Word Equations: lecture notes 2024–25

Artur Jeż

January 31, 2025

Contents

Chapter 1

Word equations: basic notions and results

1.1 Introduction

A word equation consists of a pair (u, v) of words over letters (constants) and variables and a solution is a substitution of the variables by words in letters such that the formal equality $u = v$ becomes a true equality of words (strings).

It is fairly easy to see that WordEquation reduces to Hilbert's 10th Problem, see Exercise 8. Hence in the mid 1960s there was an attempt to prove undecidability of Hilbert 10th Problem via undecidability of word equations. The program failed in the sense that Matiyasevich proved Hilbert's 10th Problem to be undecidable in 1970, but by a completely different method, which employed number theory. On the other hand, in 1977 Makanin showed in his seminal paper [40] that satisfiability of word equations is decidable. Makanin's algorithm became famous since it settled a long standing problem and also because his algorithm had an extremely complex termination proof. Furthermore Makanin extended his results to free groups and showed that the existential and positive theories in free groups are decidable [41, 42].

1.2 Definitions

Definition 1.1 (Alphabet, variables)**.** In context of word equations we usually consider a finite *alphabet* Σ and finite set of *variables* $\mathcal{X}$, which is disjoint with Σ. Elements of Σ are usually denoted by small letters $a, b, c, \ldots$. Elements of $\mathcal{X}$ are usually denoted as $X, Y, Z, \ldots$.

Definition 1.2 (Word equation, systems of word equations)**.** A *word equation* is a pair (u, v), usually written as $u = v$, where $u, v \in (\Sigma \cup \mathcal{X})^*$ is a sequence of letters and variables. A *system of word equations* is a set of word equations, usually denoted as (u_1, v_1), (u_2, v_2), $\ldots$

Definition 1.3 (Substitution, solution, length-minimal solution)**.** Given a set of variables $\mathcal{X}$ and a set of letters Σ, a *substitution* is a morphism $s : \mathcal{X} \to \Gamma^+ \supseteq \Sigma^+$. A substitution is extended to Σ as an identity (so $s(a) = a$ for $a \in \Sigma$) and to $(\Sigma \cup \mathcal{X})^*$ as a homomorphism (so $s(\alpha\beta) = s(\alpha)s(\beta)$ for $\alpha, \beta \in (\Sigma \cup \mathcal{X})^*$).

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

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

Given a solution s for the equation $u = v$ the $s(u)$ is a *solution word* for this solution of the equation.

Note that we do allow that the solution uses letters that are not in the instance, but this is a slight technical detail.

Problem: (Satisfiability of) Word Equations

Input: A system of word equations with variables $\mathcal{X}$ over Σ.

Task: Decide, whether this system has a solution.

Definition 1.4 (Cubic and quadratic systems of equations)**.** 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.

Definition 1.5 (Language Constraints)**.** *Language Constraints* for system of word equations are given as additional constraints of the form $X \in C$ or $X \notin C$, where $X \in \mathcal{X}$ 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 $s(X) \notin C$).

Example 1.1. The equation

$$aXca = abYa$$

has a solution $s(X) = baba$ and $s(Y) = abac$.

The equation

$$aX = Xa$$

has an infinite number of solutions, each is of the form $s(X) = a^k$ for some $k > 0$.

The equation

$$aXb = Y$$

has a solution $s(X) = w$ and $s(Y) = awb$ for each word w.

The equation

$$aXYX^3 = XYaY^2$$

has infinite number of solutions: Since $s(aXY)$ and $s(XYa)$ have always the same length, this is equivalent to a system of equations

$$aXY = XYa \text{ and } X^3 = Y^2 \ .$$

The former has solutions $X = a^n, Y = a^m$ and the latter ensures that $3n = 2m$.

The equation

$$XbaYb = baaababbab$$

has a solution $s(X) = baaa$, $s(Y) = bba$

1.3 Cuts and satisfiability

This section is based on [52].

Definition 1.6 (Cut, touching a cut)**.** A *cut* (in an equation) is a position between two symbols (i.e. letters, variables or '=' sign) or at the beginning or end of the equation. We generalise this notion to a cut for a solution.

A substring in $s(u)$ or $s(v)$ *overlaps* a cut α, if α is within this word. It touches a cut if it overlaps a cut or a cut is directly before or directly after it.

Fact 1.7. *There are $|u| + |v| + 2$ cuts in a word equation $u = v$.*

Definition 1.8. For a function $f : \mathcal{X} \mapsto \mathbb{N}$ a substitution (solution) s is an f-substitution (f-solution), if $|s(X)| = f(X)$ for each variable X.

Definition 1.9. Given a substitution s we say that two positions 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, reflexive and symmetric closure of $\mathcal{R}'$.

Lemma 1.10. *Consider a substitution s and the $\mathcal{R}$ relation, let f be such that s is an f-substitution. Then*

1. *There is an f-solution if and only if no equivalence class contains two positions corresponding to different constants and the sides of the equation have equal lengths when substituted with an f-substitution.*

2. *if s is a solution and there is an equivalence class containing no constants from the equation then it is not length-minimal. Moreover, the symbols at positions in this class can be filled with the same arbitrary string, in particular by ϵ, and the obtained substitution is a solution.*

3. *For any two positions $i\mathcal{R}j$ in an f-solution s' we have $s'(uv)[i] = s'(uv)[j]$.*

Proof. Rather obvious. □

Lemma 1.11. *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.*

Proof. Left as an easy exercise. □

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 $s(v)$).

Those observation lead to two simple algorithms for word equations:

Theorem 1.12. *Given a length function f, deciding, whether there is an f-solution for $u = v$, can be done in* $\mathsf{SPACE}(\log f(uv))$.

In the same space we can verify, whether this is a lenght minimal solution.

Given a bound N, we can verify in $\mathsf{SPACE}(\log(N + |uv|))$, *whether there is a solution such that its solution word is of size at most N.*

Proof. It is enough to verify the conditions from Lemma 1.10.

The computation of the length can be easily done in the given space, by going through the equations and summing the values.

To check that there are no two such positions, fo each two position with different letters we verify, whether they are connected. This complement of this is a reachability problem in a graph with positions as vertices and $\mathcal{R}'$ as an edge relation. Note that the relation is symmetric and symmetric reachability is in $\mathsf{LOGPSPACE}$, in our case the size is $f(uv)$. The complement is in the same class.

To verify that this is a length-minimal solution, we can check, whether there exist a function f' for which there is an f'-solution, but $f'(uv) < f(uv)$. We iterate over possible values of f', which takes $\mathcal{O}(\log f(uv))$ space and each check also takes $\mathcal{O}(\log f'(uv)) \leq \mathcal{O}(\log f(uv))$ space.

The last point is done similarly as in the case above. □

1.4 SLPs and satisfibility

Definition 1.13 (Straight Line Programme, SLP)**.** *Straight Line Programme* (*SLP*) is a CFG in the Chomsky normal form that generates a unique string. The *size* of the SLP is the sum lengths of its right-hand sides and for an SLP $\mathcal{A}$ it is denoted by $|\mathcal{A}|$. The unique word generated by $\mathcal{A}$ is denoted by $\mathrm{val}(\mathcal{A})$.

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 latter condition essentially means that they are in the Chomsky normal form. This increases the size of the SLP only by a constant fraction. With this assumption the size is asymptotically the same as the number of nonterminals: in this case — g.

Note, that an SLP can be seen as a word equation of a very restricted kind.

1.4.1 Equivalence of SLPs

Given two SLPs $\mathcal{A}, \mathcal{B}$ the equivalence problem is the question whether they define the same string, i.e. $\mathrm{val}(\mathcal{A}) = \mathrm{val}(\mathcal{B})$. This can clearly be tested in PSPACE, but in fact can be done in P, which we will learn later on.

1.4.2 Composition systems

In many proofs it is easier to use the 'substring' approach rather than the SLPs. 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; the proof is left as an exercise.

Lemma 1.14. *A composition system of size n can be transformed into an equivalent SLP of size $\mathcal{O}(n^2)$*

Proof. Proof is left as an exercise. This is not the best bound. □

1.4.3 Satisfiability via SLPs

This section is based on [52].

For a cut α by $(\alpha)_k$ as a string extending 2^k positions to the left of α and 2^k to the right of α; if this is not possible, then we take as many letters as possible.

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

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

where $|w_k|, |w'_k| \leq 2^{k-1}$.

By Lemma 1.11, we get that w_k and w'_k are substrings of some $(\beta)_i$ and $(\gamma)_i$. As they are of length 2^{k-1}, so when w_k overlaps β, it is within $(\beta)_k$, so we can in fact take $(\beta)_k$ and $(\gamma)_k$

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 $\mathcal{O}(n^2 \log N)$ for the smallest solution (and so also the same size for each variable). The same argument applies also to each variable

Theorem 1.15. *Given a length-minimal solution of an equation $u = v$ there is a composition system of size $\mathcal{O}(n^2 \log N)$ of a solution word.*

Exercises

Task 1 Show that a satisfiability of a system of word equations is NP-hard already when $\Sigma = \{a\}$.

Hint: This reduces to some other known equations.

Task 2 Show that the satisfiability of word equations is NP-hard when we consider only systems in which every right-hand side does not contain variables.

(Note: it might be easier to show this when we allow also ϵ as a substitution for a variable).

Task 3 Show that the problem of satisfiability of a system of word equations can be reduced to the problem of satisfiability of a single word equation, when we are allowed to add letters to the alphabet. Show the same result also when adding letters is not allowed, but $|\Sigma| \geq 2$.

Task 4 Suppose that s is a length-minimal solution of a word equation $u = v$. Let w be a substring of $s(u)$. Show that w has an occurrence that touches a cut.

Strengthen this for $|w| \geq 2$: in this case w overlaps a cut.

Strengthen this for $w = a \in \Sigma$: in this case a occurs in u or in v.

Conclude that without loss of generality the length-minimal solutions do not use letters outside the alphabet Σ.

Hint: Using the inductive definition of the transitive closure may be helpful.

Task 5 Reduce the satisfiability problem for word equations to the satisfiability problem of cubic word equations, i.e. when each variable occurrs at most three times in the system of word equations.

Task 6 Show that the problem of word equations with context-free constraints is undecidable. Here "context-free constraints" means that we apart from the equations, we allow also condition of the form $X \in L$, where X is any variable and L is a CFG. For s to be a solution we require then that $s(X) \in L$.

Task 7 Show that the Intersection Problem for DFAs

Problem: Non-emptiness of Intersection for DFAs
Input: DFAs (deterministic finite automata) $D_1, D_2, \ldots, D_m$
Task: Decide, whether the intersection of their languages is non-empty

is PSPACE-hard.
Show that this problem is in PSPACE even when we allow NFAs.
Deduce from this that word equations with regular constraints are PSPACE-hard; as in Task 6 the regular constraints mean that apart from the equations, we allow also condition of the form $X \in L$, where X is any variable and L is regular language, given a by a DFA and we require that $s(X) \in L$.

Task 8 (Long: two points) Consider a mapping from $\Sigma = \{a, b\}$ to 2×2 matrices over $\mathbb{N}$, defined as

$$\varphi(a) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } \varphi(b) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} .$$

Extend this to Σ^* as a homomorphism.

Show that for any $w \in \Sigma^*$ its image is a a matrix with a determinant one.

Show that this mapping is injective; to do this, consider, what are the rows of a matrix $\varphi(a) \begin{bmatrix} n & n' \\ m & m' \end{bmatrix}$ and what are the rows of $\varphi(a) \begin{bmatrix} n & n' \\ m & m' \end{bmatrix}$. Deduce from this that looking at the matrix $M_w = \varphi(w)$ we can determine the left-most letter of w by looking at rows of M_w.

Show that if a 2×2 matrix M with determinant 1 and all natural entries can be represented as either $\varphi(a)M'$ or $\varphi(b)M'$, where M' has a determinant 1 and all natural entries. Again: compare the rows.

Deduce from this that φ is an isomorphism between Σ^* and 2×2 matrices with determinant 1 and all entries natural.

