrefSuff introduce new letters to the rules and it can be estimated, that in total they introduced $\mathcal{O}(1)$ letters to a rule in each phase. Thus, bound $\mathcal{O}(1)$ on each rules' length holds.

THis yields a running time for checking the equivalence of SLPs of $\mathcal{O}(nb \log N \log n)$. In order to replace the running time by $\mathcal{O}(nb \log N)$ it is enough to ensure that compression of blocks runs in (amortised) linear time.

7.3 Faster block compression

Length representation The intuition is as follows: while the a blocks can have exponential length, most of them do not differ much, as they are obtained by concatenating letters a that occur explicitly in the grammar. Such concatenations can in total increase the lengths of a blocks by $|\mathcal{A}|$. Still, there are blocks of exponential length: these 'long' blocks are created only when two blocks coming from two different non-terminals are concatenated. However, there are only n concatenations of nonterminals (one per production), and so the total number of 'long' blocks 'should be' at most n. Of course, the two mentioned ways of obtaining blocks can mix, and our representation takes this into the account: we represent each block as a concatenation of two blocks: 'long' one and 'short' one:

- the 'long' corresponds to a block obtained as a concatenation of two nonterminals, such a long block is common for many blocks of letters;
- the 'short' one corresponds to concatenations of letters occurring explicitly in $\mathcal{A}$, this length is associated with the given block alone.

More formally: we store a list of *common lengths*, i.e. the lengths of common long blocks of letters. Each block-length ℓ is represented as a sum $c + o$, where c is one of the common lengths and o (*offset*) is a number associated with ℓ. Internally, ℓ is represented as a number o and a pointer to c. One of the common lengths is 0. In the intermediate construction some blocks can be represented without the common length, which is different than having a common length 0. The construction will guarantee that each offset is at most $|\mathcal{A}|$.

Creating a common length

During the blocks compression: each prefix and suffix popped from nonterminals inside the rule is represented as a common length and an offset in the following way (we add to them the common length 0 if they are represented without a common length). Then we calculate the lengths of blocks inside the rule for X_i: the explicit letter inside the rule simply increase the offset, the blocks that are formed solely from explicit letters do not have a common length. If any length of the block is represented as a sum of two common lengths (and perhaps some offset), we create a new common length for it.

Before proceeding, let us note on how large the offsets may be and how many of them are.

Lemma 32. *There are at most $n+1$ common lengths. There are at most $|\mathcal{X}| + n$ offsets in total and largest offset is at most $|\mathcal{X}|$.*

Proof. One common length is 0. Each other common length is created inside a rule for a nonterminal. Furthermore, at most one common length can be created inside a single rule: for a new common length c to be created there have to be two nonterminals X_j and X_k inside a rule before popping and the block a^c is created between those two nonterminals.

Creation of an offset corresponds to an explicit letter in $\mathcal{A}$, so there are at most $|\mathcal{A}|$ offsets created.

An offset is created or increased, when an explicit letter a (not in a compressed form) is concatenated to the block of as. One letter is used once for this purpose and there is no other way to increase an offset, so the maximal of them is at most $|\mathcal{A}|$. □

Comparing lengths

Since we intend to sort the lengths, we need to compare the lengths of two numbers represented as common lengths with offsets, say $o + c$ and $o' + c'$. Since the common lengths are so large, we expect that we can compare them lexicographically, i.e.

$$o + c \geq o' + c' \iff \begin{cases} c > c', & \text{or} \\ c = c' \wedge o \geq o' \ . \end{cases} \tag{2}$$

Furthermore (2) allows a simple way of sorting the lengths of maximal blocks:

- we first sort the common lengths (by their values)

- then for each common length we (separately) sort the offsets assigned to this common length.

While (2) need not to be initially true, we can apply a couple of patches that make it true. Before that however, we need the common lengths to be sorted. We sort them using RadixSort and treating each common length as a series of bits. Although this looks more expensive, it allows a nice amortised analysis as demonstrated later, see Lemma 36.

Lemma 33. *Let $c_1, c_2, \ldots, c_k$ be the common lengths. We can sort them in $\mathcal{O}(k + \sum_{i=1}^{k} \log(c_i))$.*

The sorting is done by a standard implementation of RadixSort that sorts the numbers of different lengths.

The problem with (2) is that even though c_i and c_j are so large, it can still happen that $|c_i - c_j|$ is small. We fix this naively: first we remove some common lengths so that $c_{i+1} - c_i > |\mathcal{A}|$. A simple greedy algorithm does the job in linear time. Since common lengths are removed, we need to change the representations of lengths: when o was assigned to removed c consider the c_i and c_{i+1} that remained in the sequence and $c_i < c < c_{i+1}$. We reassign $\ell = c + o$ to either c_i or c_{i+1}: if $o + c \geq c_{i+1}$ then we reassign it to c_{i+1} and otherwise to c_i. It can be shown that in this way all offsets are at most $2|\mathcal{A}|$ and that (2) holds afterwards.

Lemma 34. *Given a sorted list of common lengths $c_1 \leq c_2 \leq \cdots \leq c_k$ we can in $\mathcal{O}(\sum_{i=1}^{k} \log(c_i) + k)$ time choose its sublist and reassign offsets (preserving the represented lengths) such that all offsets are at most $2|\mathcal{A}|$ and (2) holds.*

Proof. We include $c_0 = 0$ for simplicity of presentation.

Firstly, we calculate the differences Δ between consecutive cs and store those that are at most $|\mathcal{A}|$. This can be done in $\mathcal{O}(\sum_{i=1}^{k} \log(c_i) + k)$ time: for two consecutive c_{i+1} and c_i we calculate their difference in time at most $\mathcal{O}(\log c_{i+1} + 1)$.

Given a sorted list $c_0 \leq c_1 \leq c_2 \leq \cdots \leq c_k$ of common lengths and their differences $\Delta_1, \Delta_i, \ldots$ we choose its subsequence $0 = c'_0 \leq c'_1 \leq c'_2 \leq \cdots \leq c'_{k'}$ such that the distance between any two consecutive common lengths in it is at least $|\mathcal{A}|$, i.e. $c'_{i+1} - c'_i \leq |G\mathcal{A}|$. This is done naively: we choose $c_0 = 0$ and then go through the list. Having last chosen c we look for the smallest common length c' such that $c' - c > |\mathcal{A}|$ and choose this c' as the next element. The condition $c' - c > |\mathcal{A}|$ is verified by summing up appropriate Δs. Since there are k common lengths in the beginning and adding defined Δs can be done in $\mathcal{O}(1)$ time, this can be done in $\mathcal{O}(k)$ time. As the last step, we also update the Δs, so that they still show the difference between consecutive common lengths.

For any removed c such that $c_i < c < c_{i+1}$ we reassign offsets assigned to c as described above: for o assigned to c, if $c + o \geq c_{i+1}$ (which can be equivalently stated as $c_{i+1} - c \leq o$) then we reassign o to c_{i+1}, otherwise to c_i. As $o \leq |\mathcal{A}|$, this condition can be verified using Δs alone.

When o is reassigned to c_i then $c - c_i \leq |\mathcal{A}|$ (as c was removed) and if to c_{i+1} then $c_{i+1} - c \leq o \leq |\mathcal{A}|$. Thus, in any case the new offset o' can be calculated using Δs and o, so in constant time. As there are $\mathcal{O}(|\mathcal{A}|)$ offsets, see Lemma 32, all reassigning takes $\mathcal{O}(|\mathcal{A}|)$ time in total.

Let o' be the offset after the reassignment. Then

- $o' \leq 2|\mathcal{A}|$: since $o \leq |\mathcal{A}|$ and the only way to increase it is to reassign it to c_i. Since c is removed, it holds that $c - c_i \leq |G|$. Hence $o' = o + (c - c_i) \leq |\mathcal{A}| + |\mathcal{A}|$.

- When o_i is assigned to c_i then $o_i + c_i < c_{i+1}$: indeed, if o_i was reassigned from $c > c_i$ then by definition $c_i + o_i = c + o < c_{i+1}$; if o_i was originally assigned to c_i or it was reassigned from c_{i-1} then $o_i < |G|$ and so $c_i + o_i \leq c_i + |\mathcal{A}| < c_{i+1}$.

Note that the second item above implies (2). Hence the claim of the Lemma holds. □

Now, since (2) holds, in order to sort all lengths it is enough to sort the offsets within groups. We do it simultaneously for all groups: offset o_j assigned to common length c_i is represented as (i, o_j), we sort these pairs lexicographically, using RadixSort. Since the offsets are at most $2|\mathcal{A}|$ and there are at most $|\mathcal{A}|$ of them and there are at most $\mathcal{O}(n)$ common lengths (see Lemma 32 for all those three estimations) RadixSort sorts them in $\mathcal{O}(|\mathcal{A}|)$ time.

Lemma 35. *When common lengths whose bitvectors are sorted and satisfy* (2)*, sorting all lengths of blocks takes* $\mathcal{O}(|\mathcal{A}|)$ *time.*

It is left to bound the time needed to identify and sort the common lengths. Due to Lemma 33 and Lemma 34 this cost is $\mathcal{O}(k + \sum_{i=1}^{k} \log(c_i))$, where $c_1, \ldots, c_k$ are all common lengths. Hence we can assign $\mathcal{O}(1 + \log(c))$ cost to a common length c and redirect this cost towards the rule, in which c was created. The $\mathcal{O}(1)$ cost for maintaining the common length 0 is redirected towards the rule for X_1. We estimate the total such cost over the whole run.

Lemma 36. *For a single rule, the cost redirected from common lengths towards this rule during the whole run is* $\mathcal{O}(\log N)$*.*

Proof. First of all, we can consider $\mathcal{O}(\log(c))$ instead of $\mathcal{O}(1 + \log(c))$: the 1 is added $\mathcal{O}(1)$ times per phase and there are $\mathcal{O}(\log N)$ many phases. Secondly, the cost associated with the common length 0 is $\mathcal{O}(1)$ per phase, so $\mathcal{O}(\log N)$ in total. As said, we associate it with the rule for X_1.

Each other common length $c > 0$ (of some a block) is created inside a rule for some X_i, let this rule be $X_i \to uX_jvX_kw$ (by definition two nonterminals in a rule are needed for a creation of a new common length). Then $\mathrm{val}(X_j)$ right-popped an a-suffix and $\mathrm{val}(X_k)$ left-popped an a-prefix, moreover, $v \in a^*$.

If the creation of the common length c removes a nonterminal from the rule for X_i then this happens at most twice for this rule and the associated cost is $\mathcal{O}(\log N)$.

So consider the case in which no nonterminal is removed from the rule during the creation of a new common length c of an a-block. Consider all such creations of powers in a rule for X_i. Let the consecutive letters, whose blocks were compressed, be $a^{(1)}$, $a^{(2)}$, $\ldots$, $a^{(\ell)}$ and the corresponding blocks' lengths c_1, c_2, $\ldots$, c_ℓ (the c_ℓ repetitions of $a^{(\ell)}$ are replaced by $a^{(\ell+1)}$). Observe, that $a^{(i+1)}$ does not need to be $a^{(i)}_{c_i}$, as there might have been some other compression in between.

Define *weight*: for a letter it is the length of the represented string in the original instance. Consider the weight of the strings between X_j and X_k. Clearly, after the i-th blocks compression it is exactly $c_i \cdot \mathrm{w}(a^{(i)})$. We show that

$$\mathrm{w}(a^{(i+1)}) \geq c_i \, \mathrm{w}(a^{(i)}) \ . \tag{3}$$

Right after the i-th blocks compression the string between X_j and X_k is simply $a^{(i)}_{c_i}$. After some operations, this string consists of c_{i+1} letters $a^{(i+1)}$. All operations do not remove the symbols from the string between two nonterminals in a rule (removing of

leading a_R or ending a_L from t cannot affect letters between nonterminals). Recall that we can think of the recompression as building of an SLP for p and t. In particular, one of the letters $a^{(i+1)}$ derives $a^{(i)}_{c_i}$ (which is the letter that replaced the block of c_i letters $a^{(i)}$). Since in the derivation the weight of the right and left hand sides are equal, it holds that

$$\mathrm{w}(a^{(i+1)}) \geq \mathrm{w}(a^{(i)}_{c_i}) = c_i \cdot \mathrm{w}(a^{(i)}) \ .$$

Thus $\mathrm{w}(a^{(\ell)}) \geq \prod_{i=1}^{\ell-1} c_i$. Recall that we consider only the cost of letters that occur in the pattern. Hence, $a^{(\ell)}$ (or some heavier letter) occurs in the pattern, and so $M \geq \mathrm{w}(a^{(\ell)})$ Hence,

$$\log(N) \geq \log\left(\prod_{i=1}^{\ell} c_i\right) = \sum_{i=1}^{\ell} \log c_i.$$

□

8 Approximation of the smallest SLP using recompression

We have already given a procedure that turns an explicitly given string into an SLP, see Algorithm 1, and proved during classes that it runs in linear time (with some assumptions on the computational model). We now show that it also yields a good approximation ratio.

To bound the cost of representation of letters introduced during the construction of the grammar, we start with the smallest grammar $\mathcal{G}$ generating (the input) t and then modify the grammar so that it generates t (i.e. the current string kept by texttoSLP) after each of the compression steps. Then the cost of representing the introduced letters is paid by various credits assigned to $\mathcal{G}$. Hence, instead of the actual representation cost, which is difficult to estimate, we calculate the total value of issued credit. Note that this is entirely a mental experiment for the purpose of the analysis, as $\mathcal{G}$ is not stored or even known to the algorithm. We just perform some changes on it depending on the texttoSLP actions.

Pair compression corresponds to a 'production'

$$c \to ab \tag{4a}$$

and similarly replacing a^ℓ with a_ℓ corresponds to a 'production'

$$a_\ell \to a^\ell \ . \tag{4b}$$

8.1 Intuition and road map

8.1.1 Modifying the grammar

The modification on $\mathcal{G}$ are the standard one: when we perform the block compression on the text, we (as a mental experiment) uncross all blocks and then perform the block compression on $\mathcal{G}$. Similarly, when we perform the pair compression (according to appropriate partition) we first uncross all pairs from this partition and then perform the pair compression (according to this partition).

The crucial part of the analysis is the modification of $\mathcal{G}$ according to the compression performed on t. The terms nonterminal, rules, etc. always address the optimal grammar

G (or its transformed version). To avoid confusion, we do not use terms 'production' and 'nonterminal' for a that replaced some substring in t (even though this is formally a nonterminal of the constructed grammar). Still, when a new 'letter' a is introduced to t we need to estimate the length of the 'productions' in the constructed grammar that are needed for a (note that we can of course use all letters previously used in t). The 'productions' introduced for a is called a *representation of a letter* a, the sum of lengths of those 'productions' is a *cost of representation of a letter* a (or simply: *representation cost*). in production (4a) then the representation cost is 2 (as we have only one rule $c \to ab$) and in a rule (4b) we have a cost ℓ; the latter cost can be significantly reduced, for instance for a^{12} we can have a representation cost of 8 instead of 12, when we use a subgrammar $a_2 \to aa$, $a_3 \to a_2 a$, $a_6 \to a_3 a_3$ and $a_{12} \to a_6 a_6$.

8.1.2 Block compression

So far we have not explained, how exactly we represent the letters a_ℓ.

Lemma 37. *Given a list $1 < \ell_1 < \ell_2 < \cdots < \ell_k$ we can represent letters $a_{\ell_1}, a_{\ell_2}, \ldots, a_{\ell_k}$ that replace blocks $a^{\ell_1}, a^{\ell_2}, \ldots, a^{\ell_k}$ with a cost*

$$\mathcal{O}\left(\sum_{i=1}^{k}[1 + \log(\ell_i - \ell_{i-1})]\right) \quad ,$$

where $\ell_0 = 0$.

Proof. Firstly observe that without loss of generality we may assume that the list $\ell_1, \ell_2, \ldots, \ell_k$ is given to us in a sorted way, as it can be easily obtained from the sorted list of occurrences of blocks. For simplicity define $\ell_0 = 0$ and let $\ell = \max_{i=1}^{k}(\ell_i - \ell_{i-1})$.

In the following, we shall define rules for certain new letters a_m, each of them 'derives' a^m (in other words, a_m represents a^m). For each $1 \le i \le \log \ell$ introduce a new letter a_{2^i}, defined as $a_{2^i} \to a_{2^{i-1}} a_{2^{i-1}}$, where a_1 simply denotes a. Clearly a_{2^i} represents a^{2^i} and the representation cost summed over all $i \le \ell$ is $2 \log \ell = \mathcal{O}(\log \ell)$.

Now introduce new letters $a_{\ell_i - \ell_{i-1}}$ for each $i > 0$, which shall represent $a^{\ell_i - \ell_{i-1}}$. They are represented using the binary expansion, i.e. by concatenation of at most $1 + \log(\ell_i - \ell_{i-1})$ from among the letters $a_1, a_2, a_4, \ldots, a_{2^{\lfloor \log(\ell_i - \ell_{i-1}) \rfloor}}$. This has a representation cost $\mathcal{O}(\sum_{i=1}^{k}[1 + \log(\ell_i - \ell_{i-1})])$.

Lastly, each a_{ℓ_i} is represented as $a_{\ell_i} \to a_{\ell_i - \ell_{i-1}} a_{\ell_{i-1}}$, which has a total representation cost $\mathcal{O}(k)$.

Summing up $\mathcal{O}(\log \ell)$, $\mathcal{O}(\sum_{i=1}^{k}[1 + \log(\ell_i - \ell_{i-1})])$ and $\mathcal{O}(k)$ we obtain $\mathcal{O}(\sum_{i=1}^{k}[1 + \log(\ell_i - \ell_{i-1})])$. □

8.1.3 Paying the representation cost: credit

With each explicit letter we associate two units of *credit* and pay most of the cost of representing the letters introduced during texttoSLP with these credits. More formally: when the algorithm modifies $\mathcal{G}$ and in the process it creates an occurrence of a letter, we *issue* (or pay) 2 new credits. On the other hand, if we do a compression step in G, then we remove some occurrences of letters. The credit associated with these occurrences is then *released* and can be used to pay for the representation cost of the new letters introduced by the compression step as well as for the credit for the newly introduced letters (so that the algorithm does not issue new credit). For pair compression the released credit indeed

suffices to pay both the credit of the new letters occurrences and their representation cost, but for chain compression the released credit does not suffice, as it is not enough to pay the representation cost. Here we need some extra amount that is estimated separately later on in Section 8.3. In the end, the total cost is the sum of credit that was issued during the modifications of G plus the value that we estimate separately in Section 8.3.

8.1.4 Additional cost

The additional cost of representing letters during the block compression is estimated separately. In most cases, the cost of creating blocks can be cover by released credit, the only exception is when two long blocks of a are joined together. This can happen only between nonterminals in some rule of G and then the additional cost is charged towards this rule. Then we show that one rule has only $\mathcal{O}(\log N)$ cost charged to it: if we charge $\sum_i \log \ell_i$ cost to a rule, then it originally derived a word of length at least $\prod_i \ell_i$. This is described in detail in Section 8.3.

8.2 Estimation of credit

We now analyse the credit during the pair compression and the amount of credit that is issued during uncrossing.

Lemma 38. *During pair compression (for non-crossing pairs) the new letters introduced to G and their representation costs are covered by the released credit.*

This is obvious: each occurrence of an explicit pair in the grammar has 4 units of credit, after the compression they have 2 units of credit and the 2 released credit is used for the representation cost.

Lemma 39. *Pair uncrossing introduces $2m$ credit, where m is the number of occurrences of nonterminals in G.*

We introduce at most two letters per nonterminal
The case of block compression is a little more delicate.

Lemma 40. *Uncrossing of blocks followed by blocks compression introduces $2m$ credit, where m is the number of occurrences of nonterminals in G.*

While we can pop many letters, all of them are compressed into a single one during the blocks compression.

Lemma 41. *The total amount of issued credit is $\mathcal{O}(g \log n)$.*

There is a linear amount of credit issued per phase and there are $\mathcal{O}(\log n)$ many phases.
This ends the analysis for pair compression.

8.3 Calculating the cost of representing letters in block compression

The issued credit is enough to pay the 2 credit for occurrences of letters introduced during texttoSLP and the released credit is enough to pay the credit of the letters introduced during the pair compression and their representation cost. However, credit alone cannot

cover the representation cost of letters replacing blocks. The appropriate analysis is presented in this section. The overall plan is as follows: firstly, we define a scheme of representing the letters based on the grammar G and the way G is changed by BlockComp (the *G-based representation*). Then for such a representation schema, we show that the cost of representation is $\mathcal{O}(g \log N)$. Lastly, it is proved that the actual cost of representing the letters by texttoSLP (the texttoSLP-*based representation*) is smaller than the G-based one, hence it is also $\mathcal{O}(g \log N)$.

8.3.1 G-based representation

The intuition is as follows: while the a blocks can have exponential length, most of them do not differ much, as in most cases the new blocks are obtained by concatenating letters a that occur explicitly in the grammar and in such a case the released credit can be used to pay for the representation cost. This does not apply when the new block is obtained by concatenating two different blocks of a (popped from nonterminals) inside a rule. However, this cannot happen too often: when blocks of length p_1, p_2, ..., p_ℓ are compressed (at the cost of $\mathcal{O}\Big(\sum_{i=1}^{\ell}(1+\log p_i)\Big) = \mathcal{O}(\log(\prod_{i=1}^{\ell} p_i))$, as each $p_i \geq 2$), the length of the corresponding text in the input text is $\prod_{i=1}^{\ell} p_i$, which is at most N. Thus $\mathcal{O}\Big(\sum_{i=1}^{\ell}(1+\log p_i)\Big) = \mathcal{O}(\log \prod_{i=1}^{\ell} p_i) = \mathcal{O}(\log N)$ cost per nonterminal is scored.

Getting back to the representation of letters: we create a new letter for each a block in the rule $X_i \to \alpha_i$ after uncrossing of blocks popped prefixes and suffixes from $X_1, \ldots, X_{i-1}$ but before it popped letters from X_i. (We add the artificial empty block ϵ to streamline the later description and analysis.) Such a block is a *power* if it is obtained by concatenation of two a-blocks popped from nonterminals inside a rule (and perhaps some other explicit letters a), note that this power may be then popped from a rule (as it may be a prefix or suffix in this rule). This implies that in the rule $X_i \to uX_jvX_kw$ the popped suffix of X_j and popped prefix of X_k are blocks of the same letter, say a, and furthermore $v \in a^*$. Note that it might be that one (or both) of X_j and X_k were removed in the process (in this case the power can be popped from a rule as well). For each block a^ℓ that is not a power we may uniquely identify another block a^k (perhaps ϵ, not necessarily a power) such that a^ℓ was obtained by concatenating $\ell - k$ explicit letters to a^k in some rule.