Deduce from this that satisfiability of word equation over $\Sigma = \{a, b\}$ reduces to the satisfiability of equations over natural numbers (to do this, represent a $\varphi(X)$ as a matrix of variables representing natural numbers).

Task 9 Show that the composition system of size n can be turned into an SLP of polynomial size. How small you can make the polynomial?

Chapter 2

Satisfiability of word equations in PSPACE

Idea

In Section 1.4.3 we showed that there is an SLP for the length-minimal solution of size $\text{poly}(n, \log N)$. The construction of the SLP was top-down and it required an external bound on the size of the SLP, i.e. N. In this chapter we show that a bottom-up approach for such a construction is more useful, as we do not need an upper-bound on the size of the solution.

2.1 Bottom-up costruction of an SLP for a word

Definition 2.1. Two different letters ab are called a *pair*. An occurrence of a^ℓ in w is a *maximal block* if it cannot be extended to the right nor left.

A pair compression of ab in w replaces each occurrence of ab by occurrence of a fresh letter c. A block compression of a in w for each ℓ replaces all maximal blocks a^ℓ with a fresh letter a_ℓ. To simplify the description, for p: a pir of different letters or a letter, we will say that we compress p.

"a_ℓ" is just a naming convention, it does not store any information about ℓ.

The following algorithm builds an SLP for a word. Our goal is to simulate it on the solution word.

Algorithm 1 Compression of a word w

```
1: while |w| > 1 do
2:     L ← list of letters in w
3:     choose p a pair of different letters from L or a letter from L
4:     compress blocks of p.
```

2.2 Soundness and completeness

Definition 2.2 (Soundness, completeness)**.** A nondeterministic procedure *is sound*, when given an 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 2.3. *The following operations are sound:*

1. *replacing all occurrences of a variable X with wXw' for arbitrary $w, w' \in (\Sigma \cup \mathcal{X})^*$;*
2. *replacing some occurrences of a word $w \in \Sigma^+$ (in u and v) with a fresh letter c;*
3. *removal of a variable X from the equation.*

Proof. In the first case, if s' is a solution of $u' = v'$ then s defined as $s(X) = s'(wXw')$ 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 $s(X) = \epsilon$ and $s(Y) = s'(Y)$ in other cases. □

2.3 Crossing and Noncrossing Pairs and Blocks

Definition 2.4. Given an equation $u = v$ and a substitution s and an occurrence of a substring $w \in \Sigma^+$ in $s(u)$ (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 substitution of $s(X)$ for a single 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*. 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.

Unless explicitly stated, we consider crossing/non-crossing pairs ab in which $a \neq b$.

Lemma 2.5. *Given an equation with n occurrences of variables the number of different crossing pairs and blocks is at most $2n$.*

Proof is left as an easy exercise.

2.4 Compression of noncrossing pairs and blocks

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

1: let $c \in \Sigma$ be an unused letter
2: replace each explicit ab in u and v by c

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

1: **for** each explicit a occurring in u or v **do**
2: **for** each ℓ **do**
3: let $a_\ell \in \Sigma$ be an unused letter
4: replace every explicit a's maximal ℓ-block occurring in u or v by a_ℓ

Lemma 2.6. 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: the new equation $u' = v'$ has a solution s' such that $s'(u')$ is obtained by compression of pair ab in $s(u)$.*

Similarly, BlockCompNCr(a) *is sound and when a has no crossing blocks in $u = v$ (with respect to some solution s) it is complete: the new equation $u' = v'$ has a solution s' such that $s'(u')$ is obtained by compression of each maximal block a^ℓ in $s(u)$ into a_ℓ.*

In particular, in both cases if anything was compressed, so $(u, v) \neq (u', v')$ then $|s'(u')| < |s(u)|$.

Proof. From Lemma 2.3 it follows that both $\mathsf{PairCompNCr}(a, b)$ and $\mathsf{BlockCompNCr}(a)$ 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:

explicit the explicit occurrences of ab are replaced by $\mathsf{PairCompNCr}(a, b)$,

implicit the implicit ones are replaced by the definition of s' and by the assumption

crossing there are no crossing occurrences.

In particular, if anything was compressed in the equation then $|s'(u')| < |s(u)|$.

The same argument 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. □

2.5 Uncrossing

In the following, we always assume that a solution for each variable is non-empty (and remove such variables otherwise).

Algorithm 4 $\mathsf{Pop}(a, b)$

```
1: for X ∈ 𝒳 do
2:     if the first letter of s(X) is b then                          ▷ Guess
3:         replace each X in u and v by bX          ▷ Implicitly change s(X) = bw to s(X) = w
4:         if s(X) = ϵ then                                           ▷ Guess
5:             remove X from u and v
6:     ...                          ▷ Perform a symmetric action for the last letter and a
```

Lemma 2.7. *The* $\mathsf{Pop}(a, b)$ *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'*).*

Algorithm 5 Pop (a) Popping a-prefixes and a-suffixes

```
1: for X ∈ 𝒳 do
2:     let ℓ_X and r_X be the lengths of the a-prefix and suffix of s(X)          ▷ Guess
                                                  ▷ If s(X) = a^{ℓ_X}_X then r_X = 0
3:     replace each X in u and v by a^{ℓ_X} X a^{r_X}      ▷ ℓ_X and r_X are stored as bitvectors,
                                          ▷ implicitly change s(X) = a^{ℓ_X} w a^{r_X} to s(X) = w
4:     if s(X) = ϵ then                                                    ▷ Guess
5:         remove X from u and v
```

Lemma 2.8. $\mathsf{Pop}(a)$ *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 a-suffix of* $s(X)$*. Then when* Pop *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'*.*

2.6 The algorithm

Algorithm 6 WordEqSat Checking the satisfiability of a word equation

```
1: while |U| > 1 or |V| > 1 do
2:     L ← list of letters (in the equation)
3:     choose p a letter or pair from L                    ▷ Guess
4:     if p is crossing then
5:         uncross p
6:     compress p
7: Solve the problem naively          ▷ With sides of length 1, the problem is trivial
```

Lemma 2.9. *Given an equation $u = v$ and its length-minimal solution s the length of the maximal a-block in $s(u)$ is $2^{\mathcal{O}(|uv|)}$.*

The proof is given in later Chapter 7 and it follows from a more general bound on the exponent of periodicity.

Lemma 2.10. *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.*

Note that the bound does not hold for all nondeterministic choices, but by using standard techniques we can bound the space available to the algorithm and reject the computation that try to exceed this space.

Proof. As we do not add occurrences of variables, the equation has at most n occurrences of variables.

If there is an uncrossing pair or letter, then we compress it. This does not increase the size of the equation.

So consider the case in which there are no non-crossing pairs and no non-crossing letters. For an occurrence of a letter a in $u = v$ and p; a pair or a letter we say that this occurrence is covered by p, when some occurrence of p includes this occurrence of letter. Note that each letter in the equation is covered by some occurrence of some p.

Choose p which covers most letters in the equation, let the equation have m letters. As there are non non-crossing letters and pairs, there are at most $2n$ different ps, hence some covers at least

$$\frac{m}{2n}$$

different occurrences; fix such p. Uncrossing of p introduces at most $2n$ letters to the equation. Compression of p removes at least $\frac{m}{4n}$ letters from the equation. Hence uncrossing and compressing p leads to an equation with

$$m - \frac{m}{4n} + 2n = (1 - \frac{1}{4n})m + 2n$$

Now, if $m \leq 8n^2$ then also above bound yields a bound of at most $8n^2$ on the size of the equation, hence for appropriate non-deterministic choices the equation has length at most $8n^2$. □

Theorem 2.11. *Satisfiability of word equations is in* PSPACE.

Proof. First of all, the whole algorithm runs in polynomial space.

As all subprocedures are sound, we never return YES for an unsatisfiable equation.

If the equation is satisfiable, then after each compression step, which changes something, we end up with an equation with a shorter solution word (for a length-minimal solution). Thus we cannot cycle. So we can have counter, which after visiting large enough number of equations tells us to reject. □

Exercises

Task 10 Show that for a word equation with m occurrences of variables the sum of numbers of different crossing pairs and different letters with crossing blocks is at most $2m$.

Task 11 Let s be a length-minimal solution of a word equation $u = v$. Show that

- Let ab occur in $s(u)$. Show that ab has a crossing or explicit occurrence in $s(u)$ or $s(v)$ (with respect to s).
- Let a occur in $s(u)$. Show that a occurs in u or v, i.e. it has an explicit occurrence.
- Let a^ℓ be a maximal block in $s(u)$. Show that it has a crossing, explicit occurrence or it is a prefix or suffix of some $s(X)$ (so in other words: it touches the cut). It might help to look at $ba^\ell c$.

Task 12 Show that we can uncross and compress all blocks of all letters in parallel, i.e. as one procedure that pops at most one prefix and one suffix per occurrence of variable.

Task 13 A *partition* of an alphabet Σ is a pair (Σ_1, Σ_2) such that $\Sigma_1 \cup \Sigma_2 = \Sigma$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$.

Show that we can uncross and compress a set of pairs $\{a_i b_i\}_{i \in I}$ in parallel, assuming that $a_i \in \Sigma_1$ and $b_i \in \Sigma_2$ for each $i \in I$.

Task 14 Consider a word $w \in \Sigma^*$ such that none of its two consecutive letters are the same. Occurrence of letters from an occurrence of a pair ab in w is *covered* by a partition (Σ_1, Σ_2) if $a \in \Sigma_1$ and $b \in \Sigma_2$. Show that there is a partition of Σ such that it covers at least $\frac{|w|-1}{2}$ letters in w. Show that it can be computed in linear time.

Generalise this observation to a word equation with a solution s (and at most n occurrences of variables).

Hint: Reduce this problem to calculation of a maximal (weighted) cut in a graph. It has a simple randomised solution which can be derandomised using expected value approach. It is described in Michael Mitzenmacher, Eli Upfal *Probability and computing* book [44] as well as in Vijay Vazirani *Approximation algorithms* book [65].

Task 15 Using Tasks 12–14 devise an algorithm for word equation that keeps a linear-size equation; the algorithm can use more memory when processing the equations, moreover, at some point it will have to store blocks a^{cn}, but we treat them as size-1. (The latter is a cheat, but we will learn how to deal with this later on).

Task 16 Using Task 15 devise an algorithm that verifies the equivalence of SLPs in polynomial time.

You should probably have it run in two modes: one aims at reducing $|\mathcal{A}|$ by a constant fraction and the other at reducing $|\operatorname{val}(\mathcal{A})|$ by a constant fraction; here $\operatorname{val}(\mathcal{A})$ denotes the word derived by $\mathcal{A}$.

There are are some uniteresting details concerning the exact computational model, so you can ignore the $\log n$ factors.

Note: it is an open problem, whether this can be solved significantly faster than $\mathcal{O}(|\mathcal{A}| \log |\operatorname{val}(\mathcal{A})|)$.

Task 17 [(Long and tedious, but not that difficult), 2 points] The goal of this task is to create a variant of algorithm that performs only compression of pairs, perhaps pairs of the same letter.

Note: we do not use the bound on the exponent of periodicity.

The reason why we cannot use compression of pairs aa is that they can overlap and the compression is ambiguous, for instance consider an equation $aX = Xa$ (all its solutions have $s(X) \in a^*$). We cannot pair letters in X and in $s(aX)$ in the same way.

However, this can be walked around: observe, that a and X commute, as they both represent blocks of a. Thus we can change aX to Xa on the left-hand side, without affecting the equation.

Show, that if there is a particular letter a, such that each variable either:

1. has no a-prefix and no a-suffix *or*

2. is a block of a

then we can rearrange the variables and perform the aa-pair compression. This should pop at most 1 letter from each variable.

Show that afterwards 1–2 is satisfied for a', which represents aa.

To reach an equations satisfying 1–2 we pop a-prefixes and a-suffixes of variables, but represent them as variables.

However, this is not yet enough, as we pile up with many letters popped from variables. To remedy this, we *type* the letters that represent compressed blocks of a: initially we type a and variables satisfying 2; then we additionally perform pair compression for letters that are a-typed. Show that in this way 1 can be generalised: there is no prefix and suffix of a-typed letters.