Lemma 42. *For each block a^ℓ represented in the G-based representation that is not a power there is block a^k (perhaps $k = 0$) such that a^k is also represented in G-based representation and a^ℓ was obtained in a rule by concatenating $\ell - k$ explicit letters that existed in the rule to a^k.*

Note that the block a^k is not necessarily unique: it might be that there are several a^ℓ blocks in G which are obtained as different concatenations of a^k and $\ell - k$ explicit letters.

Proof. Let a_ℓ be created in the rule for X_i, after popping prefixes and suffixes from $X_1, \ldots, X_{i-1}$. Consider, how many popped prefixes and suffixes take part in this a^ℓ.

If two, then it is a power, contradiction.

If one, then let the popped prefix (or suffix) be a^k. Since it was popped, say from X_j, then a^k was a maximal block in X_j before popping, so it is represented as well. Then in the rule for X_i the a^ℓ is obtained by concatenating $\ell - k$ letters a to a^k. None of those letters come from popped prefixes and suffixes, so they are all explicit letters that were present in this rule.

If there are none popped prefixes and suffixes that are part of this a^ℓ, then all its letters are explicit letters from the rule for X_i, and we treat it as a concatenation of k explicit letters to ϵ. □

We represent the blocks as follows:

1. for a block a^ℓ that is a power we represent a_ℓ using the binary expansion, which costs $\mathcal{O}(1 + \log \ell)$;
2. for a block a^ℓ that is obtained by concatenating $\ell - k$ explicit letters to a block a^k (see Lemma 42) we represent a_ℓ as $a_k \underbrace{a \cdots a}_{\ell-k \text{ times}}$, which has a representation cost of $\ell - k + 1$, this cost is covered by the $2(\ell - k) \geq \ell - k + 1$ credit released by the $\ell - k$ explicit letters a. Note that the credit released by those letters was not used for any other purpose. (Furthermore recall that the 2 units of credit per occurrence of a_ℓ in the rules of grammar are already covered by the credit issued by BlockComp.)

We refer to cost in 1 as the *cost of representing powers* and redirect this cost to the nonterminal in whose rule this power is created. The cost in 2, as marked there, is covered by released credit.

8.3.2 Cost of G-based representation

We now estimate the cost of representing powers. The idea is that if nonterminal X_i is charged the cost of representing powers of length $p_1, p_2, \ldots, p_\ell$, which have representation cost $\mathcal{O}(\sum_{i=1}^{\ell} 1 + \log p_i) = \mathcal{O}(\log(\prod_{i=1}^{\ell} p_i))$, then in the input this nonterminal generated a text of length at least $p_1 \cdot p_2 \cdots p_\ell \leq N$ and so the total cost of representing powers is $\mathcal{O}(\log N)$ (per nonterminal). This is formalised in the lemma below.

Lemma 43. *The total cost of representing powers by G-based representation charged towards a single rule is $\mathcal{O}(\log N)$.*

This is shown in the same way as in Lemma 36.

Corollary 44. *The cost of G-based representation is $\mathcal{O}(g + g \log N)$.*

Proof. Concerning the cost of representing powers, by Lemma 43 we redirect at most $\mathcal{O}(\log N)$ against each of the $m \leq g$ rules of G. The cost of representing non-powers is covered by the released credit; the initial value of credit is at most $2g$ and at most $\mathcal{O}(g \log N)$ credit is issued during the whole run of texttoSLP, which ends the proof. □

8.3.3 Comparing the G-based representation cost and texttoSLP-based representation cost

We now show that the cost of texttoSLP-based representation is at most as high as G-based one. We first represent G-based representation cost using a weighted graph $\mathcal{G}_G$, such that the G-based representation is (up to a constant factor) $w(\mathcal{G}_G)$, i.e. the sum of weights of edges of $\mathcal{G}_G$.

Lemma 45. *The cost of G-based representation of all blocks is $\Theta(w(\mathcal{G}_G))$, where nodes of $\mathcal{G}_G$ are labelled with blocks represented in the G-based representation and edge from a^ℓ to a^k, where $\ell > k$, has weight $\ell - k$ or $1 + \log(\ell - k)$ (in this case additionally $k = 0$). Each node (other than a and ϵ) has at least one outgoing edge.*

The former corresponds to the representation cost covered by the released credit while the latter to the cost of representing powers.

Proof. We give a construction of the graph $\mathcal{G}_G$.

Fix the letter a and consider any of the blocks a^ℓ that is represented by G, we put a node a^ℓ in $\mathcal{G}_G$. Note that a single a^ℓ may be represented in many ways: different occurrences of a^ℓ are replaced with a_ℓ and may be represented in different ways (or even twice in the same way), this means that $\mathcal{G}_G$ may have more than one outgoing edge per node.

- when a^ℓ is a power, we create an edge from the node labelled with a^ℓ to ϵ, the weight is $1 + \log \ell$ (recall that this is the cost of representing this power);
- when a_ℓ is represented as a concatenation of $\ell - k$ letters to a_k, we create and edge from the node a^ℓ to a^k, the weight is $\ell - k$ (this is the cost of representing this block; it was paid by the credit on the $\ell - k$ explicit letters a).

Then the sum of the weight of the created graph is a cost of representing the blocks using the G-based representation (up to a constant factor). □

Similarly, the cost of texttoSLP-based representation has a graph representation $\mathcal{G}_{\mathsf{texttoSLP}}$.

Lemma 46. *The cost of* texttoSLP*-representation for blocks of a letter a is $\Theta(w(\mathcal{G}_{\mathsf{texttoSLP}}))$, where the nodes of $\mathcal{G}_{\mathsf{texttoSLP}}$ are labelled with blocks represented by* texttoSLP*-representation and it has an edge from a^ℓ to a^k if and only if ℓ and k are two consecutive lengths of a-blocks. Such an edge has weight $1 + \log(\ell - k)$.*

Proof. Observe that this is a straightforward consequence of the way the blocks are represented: Lemma 37 guarantees that when blocks $a^{\ell_1}, a^{\ell_2}, \ldots, a^{\ell_k}$ (where $1 < \ell_1 < \ell_2 < \cdots < \ell_k$) are represented the texttoSLP-representation cost is $\mathcal{O}(\sum_{i=1}^{k}[1 + \log(\ell_i - \ell_{i-1})])$, so we can assign cost $1 + \log(\ell_i - \ell_{i-1})$ to a^{ℓ_i} (and make it the weight on the edge to the previous block). □

We now show that $\mathcal{G}_G$ can be transformed to $\mathcal{G}_{\mathsf{texttoSLP}}$ without increasing the sum of weights of the edges. This is done by simple redirection of edges and changing their cost.

Lemma 47. *$\mathcal{G}_G$ can be transformed to $\mathcal{G}_{\mathsf{texttoSLP}}$ without increasing the sum of weights of the edges.*

Proof. Fix a letter a, we show how to transform the subgraph of $\mathcal{G}_G$ induced by nodes labelled with blocks of a to the corresponding subgraph of $\mathcal{G}_{\mathsf{texttoSLP}}$, without increasing the sum of weights.

Firstly, let us sort the nodes according to the increasing length of the blocks. For each node a^ℓ, if it has many edges, we delete all except one and then we redirect this edge to a^ℓ's direct predecessor (say a^k) and label it with a cost $1 + \log(\ell - k)$. This cannot increase the sum of weights of edges:

- deleting does not increase the sum of weights;
- if a_ℓ has an edge to ϵ with weight $1 + \log \ell$ then $1 + \log \ell \geq 1 + \log(\ell - k)$;
- otherwise it had an edge to some $k' \leq k$ with a weight $\ell - k'$. Then $1 + \log(\ell - k) \leq \ell - k \leq \ell - k'$, as claimed (note that $1 + \log x \leq x$ for $x \geq 1$).

Some blocks labelling nodes in $\mathcal{G}_G$ perhaps do not label the nodes in $\mathcal{G}_{\mathsf{texttoSLP}}$. For such a block a^ℓ we remove its node a_ℓ and redirect its unique incoming edge to its predecessor, say $a_{\ell'}$, changing the weight appropriately. Since $1 + \log(x) + 1 + \log(y) \geq 1 + \log(x + y)$ when $x, y \geq 1$, we do not increase the total weight.

It is left to observe that if a node labelled with a^ℓ exists in $\mathcal{G}_{\mathsf{texttoSLP}}$ then it also exists in $\mathcal{G}_G$, i.e. all blocks represented in texttoSLP occur in t. After uncrossing of blocks there are no crossing blocks. So any maximal block in t (i.e. one represented by texttoSLP-based representation) is also a maximal block a^ℓ in some rule (after uncrossing of blocks), say in X_i. But then this block is present in X_i also just before uncrossing of blocks on X_i and so it is represented by G-based representation.

In this way we obtained a graph corresponding to the texttoSLP-based representation. □

Corollary 48. *The total cost of* texttoSLP*-representation is* $\mathcal{O}(g \log n)$.

Proof. By Lemma 47 it is enough to show this for the G-based representation, which holds by Corollary 44 □

9 Equality testing: similar approach

An approach that somehow paved the way to recompression was introduced by Mehlhorn, Sundar, and Uhrig [41] in their work on data structures for equality testing of dynamic strings.

In this setting, we want to create a data structure that allows the following operations.

Makesequence(s, a) Creates a sequence consisting of one letter a

Equal(s_1, s_2) tests the equality of strings s_1 and s_2

Concatenate(s_1, s_2) creates a new sequence, the concatenation of s_1 and s_2, and inserts it into the structure;

Split(s, i) Splits the sequence s at position i and inserts two resulting sequences into the data structure

All operations preserve previous strings, i.e. Concatenate and Split do not remove the original sequences from the data structure (so the operations are persistent).

Note that the SLP equivalence can be easily tested using such a data structure; in fact, this works for equivalence of composition systems.

Using their data structure (with a modified approach by Alstrup, Brodal, and Rauhe [2]) we obtain the following running times

Theorem 49. *There exists an implementation of the data structure that supports the above operations in times:*

(Where: n is the length of the strings on which we operate, m is the number of so-far performed operations, N is the bound on the size of the numbers on which we operate)

Makesequence $\mathcal{O}(\log \min(N, m))$

Equal $\mathcal{O}(1)$

Concatenate, Split $\mathcal{O}(\log n \log m \log^* N)$

With appropriate implementation, this data structure also supports the calculation of the LCP of two given strings in $\mathcal{O}(\log n)$ time. This is not covered in the lecture.

The main idea is to create a *signature* for each string in the data structure. The signatures are build in phases, and the i-th signature is used to produce the $i+1$st. Thus we can see the whole process as building an SLP for the sequences, with the additional assumption, that we want the SLPs to be equal for the same strings, even they are obtained in different way.

The signature is built using two alternating operations:

- block encoding: the first one replaces each block a^ℓ with (a, ℓ) (treated as one symbol)
- the second groups the letters into segments of length between 2 and 4 and then replaces the segments with new symbols

In this way signature building can be seen as iterative deterministic hashing. The important property is that a signatures are different for different texts.

Another property is the locality whether a letter begins or ends a fragments depends only on $\mathcal{O}(\log^* N)$ neighbours. In this way the update algorithms for concatenate and split need to perform only local changes on each of $\mathcal{O}(\log n)$ levels. Furthermore, they have to handle $\mathcal{O}(\log *N)$ elements on each level.

This is based on the following marking algorithm

Lemma 50. *For any string of numbers (whose two consecutive elements are different) with values in* $\{0, \ldots, N\}$ *there is a function that assigns to each element* 0 *or* 1 *such that*

- *the assignment depends only on* $\Delta = \log^* N + 11$ *neighbouring elements in the sequence;*
- *no two consecutive elements are assigned* 1
- *there is at least one* 1 *assigned to each three consecutive elements.*

Clearly such an assignment can be used for denoting fragments: we end each fragment at first 1.

9.1 How to calculate assignment

This is based on [6, 18].

Informally it is done as follows. We first compute a valid $\log N$-coloring. Afterwards we replace the elements in the list by their colors, consider the set of colors to be the new universe, and iterate the coloring procedure. After $\mathcal{O}(\log^* N)$ iterations we get a valid six-coloring which we then reduce by a different procedure to a three-coloring which is then used to generate the assignment.

Identify each a_i (and its color) with its binary representation (which has $\mathcal{O}(\log N)$ bits). The bits are numbered from zero and the 0-th element is always assigned 0. In each iteration every element a_i is assigned a new color by concatenating the number of the bit, where the old color of a_{i-1} and a_i differ and the value of this bit. (For the a_0 we always assign 0.)

Lemma 51. *This procedure produces a valid* 6*-coloring and has* $\mathcal{O}(\log^* N)$ *many iterations.*

It is easy to check that indeed this produces a valid coloring and that finally we end up with a 6-coloring. (Exercise)

For the number of phases note that in each phase colors are reduced from k to $2\log k+1$, so the process terminates after $\mathcal{O}(\log^* n)$ many phases.

In particular, the final colour of the node depends only on its $\mathcal{O}(\log^* N)$ many neighbours.

To go from 6 coloring to 3 coloring we replace each color $3, 4, 5$ by the smallest among $0, 1, 2$ that is not assigned to its neighbours. Then we make the assignment of 0 and 1 by assigning 1 iff the color is a local maximum. It is easy to check that it has the desired properties.

9.2 Storing

The signatures are stored in an SLP-like structure. Conceptually, for a string s we store its signature and treat each letter in the signature as a nonterminal of an SLP, in particular, we put appropriate rule for generation of the text. We then store higher and higher signatures, until a single symbol is obtained.

Comparison of two texts is done by comparing the top symbols of their signatures (and the height of the signatures). For convenience, for each nonterminal of the signature we store also the length of the represented text.

9.3 Update

String is easy. We consider the Split, Concatenation is done similarly.

To split a signature, we go in the SLP to appropriate position, we store $\mathcal{O}(\log^* N)$ elements from each side of the path on each level, when this is the assignement, or the length of the a-prefix/suffix, when the level calculates the blocks compression. Since the signatures are computed locally, this is enough to recalculate the signature. (Exercise)

We need some additional structure (say, a dictionary), to search for existing signatures. One such operation takes $\mathcal{O}(\log m)$ time, as there are at most $m \log^* N$ signatures on each level (exercise).

9.4 Comments

This can be improved in a non-trivial way to pattern searching. Furthermore, we can ensure that we store the texts in alphabetic order and can compute the LCP for two elements in the sequence in $\mathcal{O}(\log n)$ time.

10 Compressed pattern matching: Combinatorial approach

In this section, the *position* is between two consecutive letters in a word, a cut in a rule $A \to BC$ is a position corresponding to the end of $\mathrm{val}(B)$ and beginning of $\mathrm{val}(C)$. Overlapping a position/cut is defined as earlier.

The presented algorithm is in fact a little stronger: we will compute occurrences of a pattern given by an SLP P within the text given by the SLP P. This is based on [36].

The nonterminals of the $\mathcal{P}$ and $\mathcal{T}$ are $P_1, \ldots, P_m$ and $T_1, \ldots, T_n$. The size of the problem is $n + m$. The lengths of $\mathrm{val}(\mathcal{P})$ and $\mathrm{val}(\mathcal{T})$ are M, N respectively.

10.1 AP table

Lemma 52 (Basic Lemma)**.** *All occurrences of* $\mathrm{val}(\mathcal{P})$ *in* $\mathrm{val}(\mathcal{T})$ *overlapping any given position form a single arithmetical progression.*

Proof. Exercise. □

The AP-table (table of arithmetical progressions) is defined as follows: $AP[i, j]$, where $1 \leq i \leq m, 1 \leq j \leq n$, encodes the arithmetic progression of occurrences of $\mathrm{val}(P_i)$ in $\mathrm{val}(T_j)$ that overlap the cut of $\mathrm{val}(T_j)$. Such encoding uses three numbers: the starting position (with respect to the beginning of $\mathrm{val}(T_j)$), the step of the arithmetic progression and the number of elements in this arithmetic progression. Note that this arithmetic progression can be empty, in which case we represent it appropriately.

Not that AP can be used to calculate all occurrences of $\mathrm{val}(\mathcal{P})$ in $\mathrm{val}(\mathcal{T})$: fix such an occurrence. By going down the derivation tree we see that we will find T_j such $\mathrm{val}(\mathcal{P})$ occurs in $\mathrm{val}(T_j)$ overlapping a cut. Moreover, for a fixed occurrence this can happen for at most three different T_js, which can be easily identified: this happens only when in a rule $T_j \to T_{j'}T_{j''}$ the occurrence overlaps a cut but is wholly within one of $T_{j'}, T_{j''}$.

Filling $AP[1, j]$ and $AP[i, 1]$ is easy. Then we fill them in lexicographic order on $[i, j]$.

Let $P_i = P_r P_s$, we consider the case, when $|\mathrm{val}(P_r)| \geq \mathrm{val}(P_s)$, the other one is symmetric. Let γ be the cut between P_r and P_s in P_i.

We shall use a *local search procedure* $LSP(i, j, [\alpha \,..\, \beta])$, which gives the positions of occurrences of P_i in T_j that are fully contained in $\mathrm{val}(T_j)[\alpha \,..\, \beta]$.

- It can use $AP[i \,..\, k]$ for $k \leq j$.
- Assumes that $|\beta - \alpha| \leq 3|\mathrm{val}(P_i)|$
- Runs in time $\mathcal{O}(j)$
- gives at most two arithmetic progressions as an output, all posistions in one are stictly before positions in the other. Both claims require some proof.

We shall use $\mathcal{O}(1)$ local searches.

10.1.1 Filling AP using LSP

We first find occurrences of P_r (which is a bigger part), to this end we use $LSP(r, j, [\gamma - |\mathrm{val}(P_i)|, \gamma + |\mathrm{val}(P_r)|)$. Note that as $|\mathrm{val}(P_r)| \geq |\mathrm{val}(P_s)|$ we have that $|\mathrm{val}(P_i)| \leq 2|\mathrm{val}(P_r)|$, so the assumption of LSP is satisfied.

We shall now look for occurrences of P_s that extend those of P_r. As those are given by at most two arithmetic progressions, we focus on one only.

We look at endings of occurrences of P_r. They are *continental*, if they end at most $|P_s|$ from the last ending in this arithmetic progression and *seaside* otherwise.

For the continental endings note that the corresponding occurrences of P_s are all within shifted occurrences of P_r, so due to periodicity either all continental occurrences of P_r

extend by P_s or none. Thus we check one, using one local search. Note that this can be done easier, but we do not care.

For the seaside endings, let δ be the last ending of the P_r in the sequence. Then we can use the $LSP(s, j, [\delta - |\operatorname{val}(P_s)|, \delta + |\operatorname{val}(P_s)|])$. Thus we obtain 2 arithmetic progressions representing the occurrences of P_s. We can intersect them with the arithmetic progression representing endings of P_r in constant time (exercise), which is again an arithmetic progression.

As a last step we merge the obtained arithmetic progressions. Note that we know that they form an arithmetic progression by Lemma 52.

10.2 Local search procedure

We proceed in almost naive manner. For $LSP(i, j, [\alpha \mathinner{..} \beta])$ If $|\alpha - \beta| < |\operatorname{val}(P_i)|$, then we return empty set. then we look at $AP[i, j]$ and intersect the obtained arithmetic progression with $[\alpha \mathinner{..} \beta]$. Let $T_j = T_r T_s$, then we make the recursive calls, making appropriate offsets; note that we simply store the list of obtained arithmetic progressions, offsetted to the original positions.

It is easy to check, that the total recursion time is $\mathcal{O}(j)$ (exercise).

Lastly, we merge the resulting arithmetic progressions. It is easy to check that two such arithmetic progressions either are disjoint or have at most the first/last element in common. Thus we can merge them in constant time per item. The bound on two arithmetic progression follows from Lemma 52 (exercise).

11 Quadratic word equations

Since in general the satisfiability of word equations is NP-hard, it is natural to try to find a smaller subclass of this problem, which is decidable in P. Limiting the number of possible variables or the number of their occurrences are such candidates. In case of quadratic equations it is easy to give a (non-deterministic) linear-space algorithm, which preceded a general PSPACE algorithm.

Take any equation, it is of the form $x \cdots = y \cdots$ or $x \cdots = a \cdots$ (otherwise we can delete the same symbols from the equation or reject altogether). We make a nondeterministic guess: if $S(x) = \epsilon$, then we remove this variable. If not, then we by symmetry we may assume that $S(x) = a \cdots$ or $S(x) = S(y) \cdots$. Thus we make a substitution $x \leftarrow ax$ or $x \leftarrow yx$. In both cases we again reduce the equation. It is easy to see that the lengths of the equations did not increase since it is a quadratic equation, we introduced at most two new symbols, but at the same time removed exactly two due to reduction.

This procedure can be easily written down as a graph with nodes labelled with possible (systems of equations) and edges between them representing the possible steps.

It remains unknown, whether quadratic equations are in NP. This is known in case of equations in free groups, but the argument is heavy in terms of geometry, so it will not be presented here.

12 Word equations: limited number of variables

As of today, the case of word equations with 3 variables remains unknown: it is not known to be NP-hard, nor it is known to be within NP. (It is known to be within NP in some

restricted cases [51]).

On the other hand, it was shown by Charatonik and Pacholski [3] that indeed, when only two variables are allowed (though with arbitrarily many occurrences), the satisfiability can be verified in deterministic polynomial time. The degree of the polynomial was very high, though. This was improved over the years and the best known algorithm is by Dąbrowski and Plandowski [13] and it runs in $\mathcal{O}(n^5)$ and returns a description of all solutions.

12.0.1 One variable

Clearly, the case of equations with only one variable is in P. Constructing a cubic algorithm is almost trivial, small improvements are needed to guarantee a quadratic running time. First non-trivial bound was given by Obono, Goralcik and Maksimenko, who devised an $\mathcal{O}(n \log n)$ algorithm [44]. This was improved by Dąbrowksi and Plandowski [14] to $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of occurrences of the variable in the equation. Furthermore they showed that there are at most $\mathcal{O}(\log n)$ distinct solutions and at most one infinite family of solutions. Intuitively, the $\mathcal{O}(\#_X \log n)$ summand in the running time comes from the time needed to find and test these $\mathcal{O}(\log n)$ solutions.

This work was not completely model-independent, as it assumed that the alphabet Σ is finite or that it can be identified with numbers. A more general solution was presented by Laine and Plandowski [27], who improved the bound on the number of solutions to $\mathcal{O}(\log \#_X)$ (plus the infinite family) and gave an $\mathcal{O}(n \log \#_X)$ algorithm that runs in a pointer machine model (i.e. letters can be only compared and no arithmetical operations on them are allowed); roughly one candidate for the solution is found and tested in linear time. Note that there is a conjecture that one variable word equations have $\mathcal{O}(1)$ solutions (plus the infinite family), in fact, an equation with 3 solutions outside the infinite family is not known.

We present a specialisation of the recompression algorithm for word equation for the one-variable case and show that it has the running time $\mathcal{O}(n \log \#_X)$. This running time can be improved to linear, at the expense of heavy usage of stringology data structures and combinatorial analysis.

12.1 One-variable equations

Without loss of generality in a word equation $\mathcal{A} = \mathcal{B}$ one of $\mathcal{A}$ and $\mathcal{B}$ begins with a variable and the other with a letter:

- if they both begins with the same symbol (be it letter or nonterminal), we can remove this symbol from them, without affecting the set of solutions;
- if they begin with different letters, this equation clearly has no solution.

The same applies to the last symbols of $\mathcal{A}$ and $\mathcal{B}$. Thus, in the following we assume that the equation is of the form

$$A_0 X A_1 \dots A_{n_\mathcal{A}-1} X A_{n_\mathcal{A}} = X B_1 \dots B_{n_\mathcal{B}-1} X B_{n_\mathcal{B}} \ , \tag{5}$$

where $A_i, B_i \in \Sigma^*$ and $n_\mathcal{A}$ ($n_\mathcal{B}$) denote the number of X occurrences in $\mathcal{A}$ ($\mathcal{B}$, respectively). Note that exactly one of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$ is empty and A_0 is non-empty. If this condition is violated for any reason, we greedily repair it by cutting identical letters (or variables)

from both sides of the equation. We say that A_0 is the *first* word of the equation and the non-empty of $A_{n_\mathcal{A}}$ and $B_{n_\mathcal{B}}$ is the *last word*. We additionally assume that none of words A_i, B_j is empty. We later (after Lemma 4) justify why this is indeed without loss of generality.

Note that if $S(X) \neq \epsilon$, then using (5) we can always determine the first (a) and last (b) letter of $S(X)$ in $\mathcal{O}(1)$ time. In fact, we can determine the length of the a-prefix and b-suffix of $S(X)$.

Lemma 53. *For every solution S of a word equation such that $S(X) \neq \epsilon$ the first letter of $S(X)$ is the first letter of A_0 and the last the last letter of $A_{n_\mathcal{A}}$ or $B_{n_\mathcal{B}}$ (whichever is non-empty).*

If $A_0 \in a^+$ then $S(X) \in a^+$ for each solution S of $\mathcal{A} = \mathcal{B}$.

If the first letter of A_0 is a and $A_0 \notin a^+$ then there is at most one solution $S(X) \in a^+$, existence of such a solution can be tested (and its length returned) in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. Furthermore, for $S(X) \notin a^+$ the lengths of the a-prefixes of $S(X)$ and A_0 are the same.

Two comments are in place:

- Symmetric version of Lemma 53 holds for the suffix of $S(X)$.
- It is later shown that finding all solutions from a^+ can be done in linear time, see Lemma 61.

A simple proof is left as an exercise.

By TestSimpleSolution(a) we denote a procedure, described in Lemma 53, that for $A_0 \notin a^*$ establishes the unique possible solution $S(X) = a^\ell$, tests it and returns ℓ if this indeed is a solution.

12.2 Representation of solutions

Consider any solution S of $\mathcal{A} = \mathcal{B}$. We claim that $S(X)$ is uniquely determined by its length and so when describing solution of $\mathcal{A} = \mathcal{B}$ it is enough to give their lengths.

Lemma 54. *Each solution S of equation of the form (5) is of the form $S(X) = (A_0)^k A$, where A is a prefix of A_0 and $k \geq 0$. In particular, it is uniquely defined by its length.*

Proof. If $|S(X)| \leq |A_0|$ then $S(X)$ is a prefix of A_0. When $|S(X)| > |A_0|$ then $S(\mathcal{A})$ begins with $A_0 S(X)$ while $S(\mathcal{B})$ begins with $S(X)$ and thus $S(X)$ has a period A_0. Consequently, it is of the form $A_0^k A$, where A is a prefix of A_0. □

Weight

Each letter in the current instance of our algorithm OneVarWordEq represents some string (in a compressed form) of the input equation, we store its *weight* which is the length of such a string. Furthermore, when we replace X with $a^\ell X$ (or $X a^\ell$) we keep track of the sum of weights of all letters removed so far from X. In this way, for each solution of the current equation we know what is the length of the corresponding solution of the original equation (it is the sum of weights of letters removed so far from X and the weight of the current solution). Therefore, in the following, we will not explain how we recreate the solutions of the original equation from the solution of the current one. Concerning the running time needed to calculate the length of the original solution: our algorithm OneVarWordEq reports only solutions of the form a^ℓ, so we just need to multiply ℓ with the weight of a and add the weights of the removed suffix and prefix.

12.2.1 Preserving solutions

All subprocedures of the presented algorithm should preserve solutions, i.e. there should be a one-to-one correspondence between solution before and after the application of the subprocedure. However, when we replace X with $a^\ell X$ (or Xb^r), some solutions may be lost in the process and so they should be reported. We formalise these notions.

Definition 55 (Preserving solutions)**.** A subprocedure *preserves solutions* when given an equation $\mathcal{A} = \mathcal{B}$ it returns $\mathcal{A}' = \mathcal{B}'$ such that for some strings u and v (calculated by the subprocedure)

- some solutions of $\mathcal{A} = \mathcal{B}$ are reported by the subprocedure;
- for each unreported solution S of $\mathcal{A} = \mathcal{B}$ there is a solution S' of $\mathcal{A}' = \mathcal{B}'$, where $S(X) = uS'(X)v$ and $S(\mathcal{A}) = uS'(\mathcal{A}')v$;
- for each solution S' of $\mathcal{A}' = \mathcal{B}'$ the $S(X) = uS'(X)v$ is an unreported solution of $\mathcal{A} = \mathcal{B}$ and additionally $S(\mathcal{A}) = uS'(\mathcal{A}')v$.

The intuitive meaning of these conditions is that during transformation of the equation, either we report a solution or the new equation has a corresponding solution (and no new 'extra' solutions).

By $\mathrm{h}_{c\to ab}(w)$ we denote the string obtained from w by replacing each c by ab, which corresponds to the inverse of pair compression. We say that a subprocedure *implements pair compression* for ab, if it satisfies the conditions from Definition 55, but with $S(X) = u\,\mathrm{h}_{c\to ab}(S'(X))v$ and $S(\mathcal{A}) = u\,\mathrm{h}_{c\to ab}(S'(\mathcal{A}'))v$ replacing $S(X) = uS'(X)v$ and $S(\mathcal{A}) = uS'(\mathcal{A}')v$.

Similarly, by $\mathrm{h}_{\{a_\ell\to a^\ell\}_{\ell>1}}(w)$ we denote the string w with letters a_ℓ replaced with blocks a^ℓ, for each $\ell > 1$; note that this requires that we know, which letters 'are' a_ℓ and what is the value of ℓ, but this is always clear from the context. A notion of *implementing blocks compression* for a letter a is defined similarly as the notion of implementing pair compression. The intuitive meaning of both those notions is the same as in case of preserving solutions: we not loose, nor gain any solutions.

12.3 Specialisation of precodures

We now specialise the general algorithms to our specific setting. Pair compression and block compression work exactly as before. However, during popping we need to additionally verify some solutions, which may be lost.

Algorithm 9 $\mathsf{Pop}(a, b)$

1: **if** b is the first letter of $S(X)$ **then**
2: **if** $\mathsf{TestSimpleSolution}(b)$ returns 1 **then** ▷ $S(X) = b$ is a solution
3: report solution $S(X) = b$
4: replace each X in $\mathcal{A} = \mathcal{B}$ by bX
▷ Implicitly change $S(X) = bw$ to $S(X) = w$
5: **if** a is the last letter of $S(X)$ **then**
6: **if** $\mathsf{TestSimpleSolution}(a)$ returns 1 **then** ▷ $S(X) = a$ is a solution
7: report solution $S(X) = a$
8: replace each X in $\mathcal{A} = \mathcal{B}$ by Xa
▷ Implicitly change $S(X) = w'a$ to $S(X) = w'$

Lemma 56. $\mathsf{Pop}(a, b)$ *preserves solutions and after its application the pair* ab *is non-crossing.*

The only new part is the preservation of solutions. But this easily follows from Lemma 53.

Thus first uncrossing a pair ab and then compressing it as a noncrossing pair implements the pair compression.

There is one issue: the number of non-crossing pairs can be large, however, a simple preprocessing, which basically applies Pop, is enough to reduce the number of crossing pairs to 2.

Algorithm 10 PreProc Ensures that there are at most 2 crossing pairs

1: let a, b be the first and last letter of $S(X)$
2: run $\mathsf{Pop}(a, b)$

Lemma 57. PreProc *preserves solution and after its application there are at most two crossing pairs.*

Proof. It is enough to show that there are at most 2 crossing pairs, as the rest follows from Lemma 4. Let a and b be the first and last letters of $S(X)$, and a', b' such letters after the application of PreProc. Then each X is preceded with a and succeeded with b in $\mathcal{A}' = \mathcal{B}'$. So the only crossing pairs are aa' and $b'b$ (note that this might be the same pair or part of a letter-block, i.e. $a = a'$ or $b = b'$). □

Note that in order to claim that the lengths of a-prefix of $S(X)$ and A_0 are the same, see Lemma 53, we need to assume that $S(X)$ is a not block of letters. This is fine though, as this condition holds when we apply Algorithm 11.

Algorithm 11 CutPrefSuff Cutting prefixes and suffixes; assumes that A_0 is not a block of letters

Require: A_0 is not a block of letters, the non-empty of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$ is not a block of letters

1: let a be the first letter of $S(X)$
2: report solution found by $\mathsf{TestSimpleSolution}(a)$ ▷ Excludes $S(X) \in a^+$ from further considerations.
3: let $\ell > 0$ be the length of the a-prefix of A_0
▷ By Lemma 53 $S(X)$ has the same a-prefix
4: replace each X in $\mathcal{A} = \mathcal{B}$ by $a^\ell X$ ▷ a^ℓ is stored in a compressed form,
▷ implicitly change $S(X) = a^\ell w$ to $S(X) = w$
5: let b be the last letter of $S(X)$
6: report solution found by $\mathsf{TestSimpleSolution}(b)$ ▷ Exclude $S(X) \in b^+$ from further considerations.
7: let $r > 0$ be the length of the b-suffix of the non-empty of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$
▷ By Lemma 53 $S(X)$ has the same b-suffix
8: replace each X in $\mathcal{A} = \mathcal{B}$ by Xb^r ▷ b^r is stored in a compressed form,
▷ implicitly change $S(X) = wb^r$ to $S(X) = w$

Lemma 58. *Let* a *be the first letter of the first word and* b *the last of the last word. If the first word is not a block of* a*s and the last not a block of* b*s then* CutPrefSuff *preserves solutions and after its application there are no crossing blocks of letters.*

Thus we can implement the block compression by first uncrossing all letters and then compressing them all.

12.4 The algorithm

The following algorithm OneVarWordEq is basically a specialisation of the general algorithm for testing the satisfiability of word equations [23] and is built up from procedures presented in the previous section.

Algorithm 12 OneVarWordEq Reports solutions of a given one-variable word equation

1: **while** the first block and the last block are not blocks of a letter **do**
2: *Pairs* ← pairs occurring in $S(\mathcal{A}) = S(\mathcal{B})$
3: BlockComp ▷ Compress blocks, in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time.
4: PreProc ▷ There are only two crossing pairs, see Lemma 57
5: *Crossing* ← list of crossing pairs from *Pairs* ▷ There are two such pairs
6: *Non-Crossing* ← list of non-crossing pairs from *Pairs*
7: **for** each $ab \in$ *Non-Crossing* **do**
▷ Compress non-crossing pairs, in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B})|$
8: PairCompNCr(a, b)
9: **for** $ab \in$ *Crossing* **do** ▷ Compress the 2 crossing pairs, in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B})|$
10: PairComp(a, b)
11: TestSolution ▷ Test solutions from a^*, see Lemma 61

We say that a word A_i (B_i) is *short* if it consists of at most 100 letters and *long* otherwise. To avoid usage of strange constants and its multiplicities, we shall use $K = 100$ to denote this value and we shall usually say that $K = \mathcal{O}(1)$.

Lemma 59. *Consider two consecutive letters a, b at the beginning of the phase in $S(\mathcal{A})$ for any solution S. At least one of those letters is compressed in this phase.*

Proof. Consider whether $a = b$ or not:

- $a = b$: In this case they are compressed using BlockComp.
- $a \neq b$: In this case ab is a pair occurring in the equation at the beginning of the phase and so it was listed in *Pairs* in line 2 and as such we try to compress it, either in line 8 or in line 10. This occurrence cannot be compressed only when one of the letters a, b was already compressed, in some other pair or by BlockComp. In either case we are done. □

□

Lemma 60. *Consider the length of the $(\mathcal{A}, i)$-word (or $(\mathcal{B}, j)$-word). If it is long then its length is reduced by $1/4$ in this phase. If it is short then after the phase it still is. The length of each unreported solution is reduced by at least $1/4$ in a phase.*

Additionally, if the first (last) word is short and has at least 2 letters then its length is shortened by at least 1 in a phase.

Proof. We shall first deal with the words and then comment how this argument extends to the solutions. Consider two consecutive letters a, b in any word at the beginning of a

phase. By Lemma 59 at least one of those letters is compressed in this phase. Hence each uncompressed letter in a word (except perhaps the last letter) can be associated with the two letters to the right that are compressed. This means that in a word of length k during the phase at least $\frac{2(k-1)}{3}$ letters are compressed i.e. its length is reduced by at least $\frac{k-1}{3}$ letters.

On the other hand, letters are introduced into words by popping them from variables. Let *symbol* denote a single letter or block a^ℓ that is popped into a word. We investigate, how many symbols are introduced in this way in one phase. At most one symbol is popped to the left and one to the right by BlockComp in line 3, the same holds for PreProc in line 4. Moreover, one symbol is popped to the left and one to the right in line 10; since this line is executed twice, this yields 8 symbols in total. Note that the symbols popped by BlockComp are replaced by single letters, so the claim in fact holds for letters as well.

So, consider any word $A_i \in \Sigma^*$ (the proof for B_j is the same), at the beginning of the phase and let A'_i be the corresponding word at the end of the phase. There were at most 8 symbols introduced into A'_i (some of them might be compressed later). On the other hand, by Lemma 59, at least $\frac{|A_i|-1}{3}$ letters were removed A_i due to compression. Hence

$$|A'_i| \leq |A_i| - \frac{|A_i|-1}{3} + 8 \leq \frac{2|A_i|}{3} + 8\frac{1}{3} \ .$$

It is easy to check that when A_i is short, i.e. $|A_i| \leq K = 100$, then A'_i is short as well and when A_i is long, i.e. $|A_i| > K$ then $|A'_i| \leq \frac{3}{4}|A_i|$.

It is left to show that the first word shortens by at least one letter in each phase. Consider that if a letter a is left-popped from X then we created B_0 and in order to preserve (5) the first letters of B_0 and A_0 are removed. Thus, A_0 gained one letter on the right and lost one on the left, so its length stayed the same. Furthermore the right-popping does not affect the first word at all (as X is not to its left); the same analysis applies to cutting the prefixes and suffixes. Hence the length of the first word is never increased by popping letters. Moreover, if at least one compression (be it block compression or pair compression) is performed inside the first word, its length drops. So consider the first word at the end of the phase let it be A_0. Note that there is no letter representing a compressed pair or block in A_0: consider for the sake of contradiction the first such letter that occurred in the first word. It could not occur through a compression inside the first word (as we assumed that it did not happen), cutting prefixes does not introduce compressed letters, nor does popping letters. So in A_0 there are no compressed letters. But this cannot happen.

Now, consider a solution $S(X)$. We know that $S(X)$ is either a prefix of A_0 or of the form $A_0^\ell A$, where A is a prefix of A_0, see Lemma 54. In the former case, $S(X)$ is compressed as a substring of A_0. In the latter observe that argument follows in the same way, as long as we try to compress every pair of letters in $S(X)$. So consider such a pair ab. If it is inside A_0 then we are done. Otherwise, a is the last letter of A_0 and b the first. Then this pair occurs also on the crossing between A_0 and X in $\mathcal{A}$, i.e. ab is one of the crossing pairs. In particular, we try to compress it. So, the claim of the lemma holds for $S(X)$ as well. □

Lemma 61. *For $a \in \Sigma$ we can report all solutions in which $S(X) = a^\ell$ for some natural ℓ in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. There is either exactly one ℓ for which $S(X) = a^\ell$ is a solution or $S(X) = a^\ell$ is a solution for each ℓ or there is no solution of this form.*

Note that we do not assume that the first or last word is a block of as.

A proof is left as an exercise.

Running time

Concerning the running time, we first show that one phase runs in linear time, which follows by standard approach, and then that in total the running time is $\mathcal{O}(n+\#_X \log n)$. To this end we assign in a fixed phase to each $(\mathcal{A}, i)$-word and $(\mathcal{B}, j)$-word cost proportional to their lengths in this phase. For a fixed $(\mathcal{A}, i)$-word the sum of costs assigned while it was long forms a geometric sequence, so sums up to at most constant more than the initial length of $(\mathcal{A}, i)$-word; on the other hand the cost assigned when $(\mathcal{A}, i)$-word is short is $\mathcal{O}(1)$ per phase and there are $\mathcal{O}(\log n)$ phases.

Lemma 62. *One phase of* OneVarWordEq *can be performed in* $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ *time.*

This is shown in the same way as in the case of recompression for SLPs.

The amortisation is much easier when we know that both the first and last words are long. This assumption is not restrictive, as as soon as one of them becomes short, the remaining running time of OneVarWordEq is linear.

Lemma 63. *As soon as first or last word becomes short, the rest of the running time of* OneVarWordEq *is* $\mathcal{O}(n)$.

Proof. One phase takes $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ time by Lemma 62 (this is at most $\mathcal{O}(n)$ by Lemma 60) and as Lemma 60 guarantees that both the first word and the last word are shortened by at least one letter in a phase, there will be at most $K = \mathcal{O}(1)$ many phases. Lastly, Lemma 61 shows that TestSolution also runs in $\mathcal{O}(n)$. □

So it remains to estimate the running time until one of the last or first word becomes short.

Lemma 64. *The running time of* OneVarWordEq *till one of first or last word becomes short is* $\mathcal{O}(n + (n_\mathcal{A} + n_\mathcal{B}) \log n)$.

Proof. By Lemma 62 the time of one iteration of OneVarWordEq is $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. We distribute the cost among the $\mathcal{A}$ words and $\mathcal{B}$ words: we charge $\beta|A_i|$ to $(\mathcal{A}, i)$-word and $\beta|B_j|$ to $(\mathcal{B}, j)$-word, for appropriate positive β. Fix $(\mathcal{A}, i)$-word, we separately estimate how much was charged to it when it was a long and short word.

- *long:* Let n_i be the initial length of $(\mathcal{A}, i)$-word. Then by Lemma 60 the length in the $(k+1)$-th phase it at most $(\frac{3}{4})^k n_i$ and so these costs are at most $\beta n_i + \frac{3}{4}\beta n_i + (\frac{3}{4})^2 \beta n_i + \ldots \leq 4\beta n_i$.
- *short:* Since $(\mathcal{A}, i)$-wordis short, its length is at most K, so we charge at most $K\beta$ to it. Notice, that there are $\mathcal{O}(\log n)$ iterations of the loop in total, as first word is of length at most n and it shortens by $\frac{3}{4}$ in each iteration when it is long and we calculate only the cost when it is long. Hence we charge in this way $\mathcal{O}(\log n)$ times, so in total $\mathcal{O}(\log n)$.

Summing those costs over all phases over all words and phases yields $\mathcal{O}(n+(n_\mathcal{A}+n_\mathcal{B}) \log n)$. □

13 Free groups

13.1 Free groups

Given a finite alphabet Σ define Σ^{-1} as $\{a^{-1} : a \in \Sigma\}$. We define a reduction $aa^{-1} \to \epsilon$. Any word in $(\Sigma \cup \Sigma^{-1})^*$ has a unique normal form under this reduction (in which there is no factor aa^{-1}); such words are called *reduced* or *irreducible*, for a word w this normal form is called and denoted by $\mathrm{IRR}(w)$.

A free group over generators Σ consists of reduced words $\mathrm{IRR}((\Sigma \cup \Sigma^{-1})^*)$ over $(\Sigma \cup \Sigma^{-1})^*$. The multiplication of w and $w' \in \mathrm{IRR}((\Sigma \cup \Sigma^{-1})^*)$ is defined as

$$w \cdot w' = \mathrm{IRR}(ww') \ .$$

It is easy to check that this operation is well defined and that it defines a group.

13.2 Free monoids/semigroups with involution

In a similar way, we treat Σ^* as a *free monoid* or *free semigroup* over the set of generators Σ. In such a setting we talk about word equations in free monoids/semigroups.

An *involution* (defined for any monoid) is a bijection $\bar{\cdot} : M \mapsto M$ such that $\overline{\overline{x}} = x$, $\overline{xy} = \overline{y}\,\overline{x}$ for each $x, y \in M$. In case of a free monoid $(\Sigma \cup \Sigma^{-1})^*$ the involution on a letter a is defined as a^{-1}, where $(a^{-1})^{-1} = a$. In case of groups, the inverse operator is also an involution.