This should be enough for the algorithm.

Task 18 Assume that Task 17 is correct, i.e. we are able to solve word equations in (non-deterministic) polynomial space performing only compressions of the form $ab \to c$. Show that this implies that the size of the length-minimal solution is at most doubly exponential, i.e. at most $2^{2^{p(n)}}$, where p is a polynomial.

This argument does not work that easily for variant with block compression. Can you say why?

Task 19 Show that the algorithm for word equations (in some variant: choose whichever you like) in fact generates an SLP of size $\mathrm{poly}(n, \log N)$ for some solution of a word equation of size N. How low can you make the dependency on $\log N$?

Task 11 should be helpful.

Chapter 3

Free groups

3.1 Free groups

Given a finite alphabet Σ define Σ^{-1} as $\{a^{-1} : a \in \Sigma\}$. Define a reduction (rewriting) rules

$$aa^{-1} \to \epsilon, \quad a^{-1}a \to \epsilon \ . \tag{3.1}$$

Lemma 3.1. *Every word in* $(\Sigma \cup \Sigma^{-1})^*$ *has a unique normal form under the rewriting rules* (3.1).

A simple proof is left as an exercise.

Words as in Lemma 3.1 are called *reduced* or *irreducible*, for a word w this normal form is denoted by $\mathrm{IRR}(w)$.

Definition 3.2. 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.

As the normal form is unique, it holds that

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

and so we may also treat elements in $(\Sigma \cup \Sigma^{-1})^*$ as elements of the free group and the multiplication is defined inthe same way.

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

We consider word equations in free groups, defined in a natural way. From algebraic perspective they are more interesting than the semigroups. Makanin extended his results for word equations to groups [41, 42]. We can naturally see a word equation over a free group as an ordinary word equation over $(\Sigma \cup \Sigma^{-1})^*$, 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 $Y = b^{-1}$.

3.2 Free monoids/semigroups with involution

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

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.

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

3.3 Reduction: equations in groups to equations in free semigroup with involution and rational constraint

Theorem 3.3. *Given a system of equations over a free group it can transformed into a system of equations over free monoid with involution such that there is a bijection between solutions of the system of equations in the free group and solutions over the free monoid with involution that do not contain factors $a\overline{a}$. This bijection is an identity of variables that occur in both systems.*

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

Exercises

Task 20 (Newman's lemma) A rewriting system $S = \{(\ell_i, r_i)\}_{i \in I}$ is confluent, if for all s, t, u with $s \to_S^* t$ and $s \to_S^* u$ there exists v with $t, u \to_S^* v$; it is local confluent, if s, t, u with $s \to_S t$ and $s \to_S u$ there exists v with $t, u \to_S^* v$.

S is terminating, if there is no infinite chain

$$s_1 \to s_2 \to \cdots \to s_n \to \cdots .$$

Show that if S is locally confluent and terminating then it is confluent.

Chapter 4

Solving equations in free groups

By Theorem 3.3 to solve equations in free groups it is enough to solve them in free semigroups with involution and constraints $w \in \mathrm{IRR}(M)$ (and the results from Chapter 5 also ask for constraints of the form "w does not use letter a"). In general we will do this with the regular constraints.

4.1 Regular sets

Consider Σ^*, think of it as a free semigroup. A regular language is defined using an NFA 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 M_a 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 $n \times n$.

Remark. In this setting the automaton reads the word left to right; in many cases one sets $m_{p,q} = 1$ iff we can go from q to p; this has the disadvantage that we use the automaton that reads from right to left, but we can use usual column vectors to denote the configuration of an automaton.

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 (finite) rest is rejecting.

If we consider a monoid with involution, then we usually assume that the regular constraints are given by a homomorphism that also respects this involution. There is no canonical way to define the involution, it could be, say, the inverse on the Boolean matrices (if this is well defined), but could be any other operation, for instance — the transpose.

It is easy to see that if $\varphi : \Sigma^* \to \mathbb{M}$ does not respect the involution then we can take larger matrices and define the new homomorphism so that it does respect the involution (this may be a different involution than originally, though).

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

4.2 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 the (finite) set 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 (assuming that we deal with classes NP or higher), as we can initially non-deterministically guess the appropriate transition from a set of transitions.

Lemma 4.1. *Word equations with regular constraints of the form $X \in R$, $Y \notin R'$ where the regular languages are defined using NFAs that are part of the input are* NP*-equivalent to the same equation with regular constraints given by $\rho(X)$, where ρ maps letters and variables to vectors of Boolean matrices*

4.3 Model

We work with equations over $(\Sigma \cup \overline{\Sigma})^*$. Every variable X has the associated variable $\overline{X}$. We require that a solution satisfies

$$s(\overline{X}) = \overline{s(X)} \ .$$

Concerning the regular constraints, we assume that we are given $\rho_1, \ldots, \rho_m$ that are homomorphisms from $(\Sigma \cup \overline{\Sigma})^*$ to Boolean matrices (with some involution) and that they do respect the involution, i.e.

$$\overline{\rho(w)} = \rho(\overline{w}) \ .$$

They are collectively called ρ, in the sense that $\rho(w) = (\rho_1(w), \ldots, \rho_m(w))$. The input specifies $\rho(X)$ for each X and we require that a solution s satisfies the equation and for each variable

$$\rho(s(X)) = \rho(X) \ .$$

As an additional technical assumption we assume that the involution has no fixed-points, i.e. $\overline{a} \neq a$ for each $a \in \Sigma \cup \overline{\Sigma}$. This is not necessarily true for the input alphabet, but it is easy to modify the instance so that this holds (exercise).

4.4 Main issue

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. For the moment we assume that an equation comes with a (perhaps very large) alphabet and the solution should be over this alphabet.

4.5 The algorithm

In essence we are going to run the previous algorithm, a couple of modifications are needed, though. In particular, it is based on popping and compression.

4.6 Needed modifications

4.6.1 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 $\rho(X')$ is guessed and verified. We also need to guess when we remove the variable, in which case we need to have $\rho(X) = \rho(\epsilon)$.

4.6.2 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.

4.6.3 Pair compression

Since we do not want the letters $b = \overline{b}$, we never compress pairs $a\overline{a}$.

4.6.4 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 "forget" 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 (exercise).

4.6.5 Preprocessing

For technical reasons, we need to ensure that there is at least one letter in the equation (as otherwise we may end up with something like $X = X$ plus constraints). This is clearly preserved by all operations, so at the very beginning, in a preprocessing phase, we pop one letter from one variable.

4.7 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 (note 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 $\alpha : \Sigma' \to A^*$, which is constant on A and compatible with ρ, in the sense that $\rho(b) = \rho(\alpha(b))$ for all letters b. Thus, we extend the notion of a solution: a pair (s, α) is called a full solution of the equation. 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 $\alpha(\overline{a}) = \overline{\alpha(a)}$. Note that α is used only in the analysis, it is not stored or constructed by the algorithm, nor does it influence the working of the algorithm.

Definition 4.2. During the work of the algorithm that was given an equation over $(\Sigma \cup \overline{\Sigma})^*$ we denote $\Sigma_0 = \Sigma \cup \overline{\Sigma}$ and call it the *input alphabet*. Given the equation $u = v$ and a solution s the *solution's* alphabet denotes the smallest alphabet that includes all letters of $s(U)$ and the input alphabet and

the *equation's alphabet* is the smallest alphabet that includes the input alphabet and all letters in $u = v$ (except variables).

For an equation $u = v$ by a *full solution* we denote a pair (s, α) such that s is a solution of $u = v$ and α is a function from the solutions alphabet to words over the input alphabet that respects the involution and it is compatible with the constraints ρ, i.e.

- $\alpha : \Sigma \to \Sigma_0^*$, where Σ is solution's alphabet for s and Σ_0 is the input alphabet;
- $\overline{\alpha(a)} = \alpha(\overline{a})$ for each $a \in \Sigma$;
- $\rho(a) = \rho(\alpha(a))$ for each $a \in \Sigma$;
- $\alpha(a) = a$ for each letter in the input alphabet.

Example 4.1. If there are no constraints then for a given equation α can be defined in any way that respects the involution.

On the other hand, if the equation contains a letter c that has a transition $\rho(c)$ that is not realised by any word $w \in \Sigma^*$, where Σ is the input alphabet, then there is no α; this is somehow to be expected, as then c does not represent any word over the input alphabet (and in fact the algorithm cannot construct it).

It is easy to define α after a compression operation: when w is replaced with c 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 $\alpha(c) \neq \alpha(c')$).

Lemma 4.3. *For any subprocedure, if the equation $u = v$ before the procedure has a full solution (s, α) then for appropriate non-deterministic choices the new equation ($u' = v'$) has a full solution (s', α') such that $\alpha(s(u)) = \alpha'(s'(u'))$.*

Proof. If there is no compression, then $\alpha' = \alpha$. If w is compressed to c then $\alpha'(c) = \alpha(w)$. The existence of the solution follows in the same way as before. □

Definition 4.4. A solution s of an equation $u = v$ it is *simple* if the solution's alphabet is the equation's alphabet.

In other words, it uses only letters that are in the equation or were in the input equation.

Given a non-simple full 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 $\rho(s(X)) = \rho(s'(X))$). 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', α) is a full 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.

Lemma 4.5. *Suppose that (s, α) is a full solution of the equation (u, v). Then its simplification (s', α) is also a full solution of (u, v) and $\alpha(s'(u)) = \alpha(s(u))$.*

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 $\rho(b) = \rho(\alpha(b))$ we have that $\rho(s(X)) = \rho(s'(X))$. Thus, $\alpha(s'(u)) = \alpha(s(u))$. □

In other words, we can always assume that if the equation has a solution then it has a simple one.

Algorithm 7 WordEqInvRegSat Checking the satisfiability of a word equation with involution and regular constraints

```
1: Σ ← input equation
2: Pop (Σ, Σ)                                   ▷ Pop some letter from some variable
3: while u or v is not a letter do
4:     Σ' ← letters in the equation or Σ
5:     close Σ' under involution (Σ ← Σ' ∪ the involutions of Σ')
6:     choose p: a letter a ∈ Σ' or aā with a ∈ Σ' or ab ∈ Σ'^2 (here b ≠ a ≠ ā)
                                    ▷ Choose such that p has an implicit or crossing occurrence
7:     if p is crossing then
8:         uncross p
9:     Compress p
```

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

$$\mathrm{w}(s,\alpha) = |U| + |V| + 2\sum_{X\in\mathcal{X}} |UV|_X |\alpha(s(X))| \ , \tag{4.1}$$

Lemma 4.6. *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.*

Weight can be used to show the termination of the algorithm.

Lemma 4.7. *For any subprocedure, if it transforms a satisfiable equation (u, v) to a satisfiable equation $(u', v') \neq (u, v)$ then the corresponding full solution of (u', v') has a smaller weight than the full solution of (u, v).*

Proof. Note that in (4.1) 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. □

Lemma 4.8. *There is a constant c such that during the run of* WordEqInvRegSat *given an equation of size at most cn^2 with full solution (s, α) there is a p such that after the uncrossing (when needed) and compression of p the new equation has a full solution (s', α') with less weight than before and size at most cn^2.*

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.

Exercises

Task 21 An *involution* $\bar{\cdot}$ is any operation (defined in a semigroup) such that $\bar{\bar{\cdot}}$ is an identity and $\overline{ab} = \bar{b}\bar{a}$. In particular, we can define on some letters as an identity, such letters are called self-involuting.

Show that we can reduce a problem of word equations in a free semigroup with involution and regular constraints to the case in which there is no self-involuting letter.

Here regular constraints are defined using a homomorphism from the free semigroup with involution to a finite semigroup (with involution).

Task 22 Show that if a homomorphism $\rho : M \to \mathbb{B}_{n\times n}$ (so: Boolean matrices of size $n \times n$) from a free monoid with involution M into Boolean matrices does not preserve involution (in particular, the involution on $\mathbb{B}_{n\times n}$ may be undefined), then we can find a different set of Boolean matrices $\mathbb{B}_{m\times m}$ for

which the involution is defined and there is a homomorphism $\rho' : M \to \mathbb{B}_{m\times m}$ from M to $\mathbb{B}_{m\times m}$ that preserves the involution and for each set of the form $\rho^{-1}(M)$ for some $M \subseteq \mathbb{B}_{n\times n}$ there is $M' \subseteq \mathbb{B}_{m\times m}$ such that $\rho^{-1}(M) = \rho'^{-1}(M')$ (but not necessarily the other way around), i.e. regular sets defined using ρ can be also defined using ρ'.

Hint: Take $\overline{\mathbb{B}_{n\times n}}$ and consider $\mathbb{B}_{n\times n} \times \overline{\mathbb{B}_{n\times n}}$. How to define the involution?

Task 23 (2 points) Show that given a word equation over a free monoid with regular constraints given by ρ we can extend the input alphabet Σ by letters

$$\{a_\tau \,:\, \tau \in N \text{ and there is a word } w \in \Sigma^* \text{ such that } \rho(w) = \tau\}.$$

Show the equisatisfiability of the problem over the original alphabet and over such an extended alphabet. Modify the algorithm that tests the satisfiability of word equations so that it works also in case of regular constraints. Can you implement the algorithm in PSPACE?

Chapter 5

Positive theory of free groups

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

$$Q_1 X_1 X_2 X_2 \dots Q_k X_k \varphi(X_1, X_2, \dots, X_k)$$

where each Q_i is a quantifier and φ is a formula that uses only variables $X_1, X_2, \dots, X_k$, constants from appropriate domain and relations (and functions, when needed) and only $\wedge$ and $\vee$ as used as logical connectives. A *positive theory* of a structure $\mathbb{A}$ consists of positive sentences that hold in $\mathbb{A}$. The corresponding decision problem asks to decide, whether a given sentence belongs to a positive theory (of $\mathbb{A}$).

It is easy 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 [42]. 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.

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. 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 $\{k_1, k_2, \dots, k_i\}$, but not the later constants. 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 $Y = k^{-1}$.

To make the visible distinction more clear, we will use small letters for constants and capital letters for variables (usually quantified).

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 $i : A \to B$ is a homomorphism, then $\overrightarrow{i(z)}$ holds in B.

Lemma 5.1. *Let $\varphi(\vec{X})$ be a positive formula with free variables $\vec{X}$ and let $i : A \to B$ be a homomorphism onto B. Then for any vector $\vec{z}$ of elements of A if $\varphi(\vec{z})$ holds in A then $\varphi(i(\vec{z}))$ holds in B.*

Proof. We make the induction over the structure of φ. First, if φ is a relation, this holds by the definition of the homomorphism.

Then by easy induction this holds also when φ is quantifier-free (this holds for all atoms and we take a positive Boolean combinations of the atoms).

Let $\varphi(\vec{z}) = \forall X \psi(X, \vec{z})$. Then by the induction assumption it holds for $\psi(x, \vec{z})$ for each x and $\vec{z}$. Fix $\vec{z}$. When we apply the quantifier, the formula $\varphi(\vec{z})$ holds when for all $x \in A$ it holds that $\psi(x, \vec{z})$ holds. But then by the induction assumption, also $\psi(i(x), i(\vec{z}))$ holds for each $i(x)$ and this takes as values all elements of B. So also $\varphi(i(\vec{z}))$ holds in B.

The argument for the existential quantifier is similar (for a witness $x \in A$ we take the witness $i(x)$ in B). □

5.1 Notation

The input free group, with generators Σ, is denoted as $\mathbb{G}$. We shall extend our free semigroup by new elements: let $\langle E \rangle$ denote the group generated by E, the relations between elements in E are always clear from the context, usually those are free generators. Let also $G * H$ denote the free product of G and H, i.e. this group is generated by $\langle G, H \rangle$ and there are no nontrivial relations between elements of G, H. In the process, we will use many new constants $k_1, \ldots, k_m$. Then by $\mathbb{G}_{[i..j]}$ we denote $\mathbb{G} * \langle k_i \rangle * \langle k_{i+1} \rangle * \cdots * \langle k_j \rangle$.

In the proof we will also need to use the corresponding free monoid with involution. By $\mathbb{M}$ we denote the free monoid with involution with generators Σ and $\mathbb{M} * \langle k \rangle$ is defined analogously, also $\mathbb{M}_{i..j}$ is defined in a similar way. We always assume that $\overline{k} \neq k$ in those monoids.

5.2 Main result

The main result of this section is the following theorem.

Theorem 5.2. *Let $\mathbb{G}$ be a free group. Then for all $\vec{z} \in \mathbb{G}$ a positive formula with no free variables*

$$\psi(\vec{Z}) = \forall X_1 \exists Y_1 \cdots \forall X_m \exists Y_m \varphi(X_1, \ldots, X_m, Y_1, \ldots, Y_m, \vec{z}). \tag{5.1}$$

holds in $\mathbb{G}$ if and only if

$$\exists Y_1 \in \mathbb{G}_{[1..1]} \exists Y_2 \in \mathbb{G}_{[1..2]} \cdots \exists Y_m \in \mathbb{G}_{[1..m]} \varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}). \tag{5.2}$$

holds in $\mathbb{G} * \langle k_1, k_2, \ldots, k_m \rangle$.

Note that Theorem 5.2 gives the decidability of positive theory of free groups: the only extra condition is restriction of the generators for free groups, which can be modelled by regular constraints (in the group or in the semigroup case).

Remark. We give the proof in the case of equations alone, but it can be generalised ot the case of regular constraints.

Remark. Note, that Makanin's original construction gave a "concrete" word, instead of a "random" one.

5.3 Main technical Lemma

Lemma 5.3. *Let $\mathbb{M}$ be a free monoid with involution and let $\mathbb{M}_2, \ldots, \mathbb{M}_m$ be free monoids with involution that contain it and let k be constant not present in any of them.*

Let φ be a positive formula without free variables $\vec{Z}$ of the form:

$$\psi(\vec{Z}) = \forall X_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_1 \in \mathrm{IRR}(\mathbb{M}_1) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_2) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_m) \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_m) \\ \varphi(X_1, Y_1, \ldots, Y_m, \vec{Z}, \vec{Y}) \ .$$

If it holds on some sequence of elements $\vec{z} \in \mathbb{M}$ then there exist two words $s_1, s_2 \in \mathrm{IRR}(\mathbb{M})$ such that the following formula holds:

$$\exists Y_1 \in \mathrm{IRR}(\mathbb{M} * \langle k \rangle) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_2 * \langle k \rangle) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_m * \langle k \rangle) \\ \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_m * \langle k \rangle) \varphi(s_1 k s_2, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}),$$

Note that the s_1, s_2 are constants but they *can* (and actually do) depend on $\vec{z}$.

The rest of this Section is devoted to the proof of the Lemma.

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 $R = \{r_0, r_1, \ldots, r_\lambda\} \subseteq \{a, b, \overline{a}, \overline{b}\}^p$, where p is some large constant (to be established later), in particular, twice longer than any constant in the system; those constants include those in $\vec{z}$.

Consider a string

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

Roughly, this is a string that we use for $\forall X$ quantifier, but we shall replace some $r_i \ell r_{i+1}$ by $r_i k r_{i+1}$, where k is a fresh constant.

Given $|r_i| = p$ and $|\ell|$ we say that set of strings R has *enough randomness*, when each word w of length at least $(|r_i| - |\ell|)/2$ occurs in at most one of strings in $R \cup \overline{R}$ and it has at most one occurrence is such a string.

Lemma 5.4. *There is a set R with properties above that has enough randomness.*

Using Kolmogorov complexity/Probabilistic method it is easy to show that such set of strings exists, for large enough m. Alternatively, one can give explicit construction. This is left as an exercise.

The meaning of enough randomness notion is that

Lemma 5.5. *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 $r = r_{i+1}$).*

Proof. 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 $(|r| - |\ell|)/2$. So this substring has two occurrences in $R \cup \overline{R}$, which is a contradiction. □

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

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

Lemma 5.6. *Each system P_i is confluent and length reducing;in particular it has a unique normal form.*

Each of those rewriting systems is confluent and so has a unique normal form, denoted by $\kappa_i(w)$.

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 5.7. *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 5.8. *Let $\{x_j y_j = z_j\}_{j=1}^d$ be a set of equations. Then there is i such that for each j*

$$\kappa_i(x_j)\kappa_i(y_j) = \kappa_i(z_j) \tag{5.3}$$

On the other hand, if (5.3) holds then $x_j y_j = z_j$ holds as well.

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. □

proof of Lemma 5.3. Take s as the string substituted for X and all the witnesses $y_1, \dots, y_m, \vec{y}$. We then take the rewriting system guaranteed to exist by Lemma 5.8 and rewrite all the constants and witnesses. Then

- x is replaced so that it contains a single occurrence of k;
- each witness is rewritten, maybe it has occurrences of k;
- constants (inluding those in $\vec{z}$) are too short to be rewritten;
- all equations that used to hold still hold by Lemma 5.8).

- if after the rewriting the equation holds then it held also originally (we can replace k back with ℓ to get the original equation). Denote the obtained formula by ψ'.

  The rest of the argument is similar as in the case of Lemma 5.1: when we consider atoms, if an atom of $\varphi(s, y_1, \ldots, y_m, \vec{y}, \vec{z})$ holds then the corresponding atom of $\varphi(\kappa x, \kappa(y_1), \ldots, \kappa(y_m), \vec{\kappa}(y)\vec{\kappa}(z))$ holds.

  We then proceed with an induction: if a subterm of $\varphi(s, y_1, \ldots, y_m, \vec{y}, \vec{z})$ holds then the corresponding subterm of $\varphi(\kappa x, \kappa(y_1), \ldots, \kappa(y_m), \vec{\kappa}(y)\vec{\kappa}(z))$ holds.

  Lastly, we go through the existential quantifiers (the witnesses are explicitly given).

  In the other direction, if $\kappa_i(x_j)\kappa_i(y_j) = \kappa_i(z_j)$ holds, then $x_j y_j = z_j$ os obtained by replacing (all occurrences of) a constant by a string, which preserves the equality. □

5.4 Main proof: quantifier elimination

Denote by $\mathbb{G}_{[i..j]}$ the free group $G * \langle k_i, \ldots, k_j \rangle$ and introduce similar notation for the free monoid with inversion. The proof of Theorem 5.2 is done by induction on the number of the quantifiers. If there are none then we are done.

Otherwise the formula is

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

for some $m > 0$. By assumption for each $x_1, y_1, \vec{z} \in \mathbb{G}$ the formula

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

(note that x_1 and y_1 are now fixed elements) holds in $\mathbb{G}$ if and only if

$$\exists Y_2 \in \mathbb{G}_{[2..2]} \cdots \exists Y_m \mathbb{G}_{[2..m]} \varphi(x_1, k_2, \ldots, k_m, y_1, Y_2, \ldots, Y_m, \vec{z}) \ .$$

holds in $\mathbb{G}_{[2..m]}$.

As x_1, y_1 are any elements, we can take the existential quantifier over y_1 and then the universal over x_1, thus the following are equivalent:

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

and

$$\forall X_1 \in \mathbb{G} \exists Y_1 \in \mathbb{G} \exists Y_2 \in \mathbb{G}_{[2..2]} \ldots \exists Y_m \in \mathbb{G}_{[2..m]} \varphi(X_1, k_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}), \tag{5.4}$$

In the following we prove equivalence of (5.4) and (5.2).

Lemma 5.9. *For φ positive if $\vec{z}$ satisfies* (5.2) *then it satisfies* (5.4).

Proof. We use Lemma 5.1.