We shall also denote a free group with geenrators $g_1, g_2, \ldots, g_\ell$ by $F(g_1, g_2, \ldots, g_\ell)$. Given two free groups $\mathbb{G}$, $athbbG'$ by $mathbbG * \mathbb{G}'$ we denote the free groups with the set of generators that is a disjoint union of generators of $\mathbb{G}$ and $\mathbb{G}'$.

13.3 Word equation in free groups

We consider word equations in free groups. From algebraic perspective they are more interesting than the semigroups. Makanin extended his results for word equations [38, 39]. We can naturally see a word equation over a free group as an ordinary word equation over a free monoid $(\Sigma \cup \Sigma^{-1})^*$, however, we may loose some solutions in this way: consider an equation $aX = bY$. Naturally it has no solution as a word equation, but it does in a free group: take $X = a^{-1}$ and $Y = b^{-1}$.

In general, the reduction is possible, assuming that we allow regular constraints and *involution* in the equation.

13.4 Reduction: equations in groups to word equations

Firstly, each equation can be reduced to a form $XY = Z$ or $X = a$ by adding appropriate amount of variables.

Given one such equation we can replace it by a system of equations

$$X = X'R \quad Y = R^{-1}Y' \quad X'Y' = Z$$

Then any solution of the original equation gives a solution of the new system in which $X'Y'$ is irreducible and any irreducible solution of the new system gives a solution of the old one.

So it is left to turn such a system of word equations in groups into an equisatisfiable system in a free monoid with involution.

We take the equation as they are and regular constraints that say that there are no factors aa^{-1} in any variable, for any $a \in (\Sigma \cup \Sigma')$. Then we need to deal with the R^{-1}: for each such variable we introduce another equation $R^{-1} = \overline{R}$.

It is easy to see that the new system has a solution (as a semigroup) iff the original system had a solution.

Finally, note that the regular constraints about the irreducible form can be encoded in a different way.

We shall later show how to solve equations (in a free semigroup) with regular constraints and involution.

14 Positive theory of free groups and free semigroups with recognisable constraints

Given a free group $\mathbb{G}$ (the definition is similar in case of semigroups) a positive sentence is of a form

$$Q_1 x_1 Q_2 x_2 \dots Q_k x_k \varphi(x_1, x_2, \dots, x_k)$$

where each Q_i is a quantifier and φ is a formula that uses only variables $x_1, x_2, \dots, x_k$ and constants from $\mathbb{G}$, atomic formulas are equations or constraints of a form $\mu(X) = g$ and only $\wedge$ and $\vee$ as used as connectives. A *positive theory* of a free group $\mathbb{G}$ consists of positive sentences that hold in $\mathbb{G}$. The corresponding decision problem asks to decide, whether a given sentence belongs to a positive theory (of $\mathbb{G}$).

It is an easy exercise to show that positive theory of a free semigroup is undecidable (exercise). On the other hand, the positive theory of a free semigroup is decidable, as shown by Makanin [39]. Below we show this result, in a variant given by Diekert and Lohrey [10], which is somehow based on idea of Gurevich to use random words.

14.1 General comments

In essence, we proceed by a quantifier elimination: given a formula

$$\forall_1 x_1 \exists y_1 \exists y_2 \dots \exists y_k \varphi(x, y_1, y_2, \dots, y_k, \vec{z}) \tag{6}$$

(note that $\vec{z}$ is a sequence of free variables) we construct a formula

$$\exists \exists y_1 \exists y_2 \dots \exists y_k \varphi'(k, y_1, y_2, \dots, y_k, \vec{z}) \tag{7}$$

such that for each sequence of elements $\vec{z}$ the formula (6) holds if and only if (7) holds. The element k is chosen in a special way and moreover we consider those formulas in different free groups.

Processing the consecutive quantifiers leads to an existential formula (of perhaps large size in a larger free group). Decidability of the existential formula will be shown on the next lecture.

The main property of positive formulas is that they are preserved under homomorphisms: if a positive sentence $\varphi(\vec{(z)})$ (where $\vec{(z)}$ is a vector of elements) holds in some structure A and $\varphi : A \mapsto B$ is a homomorphism, then $\varphi\vec{(z)}$ holds in B. This observation is made precise at appropriate point.

We shall extend our free semigroup by new elements: $G * \{k\}$ is a free group with generators of G and k (so this is a group generated by $\Sigma \cup \{k\}$). In such a case we need to also extend μ, it is enough to define it on k. The notation μ_h^k means μ extended by defining $\mu(k) = h$. We shall use this notation also for several elements.

14.2 Quantifier elimination

Let us a fix a formula

$$\Phi(\vec{Z}) = \forall X_1 \exists Y_1 \cdots \forall X_m \exists Y_m \varphi(X_1, \ldots, X_m, Y_1, \ldots, Y_m, \vec{Z}). \tag{8}$$

Theorem 65. *For all $\vec{z}$ the formula* (8) *holds in $\mathbb{G}$ if and only if*

$$\exists Y_1 \in \mathbb{G} * F(k_1) \exists Y_2 \in \mathbb{G} * F(k_1, k_2) \cdots \exists Y_m \in \mathbb{G} * F(k_1, k_2, \ldots, k_m) \varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}). \tag{9}$$

holds in $G * F(k_1, k_2, \ldots, k_m)$.

The idea is that the universally quantified variables act like "independent constants".

The true reason for this is that since our formula holds for "any x", it means that it holds for random word (in appropriate sense) x. But such a random word has very little interference with other words (it provably has only a couple of letters that reduce) So in some sense it "is" a constant. Still, we need to allow the following variables to "use" this new constant, thus we allow Y_i to use $\{k_1, k_2, \ldots, k_i\}$. Consider a simple example $\forall X \exists Y XY = 1$. Then when we replace X with k we get $\exists Y kY = 1$ which is satisfiable for $y = k^{-1}$.

The proof is done by induction on the quantifiers.

By assumption

$$\forall X_2 \exists Y_2 \cdots \forall X_m \exists Y_m \varphi(x_1, X_2, \ldots, X_m, y_1, Y_2, \ldots, Y_m, \vec{Z}).$$

(note that x_1 and y_1 are free variables, i.e. we can take any elements for them) holds in $\mathbb{G}$ if and only if

$$\exists Y_2 \in \mathbb{G} * F(k_2) \cdots \exists Y_m \mathbb{G} * F(k_2, \ldots, k_m) \varphi(x_1, k_2, \ldots, k_m, y_1, \ldots, Y_m, \vec{z}).$$

holds in $\mathbb{G} * F(k_2, \ldots, k_n)$.

For simplicity, denote by $\mathbb{G}_i$ the $\mathbb{G} * F(k_2, k_3, \ldots, k_i)$ (note that $\mathbb{G} = \mathbb{G}_1$)

As x_1, y_1 is a free variable there, we can take the quantification over all $x_1, y_1 \in \mathbb{G}$ and get

$$\forall X_1 \exists Y_1 \forall X_2 \exists Y_2 \cdots \forall X_m \exists Y_m \varphi(X_1, X_2, \ldots, X_m, Y_1, \ldots, Y_m, \vec{Z}).$$

if and only if

$$\forall X_1 \in \mathbb{G} \exists Y_1 \in \mathbb{G}_1 \exists Y_2 \in \mathbb{G}_2 \ldots \exists Y_m \in \mathbb{G}_m \varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}), \tag{10}$$

In the following we show its equivalence to (9).

Lemma 66. *For φ positive if $\vec{z}$ satisfies* (9) *then it satisfies* (10)

Proof. We use the already mentioned fact that positive sentences are preserved by homomorphisms.

Take any $x \in \mathbb{G}$. Take a homomorphism $h(k_1) = x$ and define it as an identity on other elements. Then the homomorphism applied to $\varphi(k_1, \ldots, k_m, y_1, \ldots, y_m, \vec{z})$ yields $\varphi(x, \ldots, k_m, \varphi(y_1), \ldots, \varphi(y_m), \vec{z})$ and we can take $\varphi(y_i)$ as a witness for Y_i. □

For the proof in the other direction we shall also use the reduction to the monoid. Note that a reduction described in the previous chapter reduces the problem of equations in free groups to free semigroups. Denote by $\mathbb{M}, \mathbb{M}_i$ the free monoid (with involution) corresponding to $\mathbb{G}, \mathbb{G}_i$. Then the formula

$$\varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z})$$

is rewritten into formula

$$\exists \vec{y} \varphi'(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y})$$

where the new variables $\vec{Y}$ are used to appropriately brake down the equations.

Our technical claim is that

Lemma 67. *If*

$$\forall X_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_1 \in \mathrm{IRR}(\mathbb{M}_1) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_2) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_m) \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_m) \\ \varphi'(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}),$$

then there exist two words $s_1, s_2 \in \mathrm{IRR}(M)$ such that

$$\exists Y_1 \in \mathrm{IRR}(\mathbb{M}_1 * \{k_1, \overline{k_1}\}^*) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_2 * \{k_1, \overline{k_1}\}^*) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_m * \{k_1, \overline{k_1}\}^*) \\ \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_m * \{k_1, \overline{k_1}\}^*) \varphi(s_1 k_1 s_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}),$$

We shall first show, how we can deduce our mail claim from that. Suppose that (8) holds, i.e.

$$\forall X_1 \exists Y_1 \cdots \forall X_m \exists Y_m \varphi(X_1, \ldots, X_m, Y_1, \ldots, Y_m, \vec{Z})$$

then by the reduction to the free monoid also

$$\forall X_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_1 \in \mathrm{IRR}(\mathbb{M}_1) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_2) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_m) \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_m) \\ \varphi'(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}),$$

holds. Thus by the lemma also

$$\exists Y_1 \in \mathrm{IRR}(\mathbb{M}_1 * \{k_1, \overline{k_1}\}^*) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_2 * \{k_1, \overline{k_1}\}^*) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_m * \{k_1, \overline{k_1}\}^*) \\ \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_m \{k_1, \overline{k_1}\}^*) \varphi'(s_1 k_1 s_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}),$$

holds. So we can lift it back to the group setting, i.e. there are $s_1, s_2 \in \mathbb{G}$ such that

$$\exists Y_1 \in \mathbb{G}_1 * F(k_1) \exists Y_2 \in \mathbb{G}_2 * F(k_1) \ldots \exists Y_m \in \mathbb{G}_m * F(k_1) \varphi(s_1 k_1 s_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}),$$

consider an isomorphism defined by $h(k) = s_1^{-1} k s_2^{-1}$. Since it is an isomorphism, we can apply it on above equation. The only affected is the k_1 constant, so we get the following is equivalent:

$$\exists Y_1 \in \mathbb{G}_1 * F(k_1) \exists Y_2 \in \mathbb{G}_2 * F(k_1) \ldots \exists Y_m \in \mathbb{G}_m * F(k_1) \varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}), \tag{11}$$

But this is our intended claim.

14.3 Proof of Lemma 67

It is left to show the proof of Lemma 67.

Take two different constants a, b and fix some word ℓ of length at least 2 that use both constants. Fix $\lambda \geq 2d + 1$, where d is the number of equations. Consider a set $R = \{r_0, r_1, \ldots, r_\lambda\} \subseteq \{a, b\}^m$, where m is some large constant (to be established later). Later we will consider a string

$$s = r_0 \ell r_1 \ell \ldots r_{\lambda-1} \ell r_\lambda$$

Roughly, this is a string that we use for $\forall X$ quantifier.

We require that strings in R have *enough randomness*: each word w of length at least $(|r_i| - \ell)/2$ occurs in at most one of strings $r_0, r_1, \ldots, r_\lambda, \overline{r_0}, \overline{r_1}, \ldots, \overline{r_\lambda}$ and it has at most one occurrence is such a string.

Using Kolmogorov complexity/Probabilistic method it is easy to show that such set of strings exists, for large enough m (exercise).

The true application of enough randomness is that

Lemma 68. *If $r \in R \cup \overline{R}$ occurs in $r_i \ell r_{i+1}$ then this is either a prefix or suffix of $r_i \ell r_{i+1}$ (so $r = r_i$ or $r = r_{i+1}$).*

For the proof it is enough to try to place r within $r_i \ell r_{i+1}$, and see that it will have an overlap with r_i or r_{i+1} of length at least $(|r| - \ell)/2$, which is a contradiction.

Now, since m is large enough, we can also assume that it is longer than twice any constant that occurs in the equations.

Consider rewriting systems $P_1, \ldots, P_\lambda$, defined as

$$P_i = \{(r_{i-1} \ell r_i, r_{i-1} k_1 r_i), (\overline{r_i} \overline{\ell} \overline{r_{i-1}}, \overline{r_i} \overline{k_1} \overline{r_{i-1}})\}$$

From Lemma 68 it easily follows that each of this rewriting systems is confluent and so has a unique normal form, denote by $\kappa_i(w)$ the unique normal form of w under the rewriting system P_i.

We say that t contains the cut of (u, v) if there is an occurrence of t in uv that is not contained in u nor in v.

Lemma 69. *Given a pair of strings (u, v) there are at most two different $r_i \ell r_{i+1}$ that contain their cut.*

Proof. Otherwise there are three. So consider the first of those occurrences and the last. They overlap with at least one letter. Then the middle occurrence overlaps with at least half of its length with the first one or last one, so some r occurs in $r_i \ell r_{i+1}$, which cannot be. □

Lemma 70. *Let $\{x_j y_j = z_j\}_{j=1}^d$ be all nontrivial equations. The there is i such that for each j*

$$\kappa_i(x_j)\kappa_i(y_j) = \kappa_i(z_j)$$

Proof. For a fixed equation there are at most 2 different $r_i \ell r_{i+1}$ that contain a cut between x_j and y_j. So there is one $r_i \ell r_{i+1}$ that does not contain any cut. Hence when we calculate the normal form, each rewriting on $x_j y_j$ is done separately on x_j and y_j, which show the claim. □

Since the rewriting systems introduce a fresh constant that is not rewritten, the

$$\kappa_i(x_j)\kappa_i(y_j) = \kappa_i(z_j) \text{ implies } x_j y_j = z_j$$

holds always.

Hence this yields the proof of Lemma 67 we take s as the string substituted for x and all the witnesses. We then take the appropriate rewriting system and rewrite all the constants. By Lemma 70 x is replaced by $\kappa_i(x)$ and each y is replaced with $\kappa_i(y)$, so all true equations still hold (there may be some more true equations, but this is fine as the sentence is positive). Note that x is now replaced with a fixed string that has only one occurrence of k_1. And ys are witnesses from the allowed sets.

15 Solving equations in free groups

15.1 Recognisable/Regular sets

As a first step, we would like to define the notion of a regular language (or set) in a more algebraic setting, so that it could be also generalised to the free groups.

Consider Σ^*, think of it as a free semigroup. A regular language is defined using an automaton N, let it have n states Q. Then the transition function naturally defines (Boolean) transition matrices, whose rows and columns are indexed by Q: for a letter a the has $m_{p,q} = 1$ iff we can go from p to q using letter a. Note that such a transition matrix can be defined for each word $w \in \Sigma^*$ and so we have a natural homomorphism from Σ^* to $\mathbb{M}$, that is, the set of Boolean matrices of size $n \times n$.

A regular language can be defined using this homomorphism as well: note that a word is accepted if its transition matrix leads from starting state to final state. In other words, there is a finite amount of matrices, which are accepting, and the rest is rejecting. Using the homomorphism, there is a finite subset M' such that $R = \varphi^{-1}(M')$.

This approach can be now lifted to groups: a set R in a free group F is *regular*, if there is a homomorphism φ from F to Boolean matrices M with a subset M' such that $R = \varphi^{-1}(M')$. Note, that since we work in a group setting, we require, that the homomorphism respect the inversion, i.e. each element $\varphi(a)$ is invertible, as Boolean matrix. We would like to extend the notion of regular sets also to the case of free group.

If we consider a monoid with involution, then we usually assume that the recognisable constraints are given by a homomorphism that also respects this involution. The involution can be the inverse on the Boolean matrices (which is the case for free groups), but could any other operation, for instance — the transpose. But could any other operation that satisfies the needed properties.

We usually denote the homomorphism to matrices by ρ and talk about the *transition* of a letter.

15.2 Regular constraints

In the most convenient case, we specify the regular constraints with a series of conditions of a form $X \in R$, $X \notin R'$. Each such conditions is potentially given by a different automaton. When we move to the matrix setting, creating one matrix for all such conditions essentially corresponds to the creation of one automaton for the appropriate Boolean combination of such conditions, which is expensive. Instead, we can think that ρ assigns a tuple of matrices, rather than just one. This allows to save space.

Secondly, the list of conditions for X: $X \in R_i$, $X \notin R'_i$ can be viewed as a restriction of $\rho(S(X))$ to a couple of legal transitions. In our algorithm we think that the constraints are given by specifying the actual transition for $S(X)$. From computational point of view this is not restricting, as we can initially non-deterministically guess the appropriate transition from a set of transitions.

15.3 Main issue

As planned, we reduce solving of equations in free groups to free semigroup with involution and recognisable constraint. It turns out that the main issue is the bounding of the alphabet used in the solution. We shall deal with this problem at the end, as it distorts a little the flow of the argument. At the moment, imagine that we begin with the given alphabet and whenever we make a compression, we add the new letter into the alphabet. Note that this means that we can arrive at the same equation with different alphabets (which may mean that the shortest solution is of different length). It is *not* possible to simply remove those letters from the alphabet and from the equation, as they may be needed for the the regular constraints.

Keeping such a large alphabet *is* a problem, as we cannot give a standard PSPACE argument that an equation cannot repeat.

15.4 Goal

Our goal is to modify our approach for word equations so that it works also for free monoids with involution and regular constraints. In essence, we proceed in a similar way, so only the differences will be described.

Firstly, now a substitution S is a solution if its satisfies the equation and for each variable $\rho(S(X)) = \rho(X)$, i.e. also the constraints are satisfied. Due to involution, we also require that $S(\overline{X}) = \overline{S(X)}$.

We want to guarantee that

If an equation is satisfiable then after the appropriate compression/uncrossing the new equation also is. If the new equation is satisfiable, then also the original one was. (in fact, in both cases there should be some correspondence of the solutions).

15.5 Needed modifications

15.5.1 Constraints

Whenever we pop letters, we need to guess new values for variables, so that the total value is the same. For instance, when we replace X with aX' then it should hold that $\rho(aX') = \rho(X)$. The value for X' is guessed and verified. We also need to guess when we remove the variable, in which case we need to have $\rho(X) = \rho(\epsilon)$.

15.5.2 Involution

When we replace X with wXw' then we also need to replace $\overline{X}by\overline{wXw'} = \overline{w'}\,\overline{X}\overline{w}$.

When we compress ab to c then we also need to compress $\overline{ab}$ to $\overline{c}$. Firstly, this affects the notion of a crossing pair (ab may be crossing due to $\overline{b}\overline{a}$. Concerning the replacement, this is easy, as long as ab and $\overline{ab}$ do not overlap, which can happen only when $a = \overline{a}$ or $b = \overline{b}$. There are different possible approaches now. We present one, in which we forbid

the creation of self-involuting letters, which boils down to forbidding to compression of $a\overline{a}$ as a pair.

15.5.3 Pair compression

It is also easy to see that we can compress several pairs in parallel, as well as they are disjoint. Thus our partition are of a form $(A, \overline{A})$, which are good enough, in the sense that they also cover a constant fraction of letters. (Recall that we do not compress $a\overline{a}$.

15.5.4 Blocks and Quasiblocks compression

With such a restriction the blocks compression works as intended. We could also do the variant with only compression of two letters and using other variables for representing a-blocks, but here we need to be careful: while we can move the extra a to the left, for $\overline{a}$ we then need to move the to the right. This is fine, as $a \neq \overline{a}$

There is a problem with $(a\overline{a})^k$, as we do not compress it at all. We do this similarly to blocks compression: we replace $(a\overline{a})^k$ with $c_k\overline{c_k}$. Note that technically c_k "represents" a self-involuting string, but we "forgot" about this. But this is fine, as $c_k\overline{c_k}$ is self-involuting.

As a result, $a\overline{a}a$ is still not compressible, but this is the longest incompressible string and so we still get a PSPACE algorithm, with a constant-larger space consumption.

15.6 Letters

As already noted, we cannot assume that there is a solution over the letters that are in the equation. This is because the letters that are crossed out have non-trivial transitions and removing them changes the total transition of a substitution for a variable.

The easiest solution is the extend the initial alphabet so that it has one letter for each possible transition (not that in this way the alphabet may become exponential) and considering solutions over the letters that are in the equation and in the initial alphabet (Exercise).

We follow a slightly more involved approach, which is much more useful, when we want to describe the set of all solutions of a word equation.

The idea is that if there is a letter in the substitution for a variable that is not in the equation not it is a letter from the original equation, then in some sense it was a mistake to compress this letter in the first place. But each letter in any equation corresponds to some string of letters in the original equation. To track the meaning of constants outside the current equation, we additionally require that a solution (over an alphabet Σ') supplies some homomorphism $\alpha : \Sigma' \mapsto A^*$, which is constant on A and compatible with ρ, in the sense that $\rho(b) = \rho(\alpha(b))$ for all letters b. Thus, we change the notion of a solution: from now on it is a pair (S, α). In particular, given an equation (U, V) the $\alpha(S(U))$ corresponds to a solution of the original equation. Note, that α is a homomorphism with respect to the involution, i.e. we assume that $\alpha(\overline{a}) = \overline{\alpha(a)}$.

It is easy to define a new α after a compression operation: when w is replaced wit hc then we simply denote $\alpha(c) = \alpha(w)$ (note, that it may be that for two different letters we get that $\alpha(c) = \alpha(c')$, but this is not a problem, as we never assume that $\alpha(c) \neq \alpha(c')$). In particular, if (α, S) is a solution of an equation $U = V$ then after the compression/popping the new equation $U' = V'$ has a solution (α', S') such that $\alpha(S(U)) = \alpha'(S'(U'))$ (for appropriate non-deterministic choices).

A solution is *simple* if it uses only letters from the equation and input alphabet. Given a non-simple solution (S, α) we can replace all constants $c \notin \Sigma$ (where Σ is the alphabet of the equation) in all $S(X)$ by $\alpha(c)$ (note, that as $\rho(c) = \rho(\alpha(c))$, the $\rho(X)$ is preserved in this way). This process is called a *simplification* of a solution and the obtained substitution S' is a *simplification* of S. It is easy to show that S' is a solution and that $\alpha(S'(U)) = \alpha(S(U))$, so in some sense both S and S' represent the same solution of the original equation.