Take any $x_1 \in \mathbb{G}$. Take a homomorphism $h : \mathbb{G}_{[1..m]} \to \mathbb{G}_{[2..m]}$ defined by $h(k_1) = x_1$ and as an identity on other generators, note that it is naturally restricted to a homomorphism from $\mathbb{G}_{[1..i]}$. Take $y_1, \ldots, y_m \in$ such that $y_i \in \mathbb{G}_{[1..i]}$ such that $\varphi(k_1, \ldots, k_m, y_1, \ldots, y_m, \vec{z})$ holds. Then Lemma 5.1 yields that

$$\varphi(h(x_1), h(k_2), \ldots, h(k_m), h(y_1), \ldots, h(y_m), h(\vec{z})) = \varphi(k_1, k_2, \ldots, k_m, h(y_1), \ldots, h(y_m), \vec{z})$$

holds as well. Take $h(y_i)$ as a witness for Y_i, which shows that (5.4) holds, as claimed. □

Lemma 5.10. *For φ positive if $\vec{z}$ satisfies* (5.4) *then it satisfies* (5.2).

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

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

is rewritten into formula

$$\exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}) \varphi'(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y})$$

(note that $\vec{Y}$ may depend on $\vec{z}$) where the new variables $\vec{Y}$ are used to appropriately brake down the equations. Adding the quantifiers yields that (5.4) is equivalent to:

$$\forall X_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_{[2..2]}) \ldots \exists Y_{[2..m]} \in \mathrm{IRR}(\mathbb{M}_{[2..m]}) \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_{[2..m]}) \\ \varphi'(X_1, k_2 \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}) \ .$$

By the Lemma 5.3 if it holds then for some $s_1, s_2 \in \mathrm{IRR}(\mathbb{M})$ the formula

$$\exists Y_1 \in \mathrm{IRR}(\mathbb{M}_{[1..1]}) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_{[1..2]}) \ldots \exists Y_m \in \mathrm{IRR}(\mathbb{M}_{[1..m]}) \\ \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_{[1..m]}) \varphi'(s_1 k_1 s_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}) \ ,$$

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

$$\exists Y_1 \in \mathbb{G}_{[1..1]} \exists Y_2 \in \mathbb{G}_{[1..2]} \ldots \exists Y_m \in \mathbb{G}_{[1..m]} \varphi(s_1 k_1 s_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}) \ . \tag{5.5}$$

Consider an automorphism of $\mathbb{G}_{[1..m]}$ defined by $h(k_1) = s_1^{-1} k_1 s_2^{-1}$ and an identity on other generators (this is an automorphism, see Lemma 5.12). Since it is an isomorphism, we can apply it on (5.5), see Lemma 5.11 The only affected is the k_1 constant, so we get the following is equivalent:

$$\exists Y_1 \in \mathbb{G}_{[1..1]} \exists Y_2 \in \mathbb{G}_{[1..2]} \ldots \exists Y_m \in \mathbb{G}_{[1..m]} \varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}) \ ,$$

and this is exactly (5.2). □

Lemma 5.11. *Let $\mathbb{G}_1, \ldots, \mathbb{G}_m \leq \mathbb{G}$ be groups, $\vec{z} \in \mathbb{G}$ be elements of $\mathbb{G}$ and let $i : \mathbb{G} \to \mathbb{G}$ be an automorphism of $\mathbb{G}$ such that $i(\mathbb{G}_j) = \mathbb{G}_j$. Then*

$$\exists Y_1 \in \mathbb{G} \exists Y_2 \in \mathbb{G}_2 \ldots \exists Y_m \in \mathbb{G}_m \varphi(Y_1, \ldots, Y_m, \vec{z})$$

holds if and only if

$$\exists Y_1 \in \mathbb{G}_1 \exists Y_2 \in \mathbb{G}_2 \ldots \exists Y_m \in \mathbb{G}_m \varphi(Y_1, \ldots, Y_m, i(\vec{z}))$$

holds.

Proof. A simple proof is left as an exercise. □

Lemma 5.12. *Let $\mathbb{G} = \langle c_1, \ldots, c_m \rangle$ be a free group and consider $h : \mathbb{G} \to \mathbb{G}$ defined as*

$$\begin{aligned} h(c_1) &= g c_1 g' \\ h(c_i) &= c_i \qquad\qquad \text{for } i > 1 \ , \end{aligned}$$

where $g, g' \in \langle c_2, \ldots, c_m \rangle$. Then h is an automorphism of $\mathbb{G}$ (so an isomorphism from $\mathbb{G}$ to $\mathbb{G}$).

Proof. A simple proof is left as an exercise. □

The Lemmata 5.9 and 5.10 give the proof of Theorem 5.2.

Exercises

Task 24 The $\exists^*$-theory of word equations consists of all sentences of the form:

$$\exists_{x_1,x_2,\ldots,x_k} \varphi(x_1, x_2, \ldots, x_k)$$

where φ is quantifier-free logic formula that uses $\wedge, \vee, \neg$ as connectives and atomic formulas that are word equations that use constants from Σ^* and variables $x_1, x_2, \ldots, x_k$.

Show that we can verify sentences from this theory in PSPACE.

Hint: The algorithm will heavily employ non-determinism to reduce this case to a system of word equations. The inequalities are easy to handle: look for first differences.

Task 25 Show that a positive theory of word equations over free semigroup is undecidable. Two alternations of quantifiers are enough (one, if you put some thought into it: $\forall\exists$ is undecidable).

Hint: First make the claim about the whole theory and then eliminate the negation as in Task 24.

Task 26 Consider the positive $\exists\forall$ fragment of word equations over a semigroup (no negation). Show that it is decidable.

To this end consider first the $\forall$ positive fragment.

Hint: Universally quantified equations tend to be false.

Task 27 Show that for large enough r_i there is a set of enough random string.

Hint: The simplest proof is through Kolmogorov's complexity, but random strings should also be good.

Task 28 Show that each of the defined rewriting systems P_i is confluent and thus each term has a unique normal form (note that the rewriting system is length-reducing).

Task 29 Prove Lemma 5.11.

Task 30 Prove Lemma 5.12.

Chapter 6

Basic string combinatorics (stringology)

6.1 Periodicity

Definition 6.1 (Prefix, suffix)**.** A word u is a *prefix* of w when $w = uv$ for some v, this is denoted by $u \sqsubseteq w$, it is a *proper prefix* when additionally $u \neq w$, this is denoted by $u \sqsubset w$. Similarly v is a *suffix* (proper suffix) of w when $w = uv$ for some u (some $u \neq \epsilon$), this is denoted by $w \sqsupseteq v$ ($w \sqsupset v$, respectively).

Given a word w is *u-prefix* is the longest prefix of w from the set u^*.

Definition 6.2 (Period of a word)**.** A word $w = w[1\,..\,n]$ has a period u if

$$w = uw[1\,..\,n - |u|] \ .$$

A string p is a *border* of w when it is both a suffix and a prefix.

A string w is a *power of* (or *repetition of*) u if $w = u^k$ for some $k \geq 0$. It is a *power* (or *repetition*), if it is of the form $w = u^k$ for some $k > 1$.

Fact 6.3. *A word w has a period of length p if and only if it has a border of length $|w| - p$.*

Fact 6.4. *If a word w has a period p then it is of the form*

$$w = p^k p'$$

where p' is a prefix of p and $k \geq 1$.

Example 6.1. Consider a word *aabaaba*. It has periods *aab* and *aabaab*. It has a borders *aaba* and *a*. It is not a power. Its prefix $aabaab = (aab)^2$ is a power. It is of the form $(aab)^2 a$.

Depending of the context, the period and border are either words or the lengths of those words.

Lemma 6.5 (Periodicity Lemma)**.** *If a word w has periods p, q such that*

$$p + q \leq |w|$$

then w has a period $\gcd(p, q)$.

Corollary 6.6 (Alternative formulation)**.** *For two words u, v if*

$$uv = vu$$

then there is w and natural numbers $n, m \geq 0$ such that $u = w^n$, $v = w^m$, i.e. they are (perhaps trivial) powers of the same word.

Proof. The proof follows by an induction on the unordered pairs $\{\max(|u|, |v|), \min(|u|, |v|)\}$ sorted lexicographically. If $|u| = |v|$ then clearly $u = v$ and we are done; if one of u, v is empty then we are also done.

Otherwise, without loss of generality let $|v| < |u|$. Then from $uv = vu$ we conclude that v is a prefix of u and so $u = vu'$. Writing it down

$$vu'v = vvu' \text{ implies } u'v = vu' \ .$$

The rest follows from the induction assumption. □

The periodicity lemma has also a stronger variant

Lemma 6.7 (Strong Periodicity Lemma)**.** *If a word w has periods p, q such that*

$$p + q \leq |w| + \gcd(p, q)$$

then w has a period $\gcd(p, q)$

Corollary 6.8 (Alternative formulation)**.** *For two words u, v if uv and vu have a common prefix of length at least $|uv| - \gcd(|u|, |v|)$ then then there is w and n, m such that $u = w^n$, $v = w^m$, i.e. they are powers of a the same word.*

6.2 Failure function

Definition 6.9 (MP failure function)**.** Given a word $w = w[1\,..\,n]$ define

$$\pi_w[i] = \max\{j < i \,:\, w[1\,..\,j] \text{ is a border of } w[1\,..\,i]\}$$

In other words, for a prefix $w[1\,..\,i]$ we store the length of the longest non-trivial border (so other than whole $w[1\,..\,i]$).

Lemma 6.10. *Given a word w its failure function π_w can be computed in $\mathcal{O}(|w|)$ time.*

Example 6.2. Consider the word *aabaaba*. Then

$$\pi_{aabaaba} = [0, 1, 0, 1, 2, 3, 4] \ .$$

6.3 Primitive words

Definition 6.11. A word $u \neq \epsilon$ is *primitive* if $u = w^k$ implies $w = u$ and $k = 1$

Example 6.3. Word $p = aabaa$ is primitive, so is a word $p' = aabaaabaa$. Note that p is border of p'.

Definition 6.12. We say that word u, v are *conjugate* (or *cyclic shifts*) if there are words p, q such that

$$v = pq, u = qp.$$

Lemma 6.13. *Word u is primitive if an only if its conjugates are all pairwise different.*

Lemma 6.14. *Let u, u' be nonempty, conjugate words. Then u is primitive if and only if u' is primitive.*

Lemma 6.15. *Let u be primitive then*

$$u^2 = u'uu''$$

implies that $\{u', u''\} = \{\epsilon, u\}$.

Proof. From the statement it follows that $|u'u''| = |u|$. Furthermore, u' is a prefix of u and u'' is a suffix of u. Thus $u = u'u''$. Again from the equation we get

$$u'u''u'u'' = u'u'u''u''$$

and so $u''u' = u'u'' = u$, which implies that one of u', u'' is u and the other ϵ. □

Theorem 6.16. *Let u, v, w be primitive such that u^2 is a prefix of v^2 and v^2 of w^2. Then $|u|+|v| \leq |w|$.*

Theorem 6.17. *Given a word w there are $\mathcal{O}(\log |w|)$ different primitive p such that $p^2 \sqsubseteq w$. All such p can be found in $\mathcal{O}(|w|)$ time.*

Proof. The proof is left as an exercise. It follows from Theorem 6.16 and simple application of the MP array. □

Exercises

Task 31 Show that Lemma 6.7 follows from its variant in which $\gcd(|u|, |v|) = 1$.

Task 32 Prove Lemma 6.7, it may be easiest to prove Corollary 6.8 by adapting the proof of Corollary 6.6.

Task 33 (Alternative proof of Periodicity Lemma 6.5) Given a word $w[1\,..\,p+q]$ with periods p, q such that $\gcd(p, q) = 1$ define a graph on the positions of this word: there is an edge $\{i, j\}$ if and only if $|i - j| \in \{p, q\}$. Show that this graph is a cycle. Deduce from this that $w \in a^*$ for appropriate a.

Strengthen this to the case, when $w = w[1\,..\,p + q - 1]$.

Hint: What happens with the graph from the first point, when we remove the last node?

Task 34 Prove Theorem 6.16.

Task 35 Prove Theorem 6.17.

Task 36 Recall the linear-time construction of the MP array.

Chapter 7

Exponent of periodicity

By n we denote the length of the equation and by n_v the number of occurrences of variables in this equation.

Definition 7.1. 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 $u \in \Sigma^+$.

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 $u = v$.

The ultimate goal is to prove a well-known exponential bound on exponent of periodicity of length-minimal solutions.

Theorem 7.2 (Kościelski and Pacholski [27])**.** *Given an equation $u = v$ of length n and number of occurrences of variables n_v its exponent of periodicity* $\mathrm{per}(u = v)$ *is:*

$$\mathrm{per}(u = v) = \mathrm{poly}(n) \cdot 2^{\mathcal{O}(n_v)} \ .$$

.

This bound is known to be tight (and relatively easy to show), up to the constant in the exponent.

7.1 Idea and an example

As a gentle introduction, consider maximal a blocks in a solution $s(u)$ of an equation $u = v$. If s is length minimal then we know that each maximal block has a length which is a length of a crossing block or a-prefix or a-suffix of some variable. Hence, if $\ell_X, r_{X\,X\in\mathcal{X}}$ are the length of the a-prefixes and suffixes, then each of them is an arithmetic expression in $\ell_X, r_{X\,X\in\mathcal{X}}$. Now we would like to exchange $\ell_X, r_{X\,X\in\mathcal{X}}$ to other lengths $\ell'_X, r'_{X\,X\in\mathcal{X}}$, but this cannot be done arbitrarily.

Example 7.1. Consider an equation

$$XabXXa = aXbYYY \ . \tag{7.1}$$

It is easy to show that the solutions of of the form $s(X) = a^{\ell_X}, s(Y) = a^{\ell_Y}$ where

$$2\ell_X + 1 = 3\ell_Y \ .$$

and the other way around.

This is because some blocks "need to be" equal. We formalize this by introducing variables in place of values $\ell_X, r_X\, X\in\mathcal{X}$ and equation which enforce that the appropriate blocks (so arithmetic expressions) are equal. In our case this would be

$$2L_X + 1 = 3L_Y \ .$$

What are the exact equations? We equalize the lengths of blocks equal in s, consider the solutions of the corresponding system of linear equations, which translate to substitutions (and solutions) of the original word equation. In particular, $\ell_X, r_X\, X\in\mathcal{X}$ is a solution of linear system and it yields solution s of the word equation.

Now, if s is length-minimal then $\ell_X, r_X\, X\in\mathcal{X}$ is "minimal" in appropriate sense. And suchminimal solution can be bounded in terms of equation's parameters.

7.2 P-presentations

In order to generalize the idea from the previous section, we need to be careful: given a string w its occurrences may overlap. To solve this, we will look only at primitive words (which is enough to bound the exponent of periodicity).

Definition 7.3. Let P be a primitive word and $U_0, \ldots, U_u$ be a sequence of words. Define a function:

$$[U_0, \ldots, U_u] : \mathbb{N}^u \to \Sigma^* \text{ by } [U_0, \ldots, U_u](\ell_1, \ldots, \ell_u) = U_0 P^{\ell_1} U_1 P^{\ell_2} \cdots P^{\ell_u} U_u \ .$$

A P-presentation of a word W is a sequence $(U_0, \ldots, U_u)$ such that:

1. for $i \leq u$ P^2 is not a subword of U_i,
2. for $0 < i < u$ $P \neq U_i$,
3. for $0 < i \leq u$ P is a prefix of U_i,
4. for $0 \leq i < u$ P is a suffix of U_i,

and for some $\ell_1, \ldots, \ell_u$ we have

$$W = [U_0, \ldots, U_u](\ell_1, \ldots, \ell_u)$$

Note that only first condition is non-void if the presentation has $u = 0$.

The idea is that "powers of P" do not behave that well for small powers, i.e. single P, but they behave well for at least squares. Hence each (with some small exceptions at the beginning and end) U_i begins and ends with P.

Our main goal is to show that the P-presentation of a word is unique and that given a P-presentation of W, W' the P-presentation of WW' can be computed and that it depends only on U_u and V_0.

Theorem 7.4. *Given a primitive word P and a word W the P-presentation of W exists and it is unique; it can be computed greedily.*

Given P-presentations of W, W':

$$W = [U_0, \ldots, U_u](k_1, \ldots, k_u)$$
$$W' = [V_0, \ldots, V_v](\ell_1, \ldots, \ell_v)$$

the P-presentation of WW' is of one of the following forms, the form depends only on U_u and V_0.

- $WW' = [U_0, \ldots, U_{u-1}, V_1, \ldots, V_v](k_1, \ldots, k_{u-1}, k_u + \ell_1 + c, \ell_2, \ldots, \ell_v)$ *for some* $0 \leq c \leq 3$.
- $WW' = [U_0, \ldots, U_{u-1}, U', V_1, \ldots, V_v](k_1, \ldots, k_{u-1}, k_u + c, \ell_1 + c', \ell_2, \ldots, \ell_v)$ *for some* $0 \leq c, c' \leq 2$.
- $WW' = [U_0, \ldots, U_{u-1}, U', V', V_1, \ldots, V_v](k_1, \ldots, k_{u-1}, k_u + c, c', \ell_1 + c'', \ell_2, \ldots, \ell_v)$ *for some* $0 \leq c, c', c'' \leq 2$.

- $WW' = [U_0, \ldots, U_{u-1}, U', U'', V', V_1, \ldots, V_v](k_1, \ldots, k_{u-1}, k_u, 0, 0, \ell_1, \ell_2, \ldots, \ell_v)$.

The proof of the Theorem 7.4 is left as an exercise.

Example 7.2. $P = aabaa$, $W = aabaaaaba = Paaba$, $W' = abaaaabaa = abaaP$. Then WW' has a P-presentation $(P, aabaabaa, P)$. And this is the last case in the Theorem 7.4.

7.3 System of equations

We now allow parametrised P-presentations and parametrised words defined by them. The parametrised P-presentation can use variables instead of numbers for the powers of P. In terms of a solution: we fix a solution s and create a parametrised substitution S out of it. We then inspect the word equation and create a system of linear Diophantine equations, in variables that are used in the parametrised presentation for variables. Every solution of this system will give values for variables that turn the parametrised substitution into a true solution of the word equation.

We will use natural variable $\{L_X, R_X\}_{X \in \mathcal{X}}$ (in palce of first and last variables in the parametrized presentations of substitutions for variables) and an infinite set of variables $\{N_i\}$. Unless specifically said, each time we use a variable N_i, it is fresh, i.e. not used elsewhere.

Fix a solution s and a primitive word P. Let $s(X)$ has a P-presentation $[U_0, \ldots, U_u](k_1, \ldots, k_u)$. We create a parametrised substitution S defined on X as

$$S(X) = [U_0, \ldots, U_u](L_X, N_1, \ldots, N_{u-1}, R_X)$$

If $u = 0$ then it has no variables, if $u = 1$ then the only variable is L_X. Note that the indices in N are used only for illustration: those are variables not used elsewhere.

Let R be the right-hand side. Our goal is to calculate the P-presentation of $S(R)$. To this end we consider the consecutive prefixes $R' \sqsubseteq R$ and define the P-presentation of $S(R')$ for them.

- $R' = \epsilon$ and so it has a P-presentation (ϵ).
- When we have such a representation for $S(R')$ and we want to extend it to the P-presentation of $S(R'X)$ then by Theorem 7.4 the P-presentation of $S(R'X)$ is the presentation of $S(R')$ and $S(X)$ concatenated with some small changes in the middle.

  The parametrised P-presentation of $S(R'X)$ uses fresh variables and adds equations that formalise the equalities between old and new variables, according to Theorem 7.4. Note that at most 4 of them are not of the form $N_i = N_j$ and so at most 8 sides of the equation are other than the variables from $\{N_i\}$.
- we do a similar things for $S(R'w)$, where w is a word between variables on a side of the equation. Note that we need to compute the P-factorization of w.

In the end we get a P-factorisation of $S(R)$. We do the same for $S(L)$, where L is the left-hand side of the equation, and add equalities between corresponding variables.

In the end we get a set of linear equation $\mathcal{D}$. In total it has at most $8|uv|$ sides that are different than the variables from $\{N_i\}$.

Lemma 7.5. *For a word equation $L = R$ in the constructed system:*

- *variable L_X (R_X) is used at most $|LR|_X$ number of times*
- *at most $2n_v$ sides use a variable from $\{L_X, R_X\}_{X \in \mathcal{X}}$*
- *the sum of (absolute values of) constants is at most $|LR|/|P| + 6n_v$*

We can remove the variables N_i at the cosnt of increasing the above estiamtions twice.

Proof. A variable L_X, R_X is introduced when parsing X and is used once. For constants: the constants from processing words in the equation sum up to at most $|LR|/|P|$ and the constants introduced when merging P-representations introduce sum up to at most 6 per merge.

The variables N_i are never altered by the procedure. Hence the resulting system is equivalent to a one using only the non-trivial sides, though we might need to use such a side twice. □

Lemma 7.6. *For any prefix R' of R and a parametrised solution S let the parametrised P-presentation of $S(R')$*

$$[U_0, \ldots, U_\ell](\{L_X, R_X\}_{X\in\mathcal{X}}, \{N_i\})$$

For any numbers $\{\ell_X, r_X\}_{X\in\mathcal{X}}, \{n_i\}$ the P-presentation of $S[\{\ell_X, r_X\}_{X\in\mathcal{X}}, \{n_i\}](u')$ is

$$[U_0, \ldots, U_\ell](\{\ell_X, r_X\}_{X\in\mathcal{X}}, \{n_i\}) \ .$$

Each solution of the system $\mathcal{D}$ yields a solution of the word equation.

Lemma 7.7. *Each solution of $\mathcal{D}$ gives a solution of the word equation, obtained by replacing variables with their values in the P-presentations.*

7.4 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

$$\sum_{j=1}^{r} n_{i,j}x_j = n_i \qquad \text{for } i = 1, \ldots, m \ . \tag{7.2}$$

In the following, we are interested only in *natural* solutions, i.e. the ones in which each component is a natural number. 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 *minimal* if it satisfies (7.2) 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 7.8 (cf. [27, Corollary 4.4])**.** *For a system of linear Diophantine equations* (7.2) *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 modification of the original proof of Kościelski and Pacholski, we recall it [27].

proof, cf. [27]. The proof follows by estimation based on work of [66] and independently by [29]

Claim 7.8.1 (cf. [27])**.** *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. Let $q = (q_1, \ldots, q_r)$ be a minimal non-zero (i.e. there is a non-zero coordinate) solution. Then for each $1 \leq i \leq r$ we have $q_i \leq (w + r)M$.* □

So it remains to estimate M from Claim 7.8.1, we recall the argument of [27].

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.

7.5 Exponent of periodicity bound

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

As a first step, let us estimate the values w, r, c from Lemma 7.8 in case of system of equations $\mathcal{D}$

Lemma 7.9. *For a system of equations $\mathcal{D}$ Lemma 7.8 yields a bound of*

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

on coordinates of its minimal solutions.

This follows from Lemma 7.5.

Lemma 7.10 (cf. [27])**.** *Consider a solution s of a word equation $u = v$, and a system $\mathcal{D}$ created for it. Consider all solutions $\{\ell_X, r_X\}_{X\in\mathcal{X}}$ of this system and the corresponding solutions $S[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$. For a length-minimal s' among them the largest power w^p for a substring w^p of a solution word $s'(u)$ is $\mathcal{O}(\mathrm{poly}(n)e^{4n_v/e})$.*

Proof. We know that all $S[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$ are solutions. Let, as in the statement, s' be a length minimal among them, let it correspond to a solution $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ of $\mathcal{D}$. Then by definition ℓ'_X, (r'_X) are the first and last value in the P-presentations of $s'(X)$. Other values in the P-presentations linearly depend on them (with only non-negative coefficients and constants). We show that $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ is a minimal solution of D: suppose for the sake of contradiction that it is not. Then there is a solution $\{\ell''_X, r''_X\}_{X\in\mathcal{X}}$ of D, such that

$$\ell''_X \le \ell'_X \quad \text{and} \quad r''_X \le r'_X \quad \text{for each } X \in \mathcal{X} \tag{7.3}$$

and at least one of those inequalities is strict, without loss of generality let $\ell''_Y < \ell'_Y$.

Then the lengths of all substitutions $|s''(X)| \le |s'(X)|$ and for Y the equality is strict. Thus s' is not length-minimal, contradiction.