However, replacing single letters in substitution by long words contradicts the very idea of the method, which only shortens the solutions. We need to devise some more precise measure that can be used instead of length of the solution.

A *weight* of a solution (S, α) of an an equation (U, V) is

$$\mathrm{w}(S, \alpha) = |U| + |V| + \sum_{X \in \Omega} |UV|_X |\alpha(S(X))| \ , \tag{12}$$

It is easy to see that all compression and popping operations decrease the weight (if something changes in the equation) or keep it constant, when nothing changes. Furthermore, the simplification preserves the weight, see Lemma 71. Thus weight can be used to show the termination of the algorithm.

Lemma 71. *Suppose that (S, α) is a solution of the equation (U, V). Then the simplification (S', α) of (S, α) is also a solution of (U, V), $\alpha(S'(U)) = \alpha(S(U))$ and $\mathrm{w}(S', \alpha) = \mathrm{w}(S, \alpha)$.*

Proof. Let Σ be the alphabet of the equation and Σ' the alphabet of the solution S. Consider any constant $b \in \Sigma' \setminus \Sigma$. As it does not occur in the equation, all its occurrences in $S(U)$ and $S(V)$ come from the variables, i.e. from some $S(X)$. Then replacing all occurrences of b in each $S(X)$ by the same string w preserves the equality of $S(U) = S(V)$, thus S' is also a solution. Since we replace some constants b with $\alpha(b)$ (and $\alpha \circ \alpha = \alpha$), clearly $\alpha(S(X)) = \alpha(S'(X))$ for each variable, in particular, the weight contributed by each variable occurrence does not change. Furthermore, as $\rho(c) = \rho(\alpha(c))$ we have that $\rho(S(X)) = \rho(S'(X))$. Thus, $\alpha(S'(U)) = \alpha(S(U))$ and $\mathrm{w}(S', \alpha) = \mathrm{w}(S, \alpha)$, as claimed. □

Lemma 72. *For any subprocedure, if it transforms a satisfiable equation (U, V) to a satisfiable equation $(U', V') \neq (U, V)$ then the corresponding solution of (U', V') has smaller weight than the solution of (U, V).*

Proof. Note that in (12) the parts corresponding to the substitutions for variables do not change. But if anything changes in the equation, some constants were compressed and so the weight drops. □

This gives the termination argument of our algorithm. We proceed within PSPACE, keeping some solution, after the compression operation we replace the corresponding solution by its simplification. The weight decreases after the first operation and does not change after the second. Thus we end up in a trivial equation.

16 Representation of all solutions

As we want to describe the set of all solutions, ideally there should be a one-to-one correspondence between the solutions before and after the application of used subprocedures.

However, as those subprocedures are non-deterministic and the output depends on the non-deterministic choices, the situation becomes a little more involved. What we want to guarantee is that no solution is 'lost' in the process and no solution is 'gained': given a solution for some non-deterministic choices we transform the equation into another one, which has a 'corresponding' solution and we know a way to transform this solution back into the original equation. Furthermore, when we transform back in this way any solution of the new equation, we obtain a solution of the original equation, i.e. we do not gain solutions.

For technical reasons it is more convenient to assume that solutions assign ϵ to each variable that is not present in the equation and $\neq \epsilon$ to all other variables. (Note that this causes some slight problems, when we have constraints on variables not present in the equation, but this is not a big issue). Such substitutions are called *non-empty*.

By an *operator* we denote a function that transforms substitutions (for variables). All our operators have simple description: $S'(X)$ is usually obtained from $S(X)$ by morphisms, appending/prepending constants, etc. In particular, they have a polynomial description. We usually denote them by φ and their applications by $\varphi[S]$.

Definition 73 (Transforming the solution)**.** Given a (nondeterministic) procedure we say that it *transforms the equation* (U, V) *with a solution* (S, α) if

- Based on the nondeterministic choices and equation (U, V) we can define an operator φ, called the *corresponding inverse operator*.
- For some nondeterministic choices the procedure returns an equation (U', V') with a nonempty solution (S', α') such that $\varphi[S'] = S$. Furthermore, $\alpha(S(U)) = \alpha'(S'(U))$ and $\mathrm{w}(S', \alpha') \leq \mathrm{w}(S, \alpha)$ and if $(U, V) \neq (U', V')$ then this inequality is in fact strict.

  In such a case we also say that this procedure *transforms* (U, V) *with* (S, α) *to* (U', V') *with* (S', α').
- For every equation (U', V') that can be returned by this procedure applied on (U, V) and any of its solution (S', α') and for every operator $\varphi \in \Phi$ the $(\varphi[S'], \alpha_0)$ is a solution of (U, V) for some homomorphism $\alpha_0 : B \mapsto A^+$ compatible with ρ, where B is the alphabet of $\varphi[S'](U)$.

If a procedure transforms every equation with every nonempty solution then we say that it *transforms solutions*.

Example 1. Consider a procedure that can replace X with aX (for any constant a) and Y by bY (also for any constant b). Then it transforms the equation $X = Y$ with a solution $S(X) = cc, S(Y) = cc$ and any α: The inverse operator φ prepends a to $S(X)$ and b to $S(Y)$, where a and b are constants that were introduced by the procedure. If (S', α') is a solution of the obtained equation then $(\varphi[S'], \alpha')$ is a solution of the original equation; note that when $a \neq b$ then the obtained equation does not have solutions at all, but this is fine with the definition. Moreover, for the nondeterministic choice in which we pop c from both X and Y, the obtained equation has a solution $S'(X) = S'(Y) = c$, for which $S = \varphi[S']$.

On the other hand, this procedure does not transform solutions: a solution $S(X) = S(Y) = c$ cannot be transformed, as we do not allow empty solutions. We can modify the procedure, though: it can either replace X with aX or with a, similarly for Y. In our case $X = Y$ with $S(X) = S(Y) = c$ is transformed into $c = c$ with a solution $S(X) = S(Y) = \epsilon$. It is easy to see that this modified procedure transforms solutions.

It is easy to see that our compression/uncrossing operation do transform the solutions, but there is a slight problem with the blocks compression: when we pop a^ℓ then the inverse operator depends on ℓ. Instead, we shall use the variant, in which we only compress aa in blocks compression.

Lemma 74. *Each subprocedure of uncrossing + appropriate compression transforms the solutions.*

The proof follows by easy case inspection.

We use the variant of the recompression algorithm in which we keep an $\mathcal{O}(n^2)$ equation but at each step choose exactly one compression operation to be performed.

Lemma 75. *Suppose that (U_0, V_0) is an equation of size $\mathcal{O}(n^2)$ and $|U_0| > 1$ or $|V_0| > 1$, let it have a solution (S_0, α_0). Then for some nondeterministic choices and appropriate uncrossing and compression the returned equation (U_1, V_1) and the inverse operator φ satisfy*

- *(U_1, V_1) is of size $\mathcal{O}(n^2)$*
- *(U_0, V_0) with (S_0, α_0) is transformed to (U_1, V_1) with (S_1', α_1), φ is the corresponding inverse operator and (S_1, α_1) is a simplification of (S_1', α_1).*

Now we can use this Lemma to build the graph representation of all solutions: for the input equation (U, V) we construct a directed graph $\mathcal{G}$ which has nodes labelled with equations of size $\mathcal{O}(n^2)$. Then for such node, say labelled with (U_0, V_0), such that $|U_0| > 1$ or $|V_0| > 1$ we use Lemma 75 to list all equations (U_1, V_1) such that $(U_0, V_0), (U_1, V_1)$ satisfy the claim of Lemma 75 for some solution of (U_0, V_0). For each such equation we put the edge $(U_0, V_0) \to (U_1, V_1)$ and annotate it with the appropriate operator φ. We lastly remove the nodes that are not reachable from the starting node and those that do not have a path to an ending node.

In this way we obtain a finite description of all solution of a word equation with involution and recognisable constraints.

Theorem 76. *There exists effectively a* PSPACE *algorithm working as follows.*
Input. *A word equation with regular constraints over a free monoid with involution.*
Output. *A finite graph representation of all solutions of the equation.*

17 Terms and Unification

17.1 Labelled trees

We deal with rooted, ordered trees, usually denoted with letters t or s. Nodes are labelled with elements from a *ranked alphabet* Σ, i.e. each letter $a \in \Sigma$ has a fixed arity $\mathrm{ar}(f)$; those elements are usually called *letters*. A tree (term) is *well-formed* if a node labelled with f has exactly $\mathrm{ar}(f)$ children; we consider only well-formed trees, which can be equivalently seen as *ground terms* over Σ. In this setting Σ is usually called a *signature* and its elements *function symbols*.

17.2 What the variables represent

For trees (terms) usually the notion of equations is not used and instead we talk about the *unification*.

It is natural to ask, what the variables should represent. In the basic scenario, each variable $x \in \mathcal{X}$ represents a (well-formed) tree. In such a case the corresponding unification problem is called (*first-order*) *term unification*.

Definition 77. In (first order) term unification we are given a collection of equations $e_i \stackrel{?}{=} f_i$, where each side is a $\Sigma \cup \mathcal{X}$ labelled tree, where all elements of $\mathcal{X}$ have arity 0.

A *solution* is a mapping from $\mathcal{X}$ to a set of well-formed Σ-labelled trees that turns each formal equation into an equality; application of a solution to e simple replaces x with $S(x)$.

It is easy to show that this problem is in P.

Theorem 78. *The satisfiability of an instance of first order term unification is in* P. *In fact it can be solved in linear time.*

A simple proof is left as an exercise.

As a philosophical note: the unification problem here is solved in a top-down fashion.

17.3 Patterns

We consider not necessarily well-formed fragments of trees. Thus we want to define 'trees with holes' that represent missing arguments. Let $\mathbb{Y} = \{\bullet, \bullet_1, \bullet_2, \ldots\}$ be an infinite set of symbols of arity 0, we think of each of them as a place of a missing argument. Its elements are collectively called *parameters*. A *pattern* is a tree over a signature $\Sigma \cup \mathbb{Y}$, where each element of $\mathbb{Y}$ is treated as a constant. A pattern is *linear*, if each parameter occurs at most once in a pattern; linear patterns are also called *ground contexts*. The usual convention is that the used parameters are $\bullet_1, \bullet_2, \ldots, \bullet_k$, or $\bullet$, when there is only 1 parameter; for linear patterns we usually assume that the occurrences (according to preorder traversal of the pattern) of the parameters in the pattern is $\bullet_1, \bullet_2, \ldots, \bullet_k$. We often refer to *parameter nodes* and *non-parameter nodes* to refer to nodes labelled with parameters and non-parameters, respectively. A pattern using r parameters is called r-pattern.

17.4 Second order unification

In second order unification we allow *second order variables*, denoted by capital letters $X, Y, \ldots$ and coming from a set $\mathcal{V}$. Each such a variable X has arity $\mathrm{ar}(X)$. A *second order term* is a term built with $\Sigma \cup \mathcal{V}$. A *second order unification* consists of a sequence of equations of second-order terms. A *substitution* Sassigns to each variable X a pattern $S(X)$ whose parameters are from $\bullet_1, \bullet_2, \ldots, \bullet_{\mathrm{ar}(X)}$; note that some parameters may be unused.

We define $S(t)$ for a second order term in a natural way:

- $S(f)(t_1, \ldots, t_k) = f(S(t_1), \ldots, S(t_k))$ when $f \in \Sigma$
- $S(X)(t_1, \ldots, t_k) = (S(X))[\bullet_1/S(t_1), \ldots, \bullet_k/S(t_k)]$

where $(S(X))[\bullet_1/S(t_1), \ldots, \bullet_k/S(t_k)]$ means the term $S(X)$ with each parameter $\bullet_i$ replaced with $S(t_i)$.

A substitution is a *solution* if it turns each formal equality into tree equality of terms.

18 Linear Monadic Second Order Unification

We begin with a problem which is somehow in between second order unification and word equations.

In linear monadic second order unification we require that the substitutions for a variable are linear (so each parameter is used at most once) and we work over signature of letters of arity at most 1. So comparing to word equations, we have letters and one extra nullary symbol that is always at the end (we shall denote it by "." and ignore it). The variables do not represent words, but rather λ-functions, in the sense that X is now a function $\lambda x.w_x$, where we require that w is built solely of symbols and possibly x used once. The difference is that X can ignore its argument and simply terminate the hole term.

Example 2.

$$Xa. = Yb.$$

There is a valid solution $X = \lambda x.a.$ and $Y = \lambda y.a.$. Note that there are also other solutions.

It is easy to see that if an equation $U = V$ is satisfiable as a word equation then it is satisfiable as linear monadic second order unification, but not the other way around.

One other difference is that our encoding into one equation no longer works as a substitution may drop some other substitutions.

Since there are more solutions, intuitively it should be easier to solve such an equation. In some sense this is the case: this problem is in NP.

Theorem 79. *Satisfiability of a a linear monadic second order unification is in* NP.

Our approach is as previously, i.e. we will apply the local compression rules and keep the size of the instance small. The additional twist is that whenever possible we shall try to replace the left-most variables with closed functions, i.e. the ones that ignore their argument.

We begin with stating that our subprocedures for word equations indeed work in this setting. We need a twist, though: Pop and CutPrefSuff are also allowed to replace a variable by a ".".

Lemma 80. Pop, CutPrefSuff, BlockCompNCr, PairCompNCr *are sound and complete.*

The proof for compression operations is the same as for word equations, for popping operations some analysis is needed, as we may pop to the right from a variable that should be replaced with a closed function. This is not a problem, though.

Additionally, the exponential bound on the exponent of periodicity holds also in case of linear monadic second order unification.

Lemma 81. *Let S be the length-minimal solution of linear monadic second order unification and let w^k be a substring of $S(X)$. Then $k \leq 2^{cn}$ for some constant c, where n is the sum of length of the equations.*

Proof. Take the solution S and the variables that forget their arguments in this substitution. Modify the equations according to those variables: if a side of an equation is $\alpha X \beta$, where X forgets the argument, we replace it with $\alpha X.$. Then we obtain a system, in which no variable forgets its argument and this is equivalent to system of words equations. In particular, w^k has $k \leq 2cn$ for appropriate n. □

Hence, at any point we can ensure, in non-deterministic polynomial time, that the size of the instance is at most cn^2 for a suitable c: if not then we run compression and uncrossing until it is reduced to cn^2.

Simplifying assumptions Without loss of generality we can assume that:

- for each equation at least one of its sides begin with a variable;
- for each equation both of its sides contain a variable.

What may be surprising, is that removing letters from the left-sides of the equations is fine but removing variables is not. We consider only the case of left sides, as we are only interested in that.

Lemma 82. *The systems* $\{U_i = V_i\}_{i \in I} \cup \{aU = aV\}$ *and* $\{U_i = V_i\}_{i \in I} \cup \{U = V\}$ *are equisatisfiable for a letter* a.

The systems $\{U_i = V_i\}_{i \in I} \cup \{XU = XV\}$ *and* $\{U_i = V_i\}_{i \in I} \cup \{U = V\}$ *are in general* not *equisatisfiable for a variable* X.

Both proofs are left as exercises.

Our algorithm shall eliminate one variable using polynomially many steps, each of those steps increases the size of the instance by $\mathcal{O}(n)$. This guarantees that the whole algorithm runs in NP: after the removal the instance is of polynomial size. In polynomially many steps we reduce it to size $\mathcal{O}(n^2)$ and then iterate again, with less variables.

Removing a variable

We now describe how to eliminate one variable.

Definition 83. For a system of equations define a *dependence graph*. Its set of vertices are labelled with variables and we create an edge $X \xrightarrow{w} Y$ for each equation $XU = wYV$.

Note that if there is an equation $XU = YV$ then we add edges $X \xrightarrow{\epsilon} Y$ and $Y \xrightarrow{\epsilon} X$.

Lemma 84. *If there is an edge from* $X \xrightarrow{w} Y$ *then for each solution* S *of this system* $S(X)$ *either*

1. *is a prefix of* w*;*
2. *has a prefix* w

In particular, if $X \xrightarrow{w} Y$ *and* $X \xrightarrow{w'} Y'$ *then either*

- w *is a prefix of* w' *or*
- w' *is a prefix of* w *or*

- $S(X)$ *is a prefix of* w' *and* w.

The proof is obvious.

Corollary 85. *If there are two edges from* X *labelled with nonempty words then they have the same first letter or* $S(X) = \epsilon$ *for each solution* S.

Let us investigate, what happens with the dependence graph, when we left-pop a from X. Then for every edge $X \xrightarrow{w} Y$ we change the label to $X \xrightarrow{a^{-1}w} Y$ and for every edge $Y \xrightarrow{w} Y$ we change the label to wa (not that if there is an edge $X \xrightarrow{w} X$ then both effects apply and so we have the word $a^{-1}wa$ and so for $w = \epsilon$ we are left with an epsilon).

Define a relation on the variables: $X < Y$ if there is a path from X to Y whose labels concatenate to a non-empty word. Also, define the relation of *equivalence*: $X \sim Y$ if there is a path from X to Y whose all edges are labelled with ϵ. As such edges are bi-directional, this is an equivalence relation.

Lemma 86. *If* X *is a minimal element of* $<$*, so are all its equivalent variables.*

Lemma 87. *If* $X \sim Y$ *then in each solution either one of them is* ϵ *or they begin with the same letter.*

Proof. The proof is obvious for $X \xrightarrow{\epsilon} Y$, the rest follows by an easy induction on the length of path consisting of ϵ-edges. □

The algorithm

We can now move to the algorithm. We look at the dependence graph, as long as the relation $<$ is acyclic, we find a minimal element (say X) and left-pop a from each $Y \sim X$. Note that in this way no letters are introduced on the edges: ϵ edges are preserved (as we either pop from both their ends or from none), we take a from some edges and we do not add a to any edge: all edges incoming to X are ϵ edges and so all those variables are equivalent to X.

We change the $<$ order in this way, as new ϵ edges may have been introduced: for instance, when $X \xrightarrow{aw} Y$ and $X \xrightarrow{a} Z$ then after left-popping a from X we have $X \xrightarrow{\epsilon} Z$. In particular, we may have introduce new cycles in $<$ relation: in the example above it could be that there is an edge from Y to Z, so after popping there is a cycle from Z to Z with a non-empty label.

Such left-popping reduces the total length of edges, so it is linear in the input size.

When we finish with popping, either there is no edge with nonempty label, so all variables are equivalent and the equation is satisfiable, as we can substitute . for everything, or there is a cycle from X to X that defines a nonempty label.

In the second case we take the shortest (in terms of number of edges) such cycle, let it be from X to X and let the word on it be w_X. Note that $|w_X| = \mathcal{O}(n)$: the cycle cannot have repeating nodes, so also there are no repeating edges, so each label is used at most once and their concatenation is not longer than the input.

We want to apply pair compression which reduces w_X to a sequence of the form a^k. This is always possible: unless w_X is of desired form, it has two different consecutive labels, say ab. Then we make that ab compression and we proceed; there are linearly many such compressions.

We consider the effect of popping and pair compression on the dependence graph and our chosen cycle from X to X.

Note that popping does not affect the concatenation of labels on the cycle, unless one variable is removed (and we stop, when one variable is removed).

For the compression, we use a variant in which we left-pop b from each variable that begins with b. We claim that in this way after the uncrossing each ab that is on the cycle is on one label on the cycle: suppose that an edge ends with a and there is a sequence of ϵ edges and an edge that begins with b. But then all those variables connected with ϵ edges have the same first letter: b. So we should have popped from them all and a should not be the last label (note that we use $a \neq b$ here), contradiction. Hence the compression is performed on the ab in the labels.

Thus after the compression we have a cycle from X to X labelled with powers of a only. We claim that in each solution of this system of equations there is at least one variable Y on this cycle which has a substitution $S(Y) = a^{\ell}(\bullet)$ (that is, it does not forget the argument).

Suppose that this is not the case. As each variable on this cycle has a first letter a, every one of them has a substitution $S(Y) = a^{\ell_Y} b_Y \ldots$, where b_Y is a letter or a terminating symbol. Clearly, for equivalent variables the length ℓ and letter b has to be the same (this is clear when there is an edge from X to Y, i.e. when $X \cdots = Y \cdots$, as in this case we assume that $S(X)$ and $S(Y)$ are not sequences of a, so the first non-a symbol on both sides need to match. The rest follows by induction).

So take a variable Y on the cycle, which has the longest a-prefix and incoming edge labelled with non-empty word. This corresponds to an equation $Z \cdots = a \ldots aY \cdots$. But the a-prefix of the left-hand side is shorter than the right-hand side, contradiction.

So we can choose one variable, which has a substitution a^{ℓ}. Guess exponential ℓ, replace Y with this value and make the blocks compression (for this compression we completely disregard the dependence graph, which is now not needed at all).

19 General second order unification

In general, the second order unification is undecidable

Theorem 88. *Second order unification is undecidable.*

We encode the Satisfiability of Diophantine equations. The signature consists of constants c, c', unary symbol a and binary symbol g.

We encode a number n as a pattern $a^n(\bullet)$; we use $\underline{n}$ to denote $a^n(\bullet)$. Thus each second order variable N has an equation

$$a(N(c)) = N(a(c))$$

It is easy to check that each solution of such an equation is of the form $a^n(\bullet)$.

Without loss of generality each Diophantine equation is of a form $m+n = p$, $m \cdot n = p$ or $n = 1$. The first is easily encodable as

$$M(N(c)) = P(c)$$

the last as

$$N(c) = a(c)$$

It remains to describe how to encode multiplication.

Note that we can encode sequences of terms using g: a sequence $t_1, t_2, \ldots, t_k$ is encoded as $[t_1, t_2, \ldots, t_k]$ which is $g(t_1, g(t_2, \cdots g(t_{k-1,t_k})))$.

We introduce an auxiliary second-order variable $G(\bullet_1, \bullet_2, \bullet_3)$.

The equations in question are

$$G(c, c', [[P(c), N(c')], c]) = [[c, c'], G(M(c), \underline{1}(c'), c)]$$
$$G(c', c, [[P(c'), N(c)], c]) = [[c', c], G(M(c'), \underline{1}(c), c)]$$

We claim that they hold for M, N, P if and only if $mn = p$.