Now we can use Lemma 7.9 to get our bounds. □

As a short corollary we obtain:

Theorem 7.11 (cf. [27]). *The exponent of periodicity of a word equation $u = v$ with n_v occurrences of variables is $\mathcal{O}(\text{poly}(n)e^{4n_v/e})$.*

Proof. We can estimate the lengths of exponent of periodicity for each word P separately, by Lemma 7.10; note that it is enough to consider primitive words P. □

Exercises

Task 37 Show that the exponential bound on the exponent of periodicity is tight (the exact constant at the exponent is not tight, though).

Task 38 Show that we can use the P-presentation approach for the compression algorithm (for solving word equations): we do not guess the lengths of the a-prefixes and suffixes, but denote them as variables and we write an appropriate system of linear equations.

Show that when the word equation can be encoded using m bits (in a natural encoding) then the constructed system has size $\mathcal{O}(m)$ bits.

Hint: Unary encoding the constants, in which a constant p is encoded using p bits, may be easier for proof purposes, even though it is not efficient.

Task 39 Show that we can verify the system of linear Diophantine equations in which all constants are encoded in unary in linear space (counted in bits).

Hint: Repeatedly guess the parity of sides of all equations and divide by 2. This is a first step for decidability of Presburger's arithmetics.

Task 40 Using the bound on the size of the minimal solutions of integer programming show that the doubly exponential bound on the size of the length-minimal solution follows from the original algorithm for satisfying word equations.

Task 41 Prove Theorem 7.4.

Task 42 Show Hadamard inequality for a square matrix $N = (n_{i,j})_{i,j=1}^{k}$

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

Chapter 8

Word equations with one variable

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 [54]).

On the other hand, it was shown by Charatonik and Pacholski [4] 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.

8.1 One variable equations

It is easy to see that word equations equations with only one variable are 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 [49]. 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.

The latter work was not completely model-independent, as it assumed that the alphabet Σ is finite or that it can be identified with numbers (the Obono, Goralcik and Maksimenko [49] algorithm assumes pointer-machine model). A more general solution was presented by Laine and Plandowski [28], 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.

On the other hand, no equations with more than 3 solutions (except the infinite family) were known and it was conjectured that this is tight, i.e. that one variable word equations have at most 3 solutions (plus the infinite family). This was recently shown [48]. It is not known, whether this affects running time of any of the algorithms.

The compression-based algorithm for word equation can be specialised to one-variable case and its running time is then $\mathcal{O}(n \log \#_X)$; the running time can be improved to linear, at the expense of heavy usage of stringology data structures and combinatorial analysis [22].

8.2 One-variable equations: structure

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 begin with the same symbol (be it letter or variable), 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

$$A_0XA_1 \dots A_{n_\mathcal{A}-1}XA_{n_\mathcal{A}} = XB_1 \dots B_{n_\mathcal{B}-1}XB_{n_\mathcal{B}} \ ,$$

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. It the number of occurrences of variables at both sides are different then it is easy to show that there is at most one solution and it can be easily found (exercise). Similarly, if $A_{n_\mathcal{A}} \neq \epsilon$ then the equation can be split into two equivalent ones (and then joined in the reverse order, slightly more challenging exercise). Thus in the following we assume that the equation is of the form

$$A_0XA_1 \dots A_{n_X-1}X = XB_1 \dots B_{n_X-1}XB_{n_X} \ . \tag{8.1}$$

We first present the traditional approach to one-variable equations.

Lemma 8.1. *Given a word $x = u^i u'$, where $u' \preceq u$, testing whether it is a solution of a one-variable word equations can be done in time $\mathcal{O}(|u| + n)$ (in a pointer machine model).*

A proof is left as a simple exercise.

8.3 Via word combinatorics

8.3.1 Basic case

Lemma 8.2. *Given a word equation $Xp = qX$, if it is satisfiable then:*

- *p, q are conjugate and consequently also the primitive roots of p, q are conjugate, that is, there are u, v such that uv, vu are primitive and $p = (vu)^k$ and $q = (uv)^k$ for some $k \geq 1$;*
- *the set of solutions is $(uv)^*u$.*

Given p, q the u, v can be calculated in linear time.

Proof is left as an exercise

8.3.2 $|A_0| \leq |B_1|$

Let $B_0 \preceq B_1$, where $|B_0| = |A_0|$; then $A_0X = XB_0$. Hence by Lemma 8.2 the A_0 and B_0 are conjugate. We can calculate their primitives roots and so obtain u, v such that $A_0 = (uv)^m$, $B_0 = (vu)^m$ and $s(X) = (uv)^j u$ for some j and uv is primitive.

Lemma 8.3. *Given two words u, v such that uv is primitive solutions of (8.1) such that $s(X) = (uv)^i u$ can be found in time $\mathcal{O}(|uv| + n)$.*

Moreover, let s_j be defined as $s_j(X) = (uv)^j u$. Then among $s_1, \dots, s_n, \dots$ either none, one or all are solutions. This can be tested in linear time.

Proof. We treat $s_0(X) = u$ separately. Using Lemma 8.1 it is easy to test, whether it is a solution, in linear time.

We will calculate the (uv)-prefix of the solution word, let us begin with the left-hand side. We distinguish two cases: A_0 is and is not a power of uv; note that from case assumption we know that it actually is a power of uv, but the Lemma is later used also in more general setting, so we consider this case as well.

A_0 is not a power of uv

Claim 8.3.1. *Let $i \geq 1$. If A_0 is not a power of uv then the uv-prefix of $s_i(\mathcal{A})$ is the same as the uv-prefix of A_0uv.*

Observe first that $uv \sqsubseteq s_i(X)$ and so $A_0uv \sqsubseteq s_i(\mathcal{A})$. Let $A_0 = (uv)^k u'$ where uv is not a prefix of $u' \neq \epsilon$. If $|u'| \geq |uv|$ then we are done as the uv-prefix is $(uv)^k$. If the uv-prefix is $(uv)^{k+1}$ then it is in A_0uv and we are done. If it is at least $(uv)^{k+2}$ then it includes u' and it is continued by some v' such that $u'v' = uv$. But then the ending v' of the $k+1$st uv and the beginning u' of the $k+2$nd uv should also form uv, as they are equal to the last uv in A_0uv, contradiction with $u'v' = v'u' = uv$ which is primitive.

A_0 is a power of uv If A_0 is a power of uv then for all consecutive A_i which are of the form $v(uv)^*$ this prefix spans over them. Let A_j be the first which is not in $v(uv)^*$.

Claim 8.3.2. *Let $i \geq 1$. Let A_0 be a power of uv and all $A_{j'}$ for $j' < j$ are from the set $(vu)^*v$ and A_j is not. Then the uv-prefix of $s_i(\mathcal{A})$ is the same as the uv-prefix of $s_i(A_0XA_1 \cdots XA_juv)$.*

The argument is as in the case of Claim 8.3.1. Note that if there is no such j then the uv prefix span through the whole left-hand side.

The length of this prefix can be easily calculated in terms of i and j (and constants depending on $\mathcal{A}$)

We do a similar calculation for the right hand side, this time expressed in i and j', where $B_{j'}$ is the first of Bs that is not from the set $(vu)^*v$. A similar statement to Claim 8.3.2 holds.

Since the uv-prefixes of both sides must be equal, we obtain an equation for i, j, j'. Either it is not satisfiable (and there is no solution of this form) or it has exactly one solution or all numbers are a solution. In the second case we get one candidate $s(X) = (uv)^i u$ and it can be tested in linear time, Lemma 8.1. In the last case, we recursively deal with the remaining part of the equation (note that some care is needed at the ends, as the prefix could extend beyond the word). □

Lemma 8.4. *Given an equation (8.1) with $|A_0| \leq |B_1|$ in linear time we can return the set of solutions. It consists of $0, 1, 2$ or infinite number of solutions.*

8.3.3 $|s(X)| \geq |A_0| - |B_1| > 0$

We consider first the solutions in which $|s(X)| \geq |A_0| - |B_1| > 0$. Let A' be a prefix of A_0 of length $|A_0| - |B_1|$. Note that $A' \sqsubseteq s(X)$. Thus $A_0 s(X) = s(X) B_1 A'$. The rest of the argument is as in the case above; in particular, $s(X) = (uv)^k u$, where uv is the primitive root of A_0.

Lemma 8.5. *Given an equation (8.1) with $|A_0| > |B_1|$ in linear time we can return the set of solutions such that $|s(X)| \geq |A_0| - |B_1|$. It consists of $0, 1, 2$ or infinite number of solutions.*

8.3.4 $|A_0| - |B_1| > |s(X)| > 0$

The remaining cases are called *individual solutions*, all of them are of length smaller than $|A_0| - |B_1|$.

Let us prepend both sides of the equation with B_1. Then the right-hand sides begins with $(B_1X)^2$ and the left with B_1A_0X. As $|B_1 s(X)| \leq |A_0|$ it follows that $(B_1X)^2 \sqsubseteq B_1A_0A_0$.

Let P be the primitive root of $B_1 s(X)$. Then there are u, v such that $P = vu$ and

$$B_1 = (vu)^j v \quad \text{and} \quad s(X) = (uv)^i u \tag{8.2}$$

Moreover, $P^2 \sqsubseteq B_1A_0A_0$. There are at most $\mathcal{O}(\log |B_1A_0A_0|)$ such P and all of them can be found in linear $\mathcal{O}(|B_1A_0A_0|)$ time, see Theorem 6.17.

Now, for each such candidate P we can compare it with B_1 and obtain appropriate u, v. Then for each family of candidate solutions $s_i(X) = (uv)^i u$ we separately test s_0 in linear time and for the others we can use Lemma 8.3 to test others in linear time. This in total yields $\mathcal{O}(n \log n)$ running time for the algorithm (and this is roughly the solution of Obono, Goralcik and Maksimenko [49]).

This can be sped up: on one hand we show that in *total* linear time for each P we can reject all but two candidate solutions, thus we are left with $\mathcal{O}(\log n)$ candidate solutions. Then, assuming that

the alphabet is constant or contained in $\{1, 2, \ldots, n^c\}$ for a constant c so that RadixSort can be used on it, we can test a single candidate solution in $\mathcal{O}(n_X)$ time, see Section 8.3.5.

From the assumption on the length of the solution we get that

$$|A_0| > |s(x)| + |B_1| \geq |vu| = |P|$$

Let us consider the vu prefix of $B_1A_0A_0$. We first show that at most one of them spans through the whole $B_1A_0A_0$.

Lemma 8.6. *Suppose that the (vu)-prefix of $B_1A_0A_0$ contains at least $|vu|$ letters in the second A_0. Then uv is the primitive root of A_0.*

Proof. We know that $|A_0| > |uv|$ by the case assumption. Since B_1 ends with v, the A_0 begins with uv and this is not the whole A_0. Now, the second A_0 also begins with uv; the argument as in Claim 8.3.1 shows that the vu prefix cannot extend over the whole $|uv|$ first letters of the second occurrence of A_0, contradiction. Thus A_0 is the power of uv, so it is its primitive root. □

We verify this case (i.e. uv being the primitive root of A_0) separately using Lemma 8.3.

So in the following we can assume that the vu-prefix of $B_1A_0A_0$ ends before the first $|vu|$ letters of the second A_0; note that this is the same as the vu prefix of $B_1A_0s(X)$ as $|s(x)| \geq |uv|$ and $s(X) \sqsubseteq A_0$.

Lemma 8.7. *Given a set of primitive words $P_1, \ldots, P_k$ such that for each i $P_i^2 \sqsubseteq B_1A_0A_0$, in total time $\mathcal{O}(|B_1A_0A_0|)$ we can establish for all P_i from $P_1, \ldots, P_k$ the P_i-prefix of $B_1A_0A_0$.*

This can be done using the MP table, and is left as an exercise.

We now calculate the lengths of the P-prefixes of the right-hand side of the equation.

Lemma 8.8. *There are at most three different primitive $P = vu$ such that $B_1 = (vu)^j v$ for $j > 0$. Those candidates can be determined in linear time and for them the length of the vu-prefix of $s(\mathcal{B})$ can be determined in linear time.*

Proof.