The intended solution is as follows: define $t_k = [\underline{m \cdot k}\bullet_1, \underline{k}\bullet_2]$. Then the substitution for G is

$$[t_0, t_1, t_2, \ldots, t_{n-1}, \bullet_3]$$

while the substitution for N is $\underline{n}$, for M is $\underline{m}$ and for P is $\underline{p}$, where $nm = p$. We check that his is a solution only of the first equation, the second one is similar.

$$G(c, c', [[P(c), N(c')], c]) = [[\underline{0}(c), \underline{0}(c')], [\underline{m}(c), \underline{1}(c')], \ldots, [\underline{m \cdot (n-1)}(c), \underline{(n-1)}(c')], [\underline{p}(c), \underline{n}(c')], c]]$$

On the other hand, the value of the right hand side is

$$[[c, c'], G(M(c), \underline{1}(c'), c)] =$$
$$[[c, c'], [\underline{0(m)}(c), \underline{0(1)}(c')], [\underline{m(m)}(c), \underline{1(1)}(c')], \ldots, [\underline{(n-1)m(m)}(c), \underline{n-1(1)}(c')], c]$$

Using a simple fact that $\underline{l(\ell')} = \underline{\ell + \ell'}$ we get

$$= [[c, c'], [\underline{m}(c), \underline{1}(c')], [\underline{2m}(c), \underline{2}(c')], \ldots, [\underline{nm}(c), \underline{n}(c')], c]$$

So both sides are equal.

We proceed in the other direction. Suppose that N, M, P, G are such that they satosfy the equations

$$G(c, c', [[P(c), N(c')], c]) = [[c, c'], G(M(c), \underline{1}(c'), c)]$$
$$G(c', c, [[P(c'), N(c)], c]) = [[c', c], G(M(c'), \underline{1}(c), c)]$$

Note that we know that $N = \underline{n}$, $M = \underline{m}$ and $P = \underline{p}$. We want to show that $nm = p$.

Clearly G is a list. We compare the elements of those list one by one and use the fact hat the equation "offset" those values by 1 position in the list. Furthermore, the two equations swithc places of c and c' so we cannot have anything "constant", everythin need to come from parameters. By

$$G(c, c', [[P(c), N(c')], c]) = [[c, c'], G(M(c), \underline{1}(c'), c)]$$

we conclude that

$$G = [[c, c'] \ldots] \text{ or } G = [[\bullet_1, \bullet_2] \ldots] \text{ or G} = [\bullet_3\ ]$$

The first option is not possible due to the second equation, the third is a terminating condition that we consider later on. So let the second option hold, i.e.

$$G = [[\bullet_1, \bullet_2] \ldots]$$

We apply this to the right-hand side of the first equation and conclude that the value is

$$[[\bullet_1, \bullet_2], [M(c), \underline{1}(c')] \ldots]$$

Looking at the left-hand side we try to conclude what is the second element of the list. Again, due to c — c' symmetry this has to be $[M(\bullet_1), \underline{1}(\bullet_2)]$ or $\bullet_3$. We iterate this process, obtaining that

$$G = [t_0, t_1, t_2, \ldots]$$

Since it is finite, at some point we need to choose that the last element is $\bullet_3$. But then G is already of the form we considered before and it is easy to conclude that since this is a solution, then $nm = p$, as desired.

20 Context Unification

20.1 Introduction

20.1.1 Context unification

Solving equations, whether they are over groups, fields, semigroups, terms or any other objects, was always a central point in mathematics and the corresponding decision problems received a lot of attention in the theoretical computer science community. Solving equations can be equally seen as unification problem, as we are to unify two objects (with some variables).

Context unification is one of prominent problems of this kind, let us first introduce the objects we will work on. Given a signature, i.e. a set of function symbols of given arities, we define a ground context in a usual way, i.e. as well formed term. A ground context is a ground term with exactly one occurrence of a special constant that represents a missing argument; one should think of it as a 'hole' or a variable to be instantiated by a ground term later on. Ground contexts can be applied to ground terms, which results in a replacement of the special constant by the given ground term; similarly we can define a composition of two ground contexts, which is again a ground context. Hence we can built terms using ground contexts, treating them as function symbols of arity 1.

In context unification problem we are given a signature, a set of term variables (which shall denote ground terms) and a set of context variables (which shall denote ground contexts). Using those variables we can built terms: we simply treat each context variable as a function symbol of arity one and each term variable as a constant. A context equation is an equation between two such terms and a solution of a context equation assigns to each context variable a ground context (over the given input signature) and to each variable a ground term (over the same signature) such that both sides of the equation evaluate to the same (ground) term. The context unification is the decision problem, whether a context equation has a solution; the name comes from the fact that an equation can be equally seen as an unification: in some sense we unify the two contexts on the sides of the equation.

Context unification was introduced by Comon [7, 8] (who also coined the name) and independently by Schmidt-Schauß [53]. It found usage in analysis of rewrite systems with membership constraints [7, 8], analysis of natural language [43, 42], distributive unification [54], bi-rewriting systems [30].

In a broader sense, context unification is a special case of second-order unification, in which the argument of the second-order variable X can be used unbounded number of

times in the substitution term for X (also, there may be many parameters for a second order variable, this is however not an essential difference). On the other hand, when the underlying signature is restricted to the case when only unary function symbols and constants are allowed, the context equation is in fact a word equation (in this well-known problem we are given an equation $u = v$, where u and v are strings of letters and variables and we are to substitute the variables with strings so that this formal equation is turned into a true equality of strings). The second order unification is known to be undecidable [19] (even in very restricted cases [15, 29, 31]), however, the proofs do not apply to the case of context unification as they essentially use the fact that the argument may be used many times in the substitution term. On the other hand, the satisfiability of word equations is known to be decidable (in PSPACE [46]) and up to recently there were essentially only three different algorithms for this problem [37, 48, 46]; whether these algorithms generalise to context unification remains an open question. Hence context unification is both upper and lower-bounded by two well-studied problems.

The problem gained considerable attention in the term rewriting community [49], mainly for two reasons: on one hand it is the only known natural problem which is subsumed by second order unification (which is undecidable) and subsumes word equations (which are decidable) and on the other hand it has several ties to other problems, see Section 20.1.2. There was a large body of work focused on context unification and several partial results were obtained:

- a fragment in which any occurrence of the same context variable is always applied to the same term is decidable [8];
- stratified context unification, in which for any occurrence of a fixed second-order variable X the string of second-order variables from this occurrence to the root of the containing term is the same, is decidable [55] (this problem is even known to be NP-complete [35]);
- a fragment in which every variable and context variable occurs at most twice (such equations are usually called *quadratic*) is decidable [29];
- a fragment in which there are only two context variables is decidable [58];
- the notion of exponent of periodicity, which is crucial in algorithms for solving word equations, is generalised to context unification and so is the exponential bound on it [57];
- context unification reduces to its fragment in which the signature contains only one binary symbol and constants [33];
- context unification with one context variable is known to be in NP [16] and some of its fragments are in P [17]. It remains an open question, whether the whole problem is in P.

Note that in most cases the corresponding variants of the general second order unification remain undecidable, which gave hope that context unification is indeed decidable.

20.1.2 Extensions and connections to other problems

The context unification was shown to be equivalent to 'equality up to constraint' problem [43] (which is a common generalisation of equality constraints, subtree constraints

and one-step rewriting constraints). In fact one-step rewriting constraints, which is a problem extensively studied on its own, are equivalent to stratified context unification [42]. It is known that the first-order theory of one-step rewriting constraints is undecidable [62, 40, 64]. The case of general context unification was improved by Vorobyov, who showed that its $\forall\ \exists^8$-equational theory is Π^0_1-hard [65].

Some fragments of second order unification are known to reduce to context unification: the *bounded second order unification* assumes that the number of occurrences of the argument of the second-order variable in the substitution term is bounded by a constant; note that it *can be zero* and this is the crucial difference with context unification; cf. monadic second order unification, which can be seen as a similar variant of word equations, which is known to be NP-complete [34]. This fragment on one hand easily reduces to context unification and on the other hand it is known to be decidable [56] (in fact its generalisation to higher-order unification is decidable as well [59] and it is known that bounded second order unification is NP-complete [35]). In particular, the work presented here imply the decidability of bounded second order unification, but the obtained computational complexity is worse.

The context unification can be also extended by allowing some additional constraints on variables and context variables, a natural one allows the usage of the tree-regular constraints (i.e. we assume that the substitution for the variables and context variable come from a certain regular set of trees). It is known that such an extension is equivalent to the linear second order unification [32], defined by Levy [29]: in essence, the linear second order unification allows bounding variables on different levels of the function, which makes direct translations to context unification infeasible, however, usage of regular constraints gives enough additional power to actually encode such more complicated bounding. Note that the reductions are not polynomial and the equivalence is stated only on the decidability level.

The usage of regular constraints is very popular in case of word equations, in particular it is used in generalisations of the algorithm for word equation to the group case and essentially all known algorithms for this problem can be generalised to word equations with regular constraints [60, 11, 12].

20.1.3 Context unification and word equations

A word can be seen as a term over signature containing only unary symbols (plus some constant at the bottom) and vice versa. Thus the two compression operations for word equations generalise naturally to subterms containing only unary function symbols. Hence the recompression for terms uses the two already mentioned operations (which are applicable only to function symbols of arity one [1]) but it also introduces another local compression rule, designed specifically for terms: we replace a term $f(t_1, \ldots, t_{i-1}, c, t_{i+1}, \ldots, t_m)$ (where c is a constant) with $f'(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m)$, where f' is a fresh function symbol (i.e. not used the context equation, it can however be in Σ). While such a compression introduces new function symbols, it does not increase the maximal arity of functions in the signature, which proves to be important (as the space consumption depends on this maximal arity). This new rule requires also a generalisation of the variable replacements

[1] Note that by work of Levy [33] it is enough to consider context unification with constants and a single binary symbol. However, our algorithm will transforms the input instance and it can introduce unary symbols. So even if the input has no unary letters, we cannot guarantee that the current context equation stored by the algorithm also does not have such letters. Moreover, it remains unknown, whether such an approach can used in presence of regular constraints or for describing set of all solutions.

(x by ax or xb): when X denotes a context, we sometimes replace it with $a(X)$, where a is a unary letter, or $X(f(x_1, x_2, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_m))$, where $x_1, x_2, \ldots, x_m$ are new variables denoting full terms and '$\bullet$' denotes the place in which we apply the argument.

As in the case of word equations, the key observation is that while the variable replacements increase the size of the context equation (proportionally to the number of occurrences of variables in the context equation), the replacement rules guarantee that the size of the context equation is decreased by a constant factor (for proper nondeterministic choices). Those two effects cancel each out and the size of the context equation remains polynomial.

20.2 Compression of trees

20.2.1 Patterns

We want to replace (linear) patterns of a tree with new letters. In this section the pattern is by default a linear pattern.

We often refer to *parameter nodes* and *non-parameter nodes* to refer to nodes labelled with parameters and non-parameters, respectively. A pattern p *occurs* (at a node v) in a tree t if p can be obtained by taking a subtree t' of t rooted at v and replacing some of subtrees of t' by appropriate parameters. This is also called an *occurrence* of p in t. A pattern p is a subpattern of t if p occurs in t.

Given a tree t, its r-subpattern p occurrence and a pattern p' we can naturally replace p with p': we delete the part of t corresponding to p with removed parameters and plug p' with removed parameters instead and reattach all the subtrees in the same order; as the number of parameters is the same, this is well-defined. We can perform several replacements at the same time, as long as occurrences of patterns do not share non-parameter nodes. In this terminology, our algorithm will replace occurrences of subpatterns of t in t.

We focus on some specific patterns: A *chain* is a pattern that consists only of unary letters. We consider chains consisting only of two different unary letters, called *pairs*, and a-chains, which consists solely of letters a. A chain t' that is a subpattern of t is a *chain subpattern* of t, an occurrence of a chain subpattern a^ℓ is *a-maximal* if it cannot be extended by a nor up nor down. A pattern of a form $f(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, c, \bullet_i, \ldots, \bullet_{\mathrm{ar}(f)-1})$ is denoted by (f, i, c) for short.

We treat chains as strings and write them in the string notation (in particular, we drop the parameters) and 'concatenate' them, i.e. for two chains $s(\bullet)$ and $s'(\bullet)$ we write them as s, s' and ss' denotes the chain obtained by replacing the parameter in s by s'. We use those conventions also for 1-patterns.

20.2.2 Local compression of trees

We perform three types of subpattern compression on a tree t:

a-chain compression For a unary letter a we replace each a-maximal chain subpattern a^ℓ for $\ell > 1$ by a new unary letter a_ℓ.

ab compression For two unary letters a and b we replace each subpattern ab with a new unary letter c.

(f, i, c) compression For a constant c and letter f of arity $\mathrm{ar}(f) = m \geq i \geq 1$, we replace each subpattern (f, i, c), i.e. $f(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, c, \bullet_i, \ldots, \bullet_{m-1})$ with $f'(\bullet_1, \bullet_2, \ldots, \bullet_{i-1}, \bullet_i, \ldots, \bullet_{m-1})$ where f' is a fresh letter of arity $m-1$ added to Σ (intuitively: the constant c on i-th place is 'absorbed' by its father labelled with f).

They are all collectively called *subpattern compression*. When we want to specify the type but not the actual subpattern compressed, we use the names *pair compression*, *chain compression* and *leaf compression*. These operations are also called $\mathsf{TreePattComp}(ab, t)$, $\mathsf{TreePattComp}(a, t)$ and $\mathsf{TreePattComp}((f, i, c), t)$.

Observe that the a-chain compression and ab compression are direct translations of the operations used in the recompression-based algorithm for word equations [23]. To be more precise, both those compressions affect only chains, return chains as well, and when

a chain is treated as a string the result of those compressions corresponds to the result of the corresponding operation on strings. On the other hand, the leaf compression is a new operation that is designed specifically to deal with trees.

In the next sections the following observation, which bounds the maximal arity of the letters introduced during the compression steps, proves useful.

Lemma 89. *If the maximal degree of nodes in t is k then in t' that is obtained after subpattern*

Proof. Observe that the chain compression replaces chain of unary nodes with a single unary node. Similarly, pair compression replaces chains of length two with single unary letters. Lastly, leaf compression can only reduce the arity of a node (or keep it the same). □

20.3 Context unification

In this section we define context unification problem and the notions necessary to state it. The presentation here is slightly different than the usual one, c.f. [58], as we use the 'pattern' terminology rather than 'context' one (which is more general). Our terminology is less standard for context unification, but more popular in tree-compression approach. The differences are just in naming conventions and at appropriate places we also mention the alternative names of used concepts.

By $\mathcal{V}$ we denote an infinite set of context variables X, Y, Z, ... We also use individual term variables x, y, z, ... taken from Ω. When we do not want to distinguish between a context variable or term variable, we call it *variable* and denote by a small greek letter, like α.

Definition 90. The *terms* over Σ, Ω, $\mathcal{V}$ are ground terms with alphabet $\Sigma \cup \Omega \cup \mathcal{V}$ in which we extend ar to $\Omega \cup \mathcal{V}$ by $\mathrm{ar}(X) = 1$ and $\mathrm{ar}(x) = 0$ for each $X \in \mathcal{V}$ and $x \in \Omega$.

A *context equation* is an equation of the form $u = v$ where both u and v are terms.

We call the letters from Σ that occur in a context equation the *explicit letters* and talk about explicit occurrences of letters in a context equation. Since X represents a pattern, we write it in the string notation.

We are interested in the solutions of the context equations, i.e. substitutions that replace variables with ground terms and context variables with ground contexts, such that a formal equality $u = v$ is turned into a true equality of ground terms. More formally:

Definition 91. A *substitution* is a mapping S that assigns a 1-pattern $S(X)$ to each context variable $X \in \mathcal{V}$ and a ground term $S(x)$ to each variable $x \in \Omega$. The mapping S is naturally extended to arbitrary terms as follows:

- $S(a) := a$ for each constant $a \in \Sigma$;
- $S(f(t_1, \ldots, t_n)) := f(S(t_1), \ldots, S(t_m))$ for an m-ary $f \in \Sigma$;
- $S(X(t)) := S(X)(S(t))$ for $X \in \Omega$.

A substitution S is a *solution* of the context equation $u = v$ if $S(u) = S(v)$. The *size* of a solution S of an equation $u = v$ is $|S(u)|$, which is simply the total number of nodes in $S(u)$. A solution is *size-minimal*, if for every other solution S' it holds that $|S(u)| \leq |S'(u)|$. A solution S is non-empty if $S(X)$ is not a parameter for each $X \in \Omega$ from the context equation $u = v$.

The 1-patterns substituted for context variables are also called *ground contexts* in the literature (hence the name context variable) and the parameter is also called 'a hole' of a context.

In the following, we are interested only in non-empty solutions. Notice that this is not restricting, as for the input instance we can guess, which context variables have empty substitution in the solution and remove them.

For a ground term $S(u)$ and an occurrence of a letter a in it we say that this occurrence *comes from* u it is was obtained as $S(a)$ in Definition 91 and that it comes from X (or x) if it was obtained from $S(X)$ (or $S(x)$, respectively) in Definition 91.

Example 3. Consider a signature $\Sigma = \{f, c, c'\}$ with $\mathrm{ar}(f) = 2$ and $\mathrm{ar}(c) = \mathrm{ar}(c') = 0$ and an equation $X(c) = Y(c')$ over it. It has a solution (which is easily seen to be size-minimal) $S(X) = f(\bullet, c')$ and $S(Y) = f(c, \bullet)$ and in fact each solution needs to use f, which does not occur in the context equation. Furthermore, if we consider this equation over a signature that does not have any letter of arity greater than 1 then the equation is not satisfiable.

Restrictions on signature It is easy to observe that if Σ has no constant then there is no solution (as no term can be formed). Moreover, if Σ contains only letters of arity 0 and 1 then the input equation $u = v$ is essentially a word equation, with a tweak at the end

- if u, v end with different constants then we reject;
- if u, v end with the same variable then we remove it;
- if u (v) ends with a variable then we replace it with a fresh context variable, if it ends with a constant then we remove this constant.

It is easy to see that this procedure returns an equivalent word equation, and satisfiability of word equations is known to be in PSPACE [47, 23], also when regular constraints are allowed [11, 12].

Thus, in the following we always assume that the signature contains a constant and a letter of arity at least 2.

20.4 Compressions on the equation

We first do not consider problems that arise due to the growing signature: when we perform a subpattern compression we simply add the appropriate letter to the signature and consider the solutions over this new signature. We resolve this technical problem after stating the correctness of the algorithm, in Section 20.9.5.

A very general class of operations is sound:

Lemma 92. *The following operations are sound:*

1. *Replacing all occurrences of a variable α with $t\alpha$ throughout the $u = v$, where t is a 1-pattern or a term.*
2. *Replacing all occurrences of a context variable X with $Xf(x_1, x_2, \ldots, x_{i-1}, \bullet, x_{i+1}, \ldots, x_{\mathrm{ar}(f)})$ throughout the $u = v$ where $x_1, \ldots, x_{\mathrm{ar}(f)}$ are fresh term variables and $\mathrm{ar}(f) \geq 1$.*

3. *Replacing all occurrences of a context variable X (variable x) with a 1-pattern p (term t, respectively).*

4. *subpattern compression performed on $u = v$.*

Proof. The proof follows a simple principle: if the obtained equation $u' = v'$ has a solution S' then we can define a solution S of the original context equation by reversing the performed operation.

In 1, if S' is a solution of the new equation then $S(\alpha) = S'(t)S'(\alpha)$ is a solution.

Similarly, in 2, if S' is a solution of the new equation then $S(X) = S'(X)(f(S'(x_1)), S'(x_2), \ldots, S'(x_{i-1}), \bullet, S'(x_{i+1}), \ldots, S'(x_{\mathrm{ar}(f)})))$ is a solution of the original equation.

In 3 if S' is a solution of the new equation, we define S in the same way, but set $S(\alpha) = t$.

In 4, consider first the leaf compression. Let f' denote the letter that replaced f with child c at positions i during the (f, i, c) compression. Let S' be a solution of the new equation, we define a solution S: if $S'(\alpha)$ contains the occurrences of a letter f', then we replace the whole subterm $f'(t_1, t_2, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k)$ in $S'(\alpha)$ with $f(t_1, \ldots, t_{i-1}, c, t_{i+1}, \ldots, t_k)$. For pair compression, if the letter c that replaced ab occurs in $S(\alpha)$then we replace it with a pair ab. Similarly, for chain compression S is obtained from S' by replacing each occurrence of a letter a_ℓ with a chain a^ℓ (for all $\ell \geq 2$).

It is easy to see that in each of those cases the defined substituion is a valid solution of the original equation. □

The notion of explicit/implicit/crossing subpattern is generalised from word equtions to context unification in a natural way.

Definition 93. For an equation $u = v$ and a substitution S we say that an occurrence of a subpattern p in $S(u)$ (or $S(v)$) is

explicit with respect to S all non-parameter letters in this occurrence come from explicit letters in $u = v$;

implicit with respect to S all non-parameter letters in this occurrence come from $S(\alpha)$for a single occurrence of a variable α;

crossing with respect to S otherwise.

We say that ab (a; (f, i, c)) is a *crossing pair* (has a crossing chain; is a crossing father-i-leaf subpattern) with respect to S if it has at least one crossing occurrence (there is a crossing occurrence of an a^ℓ chain; has at least one crossing occurrence) with respect to S. Otherwise ab (a, (f, i, c)) is a *non-crossing pair* (has no crossing chain; is a non-crossing father-i-leaf subpattern) with respect to S.

Similarly as in case of word equations, the compression of non-crossing subpatterns is simply performed on the equation.

Lemma 94. PattCompNCr *is sound.*

If $u = v$ has a solution S such that one of the following holds:

- *ab is non-crossing with respect to S*
- *a has no crossing chains with respect to S*

- (f, i, c) *is non-crossing with respect to* S

then the corresponding algorithm $\mathsf{PattCompNCr}(ab, `U = V')$ *or* $\mathsf{PattCompNCr}(a, `U = V')$ *or* $\mathsf{PattCompNCr}((f, i, c), `U = V')$ *is complete. To be more precise, the returned equation* $u' = v'$ *has a solution* S' *such that* $S'(u') = \mathsf{TreePattComp}(ab, S(u))$ *(or* $S'(u') = \mathsf{TreePattComp}(a, S(u))$ *or* $S'(u') = \mathsf{TreePattComp}((f, i, c), S(u))$*, depending on the chosen compression). This solution is over a signature expanded by letters representing introduced during the subpattern compression.*

Proof. By Lemma 92 PattCompNCr is sound.

Concerning the completeness, we give the proof in the case of pair compression, it is the same also in the case of chain compression and leaf compression.

Suppose that $u = v$ has a solution S such that a, b is non-crossing with respect to S. We define a substitution S' for the obtained equation $u' = v'$ such that $S'(u') = \mathsf{TreePattComp}(ab, S(u))$ and symmetrically $S'(v') = \mathsf{TreePattComp}(ab, S(v))$. Since $S(u) = S(v)$ this shows that S' is indeed a solution of $u' = v'$ and so the second claim of the lemma holds.

The definition is straightforward: $S'(\alpha)$ is obtained by performing the a, b compression on $S(\alpha)$formally $S'(\alpha) = \mathsf{TreePattComp}(ab, S(\alpha))$.

Consider an occurrence of a pattern ab in $S(u)$ and where this chain subpattern comes from:

they both come from explicit letters Then $\mathsf{PattCompNCr}(ab, `U = V')$ will perform the ab compression on them, i.e. replace them with a letter c.

they both come from $S(X)$ or $S(x)$ Then this occurrence of ab is replaced by the definition of S'.

one of them comes from an explicit letter and one from $S(X)$ or $S(x)$ But then ab is crossing with respect to S, contradicting the assumption.

As the argument applies to every occurrence of chain subpattern ab, this shows that $S'(u') = \mathsf{TreePattComp}(ab, S(u))$. As already said, the proof in case of chain compression and leaf compression is the same, which ends the proof of the lemma. □

20.5 Uncrossing

The uncrossing of pairs and chains is done similalry as in the case of word equations, so let us move to the uncrossing of father-i-leaf pairs.

20.6 Uncrossing father-leaf subpattern

We now show how to uncross a father-i-leaf subpattern (f, i, c). As a first step, we give a more operational characterisation of the crossing father-i-leaf subpatterns. It is easy to observe that father-i-leaf subpattern (f, i, c) is crossing (with respect to a non-empty S) if and only if one of the following holds for some context variables X and y

(CFL 1) f with an i-th son y occurs in $u = v$ and $S(y) = c$ *or*

(CFL 2) Xc occurs in $u = v$ and the last letter of $S(X)$ is f and $\bullet$ is its i-th child *or*

(CFL 3) Xy occurs in $u = v$, $S(y) = c$ and f is the last letter of $S(X)$ and $\bullet$ is its i-th child.

Lemma 95. *Let S be non-empty. Then (f, i, c) is a crossing father-i-leaf subpattern if and only if one of (CFL1)–(CFL3) holds, for some context variables X and term variable y.*

As in the case of pairs and chains, the proof follows by a simple case inspection.

The modifications needed to uncross the father-leaf subpattern are in fact the only new uncrossing operations, when compared with the recompression technique for strings, however, they are similar to the one in the case of uncrossing a pair: We want to 'pop-up' c and 'pop-down' f. The former operation is trivial, but the details of the latter are not, let us present the intuition.

- In (CFL1) we *pop up* the letter c from x, which in this case means that we replace each x with $c = S(x)$. Since x is no longer in the context equation, we can restrict the solution so that it does not assign any value to x.
- In (CFL2) we *pop down* the letter f: let $S(X) = sf(t_1, \ldots, t_{i-1}, \bullet, t_i, \ldots, t_{m-1})$, where s is a 1-pattern and each t_i is a ground term and $\mathrm{ar}(f) = m$. Then we replace each X with $Xf(x_1, x_2, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m-1})$, where $x_1, \ldots, x_{m-1}$ are fresh variables. In this way we implicitly modify the solution $S(X) = s(f(t_1, t_2, \ldots, t_{i-1}, \bullet, t_i, \ldots, t_{m-1}))$ to $S'(X) = s$ and add $S'(x_j) = t_j$ for $j = 1, \ldots, m-1$. If $S'(X)$ is empty, we remove X from the equation.
- The third case (CFL3) is a combination of (CFL1)–(CFL2), in which we need to pop-down from X and pop up from y.

Algorithm 13 Uncross$((f, i, c), `U = V')$

for $x \in \Omega$ **do**
  if $S(x) = c$ **then** ▷ Guess
    replace each x in $u = v$ by c ▷ S is no longer defined on x
let $m \leftarrow \mathrm{ar}(f)$
for $X \in \mathcal{V}$ **do**
  if f is the last letter of $S(X)$, $\bullet$ is its i-th child and Xc is a subpattern in $u = v$ **then** ▷ Guess
    replace each X in $u = v$ by $X(f(x_1, x_2, \ldots, x_{i-1}, \bullet, x_{i+1}, \ldots, x_m))$
      ▷ Implicitly change $S(X) = sf(t_1, t_2, \ldots, t_{i-1}, \bullet, t_i, \ldots, t_{m-1})$ to $S(X) = s$
      ▷ Add new variables $x_1, \ldots, x_{m-1}$ to Ω and extend S by setting $S(x_j) = t_j$
    if $S(X)$ is empty **then** ▷ Guess
      remove X from the equation: replace each $X(t)$ in the equation by t
for new variables $x \in \Omega$ **do**
  if $S(x) = c$ **then** ▷ Guess
    replace each x in $u = v$ by c ▷ S is no longer defined on x

There is a subtle difference between uncrossing a pair a, b and uncrossing father-i-leaf subpaterns: for a pair popping down letter a is unconditional while the corresponding popping down of f from X is done only when it is really needed: i.e. we want to make some (f, i, c) compression, f is the last letter of $S(X)$, its i-th child is $\bullet$ and some occurrence

of X is applied on c. This assumption turns out to be crucial to bound the number of introduced variables, see Lemma 103.

Lemma 96. *Let* $\mathrm{ar}(f) \geq i \geq 1$ *and* $\mathrm{ar}(c) = 0$*, then* $\mathsf{Uncross}((f, i, c), `U = V')$ *is sound.*

It is complete, to be more precise, if $u = v$ *has a non-empty solution* S *then for appropriate non-deterministic choices the returned equation* $u' = v'$ *has a non-empty solution* S' *such that* $S'(u') = S(u)$ *and there is no crossing father-*i*-leaf subpattern* (f, i, c) *with respect to* S'*.*

Proof. The proof is similar as in the case of Lemma **??**, however, some details are different so it is supplied.

By iterative application of Lemma 92 we obtain that $\mathsf{Uncross}((f, i, c), `U = V')$ is sound.

Concerning the second part of the lemma, we proceed as in Lemma **??**: let $\mathsf{Uncross}((f, i, c), `U = V')$ always make the non-deterministic choices according to the S: we replace x with c when $S(x) = c$ and when we pop down $f(x_1, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m-1})$ from X then indeed f is the last letter of $S(X)$ and $\bullet$ labels the i-th child of f. We define the new substitution S':

- The values on old variables do not change, i.e. $S'(x) = S(x)$ for each variable x present in the context equation both before and after $\mathsf{Uncross}$.
- For a context variable X from which we did not pop a letter we set $S'(X) = S(X)$.
- For X from which $\mathsf{Uncross}$ popped down $f(x_1, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m-1})$ let $S(X) = sf(t_1, \ldots, t_{i-1}, \bullet, t_i, \ldots, t_{m-1})$ (such a representation is possible as $\mathsf{Uncross}$ guesses according to S). Then we define $S'(X) = s$ and $S'(x_j) = t_j$ for $j = 1, \ldots, i-1, i+1, \ldots, m$. Note that when s is a parameter then X is removed from the equation.
- For x that popped-up a constant we do not need to define $S(x)$ as it is no longer in the context equation.

It is easy to verify that indeed in each case the defined S' is a solution of the obtained equation $u' = v'$ and $S'(u') = S(u)$, as claimed.

So suppose that there is a crossing father-i-leaf subpattern (f, i, c) with respect to the S', i.e. one of the (CFL1)–(CFL3) holds. Note that in (CFL1) and (CFL3) there is a variable y such that $S'(y) = c$, however, by our assumption that $\mathsf{Uncross}$ always makes the choice according to the S each such a variable y was replaced with c in the context equation in line 3 or line 12. So it remains to consider the (CFL2).

So let X be as in (CFL2), i.e. the last letter of $S'(X)$ is f, the $\bullet$ is its ith child and Xc is a subpattern in $u = v$. Consider, whether X popped down a letter or not:

X popped a letter down Then for each occurrence of subpattern Xt in the context equation, the first letter of t is always some g such that $\mathrm{ar}(g) \geq 1$ (as there was no way to change this), this is a contradiction with the assumption that Xc is a subpattern in the equation.

X did not pop a letter down Consider the occurrence of a subpattern Xc. Then c was there when we decided not to pop down a letter from X in line 6. Then $\mathsf{Pop}((f, i, c), `U = V')$ should have popped the last letter of f from X, as in line 6 we were supposed to guess according to S, contradiction. □

20.7 Uncrossing patterns

We can state a general lemma about uncrossing.

Lemma 97. Uncross *is sound and complete; to be more precise, for a pattern p if $u = v$ has a non-empty solution S then for appropriate non-deterministic choices the returned equation $u' = v'$ has a non-empty solution S' such that $S'(u') = S(u)$ and p is a non-crossing subpattern with respect to S'.*

20.8 The algorithm

Now we are ready to describe the whole algorithm for testing the satisfiability of context equations.

As a preprocessing, we investigate the input signature Σ: let $k \geq 2$ be the maximal arity of letters in the equation. Let Σ', called *trimmed signature*, be the signature consisting of each letter present in the equation and additionally one letter of each arity at most k that is not present in the equation (take letters from the input signature, when possible, take fresh letters otherwise). We use the trimmed signature instead of the original one, that is, we consider the input equation over this signature; in particular, we use the notion of trimmed signature only when we emphasize it. Note that this allows bounding k, even if the original signature was infinite.

We first present a simplified variant of the algorithm ContextEqSatSimp, which at each step extends the signature by the letters created during the compression steps. Many properties are easier to explain for such simplification. Only afterwards it is explained how to ensure that the size of the signature is bounded; for such algorithm ContextEqSat we can show termination.

Algorithm 14 PreProc($`U = V`, \Sigma$) Preprocessing of the signature

```
1: Σ' ← letters in u = v
2: let k ← maximal arity of letters in Σ'
3: for i ← 0 .. k do
4:     if Σ' does not have a letter of arity i then
5:         f_i ← a letter of arity i          ▷ Choose letter from Σ' or Σ, when possible
6:         Σ' ← Σ' ∪ f_i
7: return Σ'
```

In its main part ContextEqSatSimp iterates the following operation it identifies a pattern to compress, i.e. it chooses to perform one of the compressions: ab compression, a-chain compression or (f, i, c) compression, where a, b, c, f are letters of appropriate arity. It then guesses, whether this pattern is crossing or not. If so, it performs the appropriate uncrossing. Then it performs the compression and adds the new letter (or letters, for chains compression) to Σ.

The extended algorithm ContextEqSat works in the same way, except that at the beginning of each iteration it removes from the signature the letters that are neither from the original (trimmed) signature neither are present in the current context equation; such a signature is called *equation's signature*.

The properties of ContextEqSatSimp and ContextEqSat are summarised below

Algorithm 15 ContextEqSatSimp($`U = V`, \Sigma$) Checking the satisfiability of a context equation $u = v$

```
1: Σ ← PreProc(`U = V`, Σ)
2: while |u| > 1 or |v| > 1 do
3:     choose a subpattern to compress, all letters in Σ
4:     if a-chain compression was chosen then
5:         if a has crossing chains then                          ▷ Guess
6:             CutPrefSuff(a, `U = V`)
7:         PattCompNCr(a, `U = V`)
8:         add letters representing compressed subpatterns to Σ
9:     if ab compression was chosen then                          ▷ Proceed similarly
10:    if (f, i, c) compression was chosen then                   ▷ Proceed similarly
11: Solve the problem naively          ▷ With sides of size 1, the problem is trivial
```

Theorem 98. ContextEqSatSimp *and* ContextEqSat *store an equation of length* $\mathcal{O}(n^2k^2)$*, where* n *is the size of the input equation and* k *the maximal arity of symbols from the input signature. They non-deterministically solve context equation, in the sense that:*

- *if the input equation is not-satisfiable then they never return 'YES';*
- *if the input equation is satisfiable then for some nondeterministic choices in* $\mathcal{O}(n^3k^3 \log N)$ *phases it returns 'YES', where* N *is the size of size-minimal solution.*

Clearly, those algorithms are sound, as a composition of sound procedures.

As a corollary we get an upper bound on the computational complexity of context unification.

Theorem 99. *Context unification is in* PSPACE*.*

The proofs of both theorems are postponed.

20.9 Analysis of the algorithm

The analysis focus on several points. Firstly, we show that we can trim the input signature, see Section 20.9.1, without affecting the satisfiability. Then we briefly mention the bounds on the exponent of periodicity, which helps to bound the space usage of the chain compression, see Section 20.9.2. Then, in Section 20.9.3 we give a bound on the number of occurrences of variables in the equation and show some consequences of that. As our main task, we investigate the space consumption of our algorithm, see Section 20.9.4. Lastly, we show that we can in fact work with solutions over the equation's signature, i.e. those containing only letters present in the (trimmed) input signature and letters in the current equation. In particular we do not need to store letters introduced as a result of compressions. This is done in Section 20.9.5.

20.9.1 Input signature

Trimming of signature does not affect the satisfiability and the needed space.

Lemma 100. *Consider a context equation $u = v$ over a signature Σ that contains a constant and a letter of arity at least 2. Let Σ' be the trimmed signature. Then $u = v$ has a solution over Σ if and only if it has a solution over Σ'. Furthermore, the size of the instance and of the smallest solution increase at most twice.*

Proof. Suppose that there is a solution S over Σ. We define S' over Σ'; for simplicity, denote by f_i a letter in Σ of arity i, for each $i = 0, 1, \ldots, k$. Fix a letter g in $\Sigma \setminus \Sigma'$, consider its arity.

$i = \mathrm{ar}(g) \leq k$ We replace each g in each $S(\alpha)$by f_i, obtaining S'. Since g does not occur in the equation, each g in $S(u)$ and $S(v)$ comes either from some $S(\alpha)$and so it was replace with f_i and so $S'(u) = S'(v)$.

$i = \mathrm{ar}(g) > k$ We replace each term $g(t_1, t_2, \ldots, t_i)$ by $f_2(t_1, (f_2(t_2, (\ldots (f_2(t_{m-1}, t_m)) \ldots))))$; note that f_2 is available, as $k \geq 2$. Again, as g does not occur in the equation, each of its occurrences in $S(u)$ and $S(v)$ comes from some $S(\alpha)$and those were replaced in the same way, so S' is a solution of $u = v$.

Iterating over all $g \in \Sigma' \setminus \Sigma$ in $S(u)$ yields a new solution, which is over Σ'. Concerning the size, note that for ℓ-ary function symbol we introduce at most ℓ new letters. The sum of arities of all occurrences of letters in $S(u)$ is $|S(u)| - 1$, thus we at most double the solution and the size of the size-minimal solution. Concerning the size of the instance, the equation is unchanged but we need to store additional letters. We introduce at most k new letters and the equations has a letter of arity k, so has size at least k. So we at most double the size of the instance. a similar argument

In the other direction, suppose that there is a solution over Σ'. Let us construct solution over Σ: let c' be a constant in Σ and f' a function of arity k in Σ. Then we replace each c in $S(x)$ and $S(X)$ by c' and each $f(t_1, t_2, t_2, \ldots, t_m)$, where $m \leq k$ by $f'(t_1, t_2, \ldots, t_m, c, \ldots, c)$. In the same way as above we can show that indeed such a substitution is a solution of the equation. □

We additionally show a simple observation that the maximal arity of letters in the signature considered by ContextEqSatSimp does not change. Thus, in the following, we shall just use 'k' to denote this value.

Lemma 101. *During* ContextEqSatSimp *(*ContextEqSat*) the maximal arity of letters in the signature does not change.*

Proof. Let k be the initial value of maximal arity of letters in the equations' signature, which is the same as for the input (trimmed) signature. Clearly, it cannot decrease, as all letters of the input signature are counted in.

It cannot increase either: all letters, on which we perform compression, have arity at most k and the compression operations do not increase the arity of letters. □

20.9.2 Exponent of periodicity

The following lemma shows that the size of the a-chains can be limited in case of size-minimal solutions.

Lemma 102 (Exponent of periodicity bound [57])**.** *Let S be a size-minimal solution of a context equation $u = v$ (for a signature Σ). Suppose that $S(X)$ (or $S(x)$) can be written as $ts^m t'$, where t, s, t' are 1-patterns (or t' is a ground term, respectively). Then $m = 2^{\mathcal{O}(|u|+|v|)}$.*

We use Lemma 102 only for the case when s is a unary letter, for which the proof simplifies significantly and is essentially the same as in the case of word equations [23] (which is a simplification of the general bound on the exponent of periodicity by Kościelski and Pacholski [26]).

Note that bound applies to *every* signature: given an equation, we can change the signature and the bound remains the same.

20.9.3 Occurrences of variables

In contrast to the recompression-based algorithm for word equations, ContextEqSat introduces new variables and their occurrences to the equation (when Uncross pops down a letter of arity greater than 1). At first it seems like a large issue, as the number of letters introduced to the equation in one phase depends on the number of term variables. However, we are able to bound the number of such term variables at any time by $n(k-1)$; recall that k is the maximal arity of letters in the signature, this is the place in which we essentially use that k is bounded. To this end, we need some definitions: we say that a variable x_i is *owned* by a context variable X if x_i occurred in the equation when X popped a letter down. A particular occurrence of x_i in the equation is *owned* by the occurrence of the context variable that introduced it. When a context variable X is removed from the equation the term variables it owns get *disowned* (and particular occurrences of those term variable are also disowned).

We show that each context variable owns at most $k-1$ term variables. Using this claim we can bound (in terms of n and k) the number of occurrences of term variables in the equation and the number of letters popped during the uncrossing.

Lemma 103. *Every context variable X present in $u = v$ owns at most $k-1$ term variables. Furthermore, if n_1 is the initial number of context variables, then the total number of owned and disowned term variables is $n_1(k-1)$. In particular, there are at most $n(k-1)$ occurrences of term variables in $u = v$.*

Note that the upper bound on the number of term variables *does not* depend on the non-deterministic choices of ContextEqSat.

Proof. Given an occurrence of a subterm Xt we say that this occurrence of X dominates the occurrences of term variables in t.

We show by induction two technical claims:

1. For every occurrence of a variable X the multiset of term variables, whose occurrences it owns, is the same.

2. Each occurrence of X dominates its owned occurrences of term variables.

The subclaim 1 is trivial: at the beginning, there are no owned term variables. When we introduce new X-owned term variables, we replace each X with the same $Xf(x_1, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m-1})$, in particular the set of X-owned term variables for each occurrence of X is increased by $\{x_1, \ldots, x_{m-1}\}$. When we remove occurrences of x, we remove them all at the same time. Which ends the induction.

Concerning the subclaim 2, this vacuously holds for the input instance, which yields the induction base. For the induction step, consider now the operation performed on the context equation. Any subterm compression is performed only on letters, so it cannot affect the domination. When we pop the letters from a variable x, we replace x

with ax (or remove x altogether), so this also does not affect the domination. Similarly, when we pop letters from context variables, we either replace X with aX or X with $Xf(x_1, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m-1})$, in both cases the domination of the old variables is not affected and in the last case the new variables $x_1, \ldots, x_{m-1}$ owned by this particular occurrence of X are indeed dominated by this occurrence of X.

Using those two subclaims we now show that if during Uncross X pops down a letter, then X does not own any variables. Suppose that X pops down a letter. Then in $U = V$ there is a subtree Xc for a constant c. Suppose that X owned a variable x before popping down the letter. Then by subclaim 1 the occurrence of X which is applied on c also owns occurrence of x and by 2 this occurrence is dominated by its owning occurrence of X, which is not possible, as this owning occurrence of X is part of the term Xc. As a consequence, each occurrence of a context variable owns at most $k-1$ occurrences of variables.

Now, concerning the number of term variables: let the initial number of variables (not owned nor disowned) and context variables be n_0 and n_1, where $n_0 + n_1 \leq n$. Suppose that at some point there are $n_1' \leq n_1$ context variables occurrences. Since we never introduce context variables, there are at most $n_1'(k-1)$ owned variables' occurrences, and at most $(n_1 - n_1')(k-1)$ disowned ones. This yields a bound of $n_1(k-1)$ on the number of occurrences of variables that are owned or disowned. Additionally, there are n_0 occurrences that are neither owned, nor disowned (those are the occurrences of variables that were present in the input equation). In total

$$n_1'(k-1) + (n_1 - n_1')(k-1) + n_0 = n_1(k-1) + n_0 \leq n(k-1) \ ,$$

with the inequality following from $k \geq 2$ and $n_0 + n_1 \leq n$. □

The bound on the number of occurrences of term variables allows a bound on the number of different crossing subpatterns.

Lemma 104. *For an equation $u = v$ during* ContextEqSat *and its solution S there are at most $n(k+1)$ different crossing subpatterns.*

Proof. Let n_1 and n_0 be the initial number of context variables and variables, note that $n_0 + n_1 \leq n$. By Lemma 103 the total number of variables in $u = v$ is at most $n_0 + n_1(k-1)$

Each context variable introduces at most two different crossing patters: one for its top letter and one for its last letter. A variable can introduce at most one crossing subpattern. So the number of such subpatterns is at most

$$2n_1 + n_0 + n_1(k-1) = n_0 + n_1(k+1) \leq n(k+1) \ ,$$

as claimed. □

As another consequence, we can also limit the number of new letters introduced during the uncrossing.s

Lemma 105. *Uncrossing and compression of a subpattern introduces at most $n(2k+1)$ letters to the equation.*

Proof. Consider first the pair compression. At most one letter is popped up and down from each of the context variable, which gives $2n$ letters. Also, at most one letter is popped up from each variable, and there are at most $n(k-1)$ variables, see Lemma 103, this yields $n(k-1)$ new letters. In total: $2n + n(k-1) = n(k+1) \leq n(2k-1)$.

The analysis is the same for uncrossing a-chains, except that instead of one letter we pop whole a-prefixes and a-suffixes. But they are immediately replaced with single letters, so the same estimation holds.

For the father-i-leaf subpatterns, we only pop up unary letters from variables, which gives $n(k-1)$ letters. We also pop down at most a single letter f from each context variable, together with up to $k-1$ new variables, which may be immediately after turned into letters, which yields another nk letters. So, $n(2k-1)$ letters in total. □

20.9.4 Size bounds

We can now show the crucial lemma: if a solution is satisfiable, then for some non-deterministic choices the obtained equation is also satisfiable and its size does not grow. We begin with showing the bound when the signature is not restricted and explain in the next section, that those results hold also for simple signatures.

Lemma 106. *Suppose that the equation $u = v$ has a solution S (over a signature Γ) for which there is a non-crossing subpattern with explicit occurrence in $u = v$. Then after compressing this subpattern the obtained equation is satisfiable, is smaller and has a smaller solution (over the signature Γ expanded by the letter replacing the compressed subpatterns).*

Proof. We perform the subpattern compression for appropriate subpattern. The obtained equation is clearly smaller (as there is at least one occurrence of the compressed subpattern). From Lemma 94 the obtained equation has a solution S' such that $S'(u')$ is smaller than $S(u)$. □

A similar statement can be shown also for uncrossing and compression of crossing subpatterns.

Lemma 107. *Suppose that the equation $u = v$ has a solution S (over a signature Γ) for which there is no non-crossing subpattern with explicit occurrence in $u = v$. Then there is a crossing subpattern (with respect to S) such that for appropriate non-deterministic choices after uncrossing and compressing it the equation has a smaller solution (over a signature Γ expanded by the new letters that replaced the compressed subpatterns). Additionally, if the equation has at least $48n^2k^2$ letters then for those nondeterministic choices the obtained equation has less letters.*

Proof. Take any crossing subpattern. Uncross it. By Lemma 97 the obtained equation has a solution of the same size, for which the subpattern is non-crossing. Compress this subpattern. By Lemma 106 the obtained equation has smaller solution. Note that this argument holds for the compression of *any* subpattern, as long as it has occurrences in the solution; the claim on the signature follows also from Lemma 106.

Let us move to the second claim of the lemma.

If $u = v$ has more than $n^2(2k+1)(k+1)$ occurrences of constants then it has the same amount of occurrences of father-leaf subpatterns. As there are at most $n(k+1)$ different crossing subpatterns, see Lemma 104, one of them has more than

$$\frac{n^2(2k+1)(k+1)}{n(k+1)} = n(2k+1)$$

occurrences. We uncross it and compress it. The uncrossing introduces at most $n(2k+1)$ new letters, see Lemma 105. On the other hand, at least $n(2k+1)+1$ letters are removed, and so the equation gets smaller.

If $u = v$ has at most $n^2(2k+1)(k+1)$ constants then it has at most

$$n^2(2k+1)(k+1) + n(k-1) < n^2(4k)(k+1)$$

symbols of arity 0 (the other $n(k-1)$ are the variables, see Lemma 103). Hence it also has at most this amount of nodes of arity at least 2, so all remaining nodes have arity at most 1 and at most n of them are context variables. So there are at least

$$n^2(24k)(k+1) - 2n^2(4k)(k+1) - n > n^2(15k)(k+1)$$

unary letters in the equation.

We can similarly estimate the amount of chains: each maximal chain ends with a node of arity different than 1, so there are at most

$$\underbrace{n^2(4k)(k+1)}_{\text{symbols of arity 0}} + \underbrace{n^2(4k)(k+1)}_{\text{symbols of arity at least 2}} + \underbrace{n}_{\text{context variables}} < n^2(9k)(k+1)$$

different chains.

If a chain is not a single letter, then each of its letter is covered by an occurrence of some a-maximal chain (of length greater than 1) or ab pair; by the assumption each ab pair is a crossing pair and each a has a crossing blocks. On the other hand, by Lemma 104, there are at most $n(k+1)$ different crossing subpatterns. So occurrences of one of those subpatterns cover at least

$$\frac{n^2(15k)(k+1) - n^2(9k)(k+1)}{n(k+1)} > 2n(2k+1)$$

letters. We compress this subpattern. The rest of the analysis follows as in the case of compression of father-i-leaf subpatterns, with one exception: when we pop the a-prefixes and suffixes, we introduce perhaps very long chains to the equation. They are immediately replaced with a single letter afterwards, so there is no problem with this. Moreover, any as that are part of the compressed chain and were in the equation before popping are compressed. So in total we introduce 1 letter and remove all explicit letters that are part of the compressed chains. □

A similar claim can be shown also for the size of the solution

Lemma 108. *Suppose that the equation $u = v$ has a solution S (over a signature Γ) for which there is no non-crossing subpattern with explicit occurrence in $u = v$. Then there is a crossing subpattern (with respect to S) such that for appropriate non-deterministic choices after uncrossing and compressing it the equation is larger by at most $n(2k+1)$ and it has a solution of size at most $\left(1 - \frac{1}{6n(k+1)}\right)|S(u)|$ (over a signature Γ plus the letters replacing the compressed subpatterns).*

Proof. The bound on the number of introduced letters follow from Lemma 105.

Let n_0, n_1 and n_2 be the number of letters of arity 0, 1 and at least 2 in $S(u)$. If $n_0 \geq \frac{|S(u)|}{6}$ then there are at least $\frac{|S(u)|}{6}$ different occurrences of father-leaf patterns. By Lemma 104 there are at most $n(k+1)$ different crossing subpatterns, see Lemma 104,

and so one of them has at least $\frac{|S(u)|}{6n(k+1)}$ occurrences. Its compression removes at least $\frac{|S(u)|}{6n(k+1)}$ letters from the equation. On the level of the equation we first need to uncross this subpattern and then compress it, the rest of the analysis is as in Lemma 107.

So suppose that $n_0 < \frac{|S(u)|}{6}$, so also $n_2 < \frac{|S(u)|}{6}$ and so $n_1 \geq \frac{2|S(u)|}{3}$. Except perhaps the chains of length 1, each letter in a unary chain is covered by some ab pair or a-chain of length greater than 2. Since each chain ends in a letter of arity other than 1, there are at most $n_0 + n_2 < \frac{|S(u)|}{3}$ chains and so at least $\frac{|S(u)|}{3}$ letters are covered. As there at most $n(k+1)$ different crossing subpatterns, one of them covers at least $\frac{|S(u)|}{3n(k+1)}$ letters and so its compression removes at least $\frac{|S(u)|}{6n(k+1)}$ letters from the solution. The rest of the analysis follows as in the previous case. □

We can now show the proof of the main theorem (Theorem 98) for the case of ContextEqSatSimp.

proof of Theorem 98 for ContextEqSatSimp . Suppose that we are given a satisfiable equation $u = v$. By Lemma 100 the equation is satisfiable also over the trimmed signature.

During the algorithm we ensure that the equation has at most $48n^2k^2 + n(2k+1)$ letters over the signature consisting of the trimmed signature and all letters introduced during the ContextEqSatSimp.

During the algorithm, if there is a non-crossing subpattern for some length-minimal solution, we choose it for compression (as a non-deterministic guess). This reduces the number of letters in the equation, see Lemma 106, so there are at most $48n^2k^2+n(2k+1)$ such compressions in a row. Note that each consecutive equation has smaller minimal solution, again by Lemma 106.

If there are only crossing pairs for the size-minimal solution (say S), then there are two different behaviours, depending on the size of the equation. If the equation has more than $48n^2k^2$ letters then we choose a crossing subpattern (for S) for uncrossing and compression according to Lemma 107. After uncrossing it and the compression the size of the equation and size-minimal solution decrease, see Lemma 107.

If the equation has at most $48n^2k^2$ letters then choose according to Lemma 108. Thus the size of the size-minimal solution decreases by a fraction $1 - \frac{1}{6n(k+1)}$ and the size of the equation increases by at most $n(2k+1)$.

In this way the number of letters in the equation is always at most $48n^2k^2+ n(2k+1)$: we can only increase it by compressing pairs chosen according to Lemma 108 or Lemma 107, in each case by at most $n(2k+1)$ letters. However, if the equation has more than $48n^2k^2$ then we choose the latter and the Lemma 107 guarantees that the size of the equation does not increase.

Concerning the number of phases: each compression according to Lemma 108 reduces the size of the length-minimal solution by a fraction $\left(1 - \frac{1}{6n(k+1)}\right)$, so after $6n(k+1)$ such compressions the size of the length-minimal (simple) solution reduces by a constant fraction, so there are only $\mathcal{O}(nk \log N)$ such compressions, where N is the size of the size-minimal solution. Consider now, how many other compression can there be between two compressions according to Lemma 108? Each other compression reduces the size of the equation by 1 and so there can be at most $48n^2k^2+ n(2k+1)$ of them in-between two such compressions. So there are $\mathcal{O}(n^3k^3 \log N)$ compression steps in total. □

20.9.5 Simple solutions

We now show that the ContextEqSat does not loose solutions by restricting itself to the equations' signature; moreover, the size of the size-minimal solution remains the same.

Lemma 109. *Let k be the maximal arity of symbols in the trimmed signature. Consider any equation obtained during* ContextEqSat*. If it has a solution S over a signature of arity k then it has a solution S' over the equations' signature; moreover, size of S' is not larger then the size of S.*

Proof. The proof follows a similar replacement schema as in the case of Lemma 100: take any signature Γ' and let Γ be the simple signature. If $S(u)$ uses a letter $g \in \Gamma' \setminus \Gamma$ then it must be used inside a substitution $S(X)$. We can replace all occurrences of g with a letter $f_i \in \Gamma'$ of the same arity; this is possible as there is a letter of each arity up to k in Γ and the ariyt of Γ' is bounded by k. The obtained substitution is a solution and it has the same size as S, yielding the claim. □

This allows us to show that proof of Theorem 98 in case of ContextEqSat.

proof of Theorem 98 for ContextEqSat . The proof of the Theorem is the same as in the case of ContextEqSatSimp, with one exception: we ensure that the kept solution is over the equation's signature. (Note that Lemma 106–108 apply to this setting). After a compression (and perhaps the earlier uncrossing) we get an equation over a signature extended by some letters, and a solution S' smaller than a solution S of the previous equation. It could be that S' is not over equation's signature, but by Lemma 89 it is over a signature of maximal arity at most k and so from Lemma 109 there is a simple solution whose size is at most the size of S', so in particular smaller than S.

The rest of the proof is identical, as we rely only on local choices of pair to compress for some solution. □

This allows us to show also the proof of Theorem 99.

proof of Theorem 99. By Theorem 98 the (non-deterministic) algorithm ContextEqSat stores an equation of size $\mathcal{O}(n^2k^2)$, which is stored in polynomial space. The maximal stored equation's signature has size of the equation plus the size of the trimmed signature, which is linear, see Lemma 100. The additional used space is proportional to the size of the equation, except the space needed to store the lengths of the a-chains. But this is at most polynomial, see Lemma 102. Thus the whole space usage is polynomial.

It cannot be that during the computation we reach the same equation (which necessarily has the same equations' signature): each performed compression operation shortens the length of the length-minimal solution, see Lemma 106, 107 and 108. And the size of the size-minimal solution is the same.

Hence after an appropriate number of steps during which we did not accept we can reject the input.

Lastly, by Savitch Theorem the non-deterministic polynomial space algorithm can be determinised, using at most quadratically more space. □

References

[1] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, William F. Smyth, German Tischler, and Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.*, 45(1):5, 2012.

[2] Stephen Alstrup, Gerth S. Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA*, pages 819–828, 2000. ISBN 0-89871-453-2. doi:doi.acm.org/10.1145/338219.338645.

[3] Witold Charatonik and Leszek Pacholski. Word equations with two variables. In Habib Abdulrab and Jean-Pierre Pécuchet, editors, *IWWERT*, volume 677 of *LNCS*, pages 43–56. Springer, 1991. ISBN 3-540-56730-5. doi:10.1007/3-540-56730-5_30.

[4] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

[5] Gang Chen, Simon J. Puglisi, and William F. Smyth. Fast and practical algorithms for computing all the runs in a string. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 307–315. Springer, 2007.

[6] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7. URL `http://dx.doi.org/10.1016/S0019-9958(86)80023-7`.

[7] Hubert Comon. Completion of rewrite systems with membership constraints. Part I: Deduction rules. *J. Symb. Comput.*, 25(4):397–419, 1998. doi:10.1006/jsco.1997.0185.

[8] Hubert Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998. doi:10.1006/jsco.1997.0186.

[9] Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *DCC*, pages 482–488. IEEE Computer Society, 2008.

[10] Volker Diekert and Markus Lohrey. Existential and positive theories of equations in graph products. *Theory Comput. Syst.*, 37(1):133–156, 2004. doi:10.1007/s00224-003-1110-x. URL `http://dx.doi.org/10.1007/s00224-003-1110-x`.

[11] Volker Diekert, Claudio Gutiérrez, and Christian Hagenah. The existential theory of equations with rational constraints in free groups is PSPACE-complete. *Inf. Comput.*, 202(2):105–140, 2005.

[12] Volker Diekert, Artur Jeż, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. In Edward A. Hirsch, Sergei O. Kuznetsov, Jean-Éric Pin, and Nikolay K. Vereshchagin, editors, *CSR*, volume 8476 of *LNCS*, pages 1–15. Springer, 2014. doi:10.1007/978-3-319-06686-8_1. URL `http://dx.doi.org/10.1007/978-3-319-06686-8_1`.

[13] Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *LNCS*, pages 408–419. Springer, 2004. ISBN 3-540-22849-7. doi:10.1007/978-3-540-27836-8_36.

[14] Robert Dąbrowski and Wojciech Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011. doi:10.1007/s00453-009-9375-3.

[15] William M. Farmer. Simple second-order languages for which unification is undecidable. *Theor. Comput. Sci.*, 87(1):25–41, 1991. doi:10.1016/S0304-3975(06)80003-4.

[16] Adria Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context unification with one context variable. *J. Symb. Comput.*, 45(2):173–193, 2010. doi:10.1016/j.jsc.2008.10.005.

[17] Adria Gascón, Ashish Tiwari, and Manfred Schmidt-Schauß. One context unification problems solvable in polynomial time. In *LICS*, pages 499–510. IEEE, 2015. ISBN 978-1-4799-8875-4. doi:10.1109/LICS.2015.53. URL `http://dx.doi.org/10.1109/LICS.2015.53`.

[18] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988. doi:10.1137/0401044. URL `http://dx.doi.org/10.1137/0401044`.

[19] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981. doi:10.1016/0304-3975(81)90040-2.

[20] Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC*, pages 133–142. IEEE, 2013. ISBN 978-1-4673-6037-1.

[21] Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC 2014*, pages 163–172. IEEE, 2014. doi:10.1109/DCC.2014.62. URL `http://dx.doi.org/10.1109/DCC.2014.62`.

[22] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027. URL `http://dx.doi.org/10.1016/j.tcs.2015.05.027`.

[23] Artur Jeż. Recompression: a simple and powerful technique for word equations. *Journal of the ACM*, 2015. ISSN 0004-5411/2015. doi:10.1145/2743014. URL `http://dx.doi.org/10.1145/2743014`.

[24] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[25] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. ISBN 978-3-642-38904-7, 978-3-642-38905-4.

[26] Antoni Kościelski and Leszek Pacholski. Complexity of Makanin's algorithm. *J. ACM*, 43(4):670–684, 1996.

[27] Markku Laine and Wojciech Plandowski. Word equations with one unknown. *Int. J. Found. Comput. Sci.*, 22(2):345–375, 2011. doi:10.1142/S0129054111008088.

[28] J. L. Lambert. Une borne pour les générateurs des solutions entières positives d'une équation diophantienne linéaire. *Compte-rendu de L'Académie des Sciences de Paris*, 305(1):39–40, 1987.

[29] Jordi Levy. Linear second-order unification. In Harald Ganzinger, editor, *RTA*, volume 1103 of *LNCS*, pages 332–346. Springer, 1996. ISBN 3-540-61464-8. doi:10.1007/3-540-61464-8_63.

[30] Jordi Levy and Jaume Agustí-Cullell. Bi-rewrite systems. *J. Symb. Comput.*, 22(3): 279–314, 1996. doi:10.1006/jsco.1996.0053.

[31] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Inf. Comput.*, 159(1–2):125–150, 2000. doi:10.1006/inco.2000.2877.

[32] Jordi Levy and Mateu Villaret. Linear second-order unification and context unification with tree-regular constraints. In Leo Bachmair, editor, *RTA*, volume 1833 of *LNCS*, pages 156–171. Springer, 2000. ISBN 3-540-67778-X. doi:10.1007/10721975_11.

[33] Jordi Levy and Mateu Villaret. Currying second-order unification problems. In Sophie Tison, editor, *RTA*, volume 2378 of *LNCS*, pages 326–339. Springer, 2002. ISBN 3-540-43916-1. doi:10.1007/3-540-45610-4_23.

[34] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. The complexity of monadic second-order unification. *SIAM J. Comput.*, 38(3):1113–1140, 2008. doi:10.1137/050645403. URL `http://dx.doi.org/10.1137/050645403`.

[35] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. On the complexity of bounded second-order unification and stratified context unification. *Logic Journal of the IGPL*, 19(6):763–789, 2011. doi:10.1093/jigpal/jzq010.

[36] Yury Lifshits. Processing compressed texts: A tractability border. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 228–240. Springer, 2007. ISBN 978-3-540-73436-9. doi:10.1007/978-3-540-73437-6_24.

[37] Gennadií Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).

[38] Gennadií Makanin. Equations in a free group. *Izv. Akad. Nauk SSR*, Ser. Math. 46: 1199–1273, 1983. English transl. in Math. USSR Izv. 21 (1983).

[39] Gennadií Semyonovich Makanin. Decidability of the universal and positive theories of a free group. *Izv. Akad. Nauk SSSR*, Ser. Mat. 48:735–749, 1984. In Russian; English translation in: *Math. USSR Izvestija, 25*, 75–88, 1985.

[40] Jerzy Marcinkowski. Undecidability of the first order theory of one-step right ground rewriting. In Hubert Comon, editor, *RTA*, volume 1232 of *LNCS*, pages 241–253. Springer, 1997. doi:10.1007/3-540-62950-5_75.

[41] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

[42] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. A uniform approach to underspecification and parallelism. In Philip R. Cohen and Wolfgang Wahlster, editors, *ACL*, pages 410–417. Morgan Kaufmann Publishers / ACL, 1997.

[43] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. On equality up-to constraints over finite trees, context unification, and one-step rewriting. In William McCune, editor, *CADE*, volume 1249 of *LNCS*, pages 34–48. Springer, 1997. ISBN 3-540-63104-6. doi:10.1007/3-540-63104-6_4.

[44] S. Eyono Obono, Pavel Goralcik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *MFCS*, volume 841 of *LNCS*, pages 336–341. Springer, 1994. ISBN 3-540-58338-6. doi:10.1007/3-540-58338-6_80.

[45] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *CPM*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011. ISBN 978-3-642-21457-8.

[46] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725. ACM, 1999.

[47] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. doi:10.1145/990308.990312.

[48] Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of word equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998. doi:10.1007/BFb0055097.

[49] RTA problem list. Problem 90. http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html, 1990.

[50] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

[51] Aleksi Saarela. On the complexity of Hmelevskii's theorem and satisfiability of three unknown equations. In Volker Diekert and Dirk Nowotka, editors, *Developments in Language Theory*, volume 5583 of *LNCS*, pages 443–453. Springer, 2009. ISBN 978-3-642-02736-9. doi:10.1007/978-3-642-02737-6_36.

[52] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005. doi:10.1016/j.jda.2004.08.016.

[53] Manfred Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universität, 1994.

[54] Manfred Schmidt-Schauß. A decision algorithm for distributive unification. *Theor. Comput. Sci.*, 208(1–2):111–148, 1998. doi:10.1016/S0304-3975(98)00081-4.

[55] Manfred Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Log. Comput.*, 12(6):929–953, 2002. doi:10.1093/logcom/12.6.929.

[56] Manfred Schmidt-Schauß. Decidability of bounded second order unification. *Inf. Comput.*, 188(2):143–178, 2004. doi:10.1016/j.ic.2003.08.002.

[57] Manfred Schmidt-Schauß and Klaus U. Schulz. On the exponent of periodicity of minimal solutions of context equation. In *RTA*, volume 1379 of *LNCS*, pages 61–75. Springer, 1998. ISBN 3-540-64301-X. doi:10.1007/BFb0052361.

[58] Manfred Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. *J. Symb. Comput.*, 33(1):77–122, 2002. doi:10.1006/jsco.2001.0438.

[59] Manfred Schmidt-Schauß and Klaus U. Schulz. Decidability of bounded higher-order unification. *J. Symb. Comput.*, 40(2):905–954, 2005. doi:10.1016/j.jsc.2005.01.005.

[60] Klaus U. Schulz. Makanin's algorithm for word equations—two improvements and a generalization. In Klaus U. Schulz, editor, *IWWERT*, volume 572 of *LNCS*, pages 85–150. Springer, 1990. ISBN 3-540-55124-7. doi:10.1007/3-540-55124-7_4.

[61] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 30–39. ACM, 1978.

[62] Ralf Treinen. The first-order theory of linear one-step rewriting is undecidable. *Theor. Comput. Sci.*, 208(1–2):179–190, 1998. doi:10.1016/S0304-3975(98)00083-8.

[63] Joachim von zur Gathen and Malte Sieveking. A bound on solutions of linear integer equations and inequalities. *Proceedings of AMS*, 72(1):155–158, 1978.

[64] Sergei G. Vorobyov. The first-order theory of one step rewriting in linear Noetherian systems is undecidable. In Hubert Comon, editor, *RTA*, volume 1232 of *LNCS*, pages 254–268. Springer, 1997. ISBN 3-540-62950-5. doi:10.1007/3-540-62950-5_76.

[65] Sergei G. Vorobyov. $\forall\exists^*$-equational theory of context unification is Π^0_1-hard. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *MFCS*, volume 1450 of *LNCS*, pages 597–606. Springer, 1998. ISBN 3-540-64827-5. doi:10.1007/BFb0055810.