- If B_1 has such a representation $(vu)^j v$ for $j \geq 2$ for two different P and P', where $|P| > |P'|$, in particular, P and P' are its periods. But then $|P| + |P'| < |B_1|$ and so there is a common smaller period w, moreover v is a power of v. Then also u is a power of v and this contradicts the primitivity of P, contradiction.
- If $B_1 = vuv$ then in particular $|P| \leq |B_1| < 2|P|$. But when $P_1, \ldots, P_i$ are all primitive square prefixes of $B_1A_0A_0$ then $|P_{j+2}| \geq 2|P_j|$.

In the second case the P satisfying this condition can be determined based only on the length: $|P| \leq |B_1| < 2|P|$ and there are at most two such Ps.

In the first case we need to use Lemma 8.7: using it we can establish the P for which the P-prefix of B_1 includes more than one Ps.

Then for each of those (at most 3) Ps we can establish the vu prefix of $s(\mathcal{B})$ in linear time: using an argument as in Claim 8.3.2 we are to look for the first B_k which is not in $(vu)^*v$, which can be done in linear time, and the vu-prefix of $s(\mathcal{B})$ is the vu-prefix of $s(B_1X \cdots XB_kvu)$ (or without the extra vu, when B_k is the last one).

Then the length of the vu prefix on the left-hand side is fixed and on the right-hand side it depends on $k|s(X)|$, in particular, it uniquely determines the length of $s(X)$. □

So we are left with the case in which $B_1 = v$. Note that this does not uniquely determines P, as u is not known. In this case we look for the first $B_k \neq v$. There are two cases: either the first such $B_k \in (vu)^+v$ or not. In the first case we use the same argument as in Lemma 8.8 to conclude that this can be for at most three different Ps and thus the vu-prefix can be also determined in linear time, as in Lemma 8.8.

So the last remaining case is that $B_k \neq v$ and it is not of the form $(vu)^{j'}v$ for any v. Then an argument as in Claim 8.3.2 shows that the vu-prefix of $s(\mathcal{B})$ is the same as vu-prefix of $s(XB_1XB_2X \cdots XB_kuv)$ (the special case that B_k is the last is handled separately). For a given P we can establish this by looking at the MP table of B_kuv. But as $uv \sqsubseteq A_0$, it can be established from the MP table of B_kA_0, moreover, all those calculations take in total $\mathcal{O}(B_kA_0)$ time. After that we can establish the length of the prefix on the right-hand side and determine the length of $s(X)$, as in Lemma 8.7.

8.3.5 Verification of candidate solutions

Lemma 8.9. *Using a suffix tree LCP data structure, one singular solution can be verified in $\mathcal{O}(\#_X)$ time; those data structures can be constructed in time $\mathcal{O}(n)$ time.*

8.4 Via recompression

If (8.1) 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 B_{n_X} is the *last word*. We additionally assume that none of words A_i, B_j is empty. We later (after Lemma 2.7) justify why this is indeed without loss of generality.

Note that if $s(X) \neq \epsilon$, then using (8.1) 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 8.10. *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 B_{n_X}.*

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 8.10 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 8.17.

A simple proof is left as an exercise.

By TestSimpleSolution(a) we denote a procedure, described in Lemma 8.10, 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.

8.4.1 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 8.11. *Each solution s of equation of the form (8.1) 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. □

8.4.2 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.

8.4.3 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 8.12 (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 8.12, 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.

8.4.4 Specialisation of procedures

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 8 $\mathsf{Pop}(a, b)$

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 8.13. $\mathsf{Pop}(a, b)$ *preserves solutions and after its application the pair ab is noncrossing.*

The only new part is the preservation of solutions. But this easily follows from Lemma 8.10.

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 9 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 Pop(a, b)

Lemma 8.14. 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 2.7. 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 8.10, 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 10.

Algorithm 10 Pop Cutting prefixes and suffixes; assumes that A_0 is not a block of letters

Require: A_0 is not a block of letters, the B_{n_X} is not a block of letters
1: let a be the first letter of $s(X)$
2: report solution found by 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 8.10 $s(X)$ 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 TestSimpleSolution(b) ▷ Exclude $s(X) \in b^+$ from further considerations.
7: let $r > 0$ be the length of the b-suffix of the B_{n_X}
▷ By Lemma 8.10 $s(X)$ 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 8.15. *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 as and the last not a block of bs then* Pop *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.

8.4.5 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 11 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(A) = s(B)
3:     BlockComp                          ▷ Compress blocks, in O(|A| + |B|) time.
4:     PreProc                            ▷ There are only two crossing pairs, see Lemma 8.14
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 ∈ Non-Crossing do      ▷ Compress non-crossing pairs, in time O(|A| + |B|)
8:         PairCompNCr(a, b)
9:     for ab ∈ Crossing do               ▷ Compress the 2 crossing pairs, in time O(|A| + |B|)
10:        PairComp(a, b)
11: TestSolution                          ▷ Test solutions from a*, see Lemma 8.17
```

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 $K = \mathcal{O}(1)$.

Recall, that for any 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.

Lemma 8.16. *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 **??** 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 **??**, 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 (8.1) 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 8.11. 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 8.17. *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.

8.4.6 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 8.18. *One phase of* OneVarWordEq *can be performed in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time.*

Proof. For grouping of pairs and blocks we use RadixSort, to this end it is needed that the alphabet of (used) letters can be identified with consecutive numbers, i.e. with an interval of at most $|\mathcal{A}| + |\mathcal{B}|$ integers. In the first phase of OneVarWordEq this follows from the assumption on the input. [1] At the end of this proof we describe how to bring back this property at the end of the phase.

To perform BlockComp we want for each letter a occurring in the equation to have lists of all maximal a-blocks occurring in $\mathcal{A} = \mathcal{B}$ (note that after Pop there are no crossing blocks, see Lemma 2.8). This is done by reading $\mathcal{A} = \mathcal{B}$ and listing triples (a, k, p), where k is the length of a maximal block of as and p is a pointer to the beginning of this occurrence. Notice, that the maximal block of a's may consist also of prefixes/suffixes that were cut from X by Pop. However, by Lemma 8.10 such a prefix is of length at most $|A_0| \leq |\mathcal{A}| + |\mathcal{B}|$ (and similar analysis applies for the suffix). Then each maximal block includes at most one such prefix and one such suffix thus the length of the a maximal block is at most $3(|\mathcal{A}| + |\mathcal{B}|)$. Hence, the triples (a, k, p) can be sorted by their first two coordinates using RadixSort in total time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$.

After the sorting, we go through the list of maximal blocks. 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, all blocks of the same length are consecutive on the list, and replacing them by the same letter is easily done.

To compress all non-crossing pairs, i.e. to perform the loop in line 8, we do a similar thing as for blocks: we read both $\mathcal{A}$ and $\mathcal{B}$, whenever we read a pair ab where $a \neq b$ and both a and b are not letters that replaced blocks during the blocks compression, we add a triple (a, b, p) to the temporary list, where p is a pointer to this position. Then we sort all these pairs according to lexicographic

[1] In fact, this assumption can be weakened a little: it is enough to assume that $\Sigma \subseteq \{1, 2, \ldots, \text{poly}(|\mathcal{A}| + |\mathcal{B}|)\}$: in such case we can use RadixSort to sort Σ in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ and then replace Σ with set of consecutive natural numbers.

order on first two coordinates, we use RadixSort for that. Since in each phase we number the letters occurring in $\mathcal{A} = \mathcal{B}$ using consecutive numbers, this can be done in time $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$. The occurrences of the crossing pairs can be removed from the list: by Lemma 8.14 there are at most two crossing pairs and they can be easily established (by looking at A_0XA_1). So we read the sorted list of pairs occurrences and we remove from it the ones that correspond to a crossing pair. Lastly, we go through this list and replaces pairs, as in the case of blocks. Note that when we try to replace ab it might be that this pair is no longer there as one of its letters was already replaced, in such a case we do nothing. This situation is easy to identify: before replacing the pair we check whether it is indeed ab that we expect there, as we know a and b, this is done in costant time.

We can compress each of the crossing pairs naively in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time by simply first applying the popping and then reading the equation form the left to the right and replacing occurrences of this fixed pair.

It is left to describe, how to enumerate (with consecutive numbers) letters in Σ at the end of each phase. Firstly notice that we can easily enumerate all letters introduced in this phase and identify them (at the end of this phase) with $\{1, \ldots, m\}$, where m is the number of introduced letters (note that none of them were removed during the OneVarWordEq). Next by the assumption the letters in Σ (from the beginning of this phase) are already identified with a subset of $\{1, \ldots, |\mathcal{A}| + |\mathcal{B}|\}$, we want to renumber them, so that the subset of letters from Σ that are present at the end of the phase is identified with $\{m+1, \ldots, m+m'\}$ for an appropriate m'. To this end we read the equation, whenever we spot a letter a that was present at the beginning of the phase we add a pair (a, p) where p is a pointer to this occurrence. We sort the list in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. From this list we can obtain a list of present letters together with list of pointers to their occurrences in the equation. Using those pointers the renumbering is easy to perform in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time.

So the total running time is $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. □

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 8.19. *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 8.18 (this is at most $\mathcal{O}(n)$ by Lemma 8.16) and as Lemma 8.16 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 8.17 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 8.20. *The running time of* OneVarWordEq *till one of first or last word becomes short is* $\mathcal{O}(n + n_X \log n)$.

Proof. By Lemma 8.18 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 8.16 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 $(\mathcal{A}, i)$-word is 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 $\mathcal{O}(n + n_X \log n)$. □

8.5 One variable: linear-time algorithm

The intuition gained from the analysis in the previous section, especially in Lemma 8.20 is that the main obstacle in obtaining the linear running time is the necessity of dealing with short words, as the time spend on processing them is difficult to charge. This applies to both the compression performed within the short words, which does not guarantee any reduction in length, see Lemma 8.16, and to testing of the candidate solutions, which cannot be charged to the length decrease of the whole equation.

Observe that by Lemma 8.19 as soon as the first or last word becomes short, the remaining running time is linear. Hence, in our improvements of the running time we can restrict ourselves to the case, in which the first and last word are long.

The improvement to linear running time is done by four improvements in algorithm analysis and employed data structures, which are described in details in the following subsections:

- *several equations:* Instead of a single equation, we store a system of several equations and look for a solution of such a system. This allows removal of some words from the equations that always correspond to each other and thus decreases the overall storing space and testing time. This is described in Section 8.5.2 and Section 8.5.3.
- *small solutions:* We identify a class of particularly simple solutions, called *small*, and show that a solution is reported within $\mathcal{O}(1)$ phases from the moment when it became small. In several problematic cases of the analysis we are able to show that the solutions involved are small and so it is easier to charge the time spent on testing them.
- *storage:* The storage is changed so that all words are represented by a structure of size proportional to the size of the *long words*. In this way the storage space decreases by a constant factor in each phase and so the running time (except for testing) is linear. This is explained in Section 8.5.3
- *testing:* The testing procedure is modified, so that the time it spends on the short words is reduced. In particular, we improve the rough estimate that one TestSimpleSolution takes time proportional to the equation to an estimation that actually counts for each word whether it was included in the test or not. Section 8.5.4 is devoted to this.

8.5.1 Suffix arrays and lcp arrays

We use a standard data structure for comparisons on strings: a suffix array $SA[1\,..\,m]$ for a string $w[1\,..\,m]$ stores the m non-trivial suffixes of w, that is $w[m], w[m-1\,..\,m], \ldots, w[1\,..\,m]$ in (increasing) lexicographical order. In other words, $SA[k] = p$ if and only if $w[p\ldots m]$ is the k-th suffix according to the lexicographical order. It is known that such an array can be constructed in $\mathcal{O}(m)$ time [24] assuming that RadixSort is applicable to letters, i.e. that they are integers from $\{1, 2, \ldots, m^c\}$ for some constant c. We assume explicitly that this is the case in our problem.

Using a suffix array the equality testing for substrings of w reduces to the *longest common prefix* (lcp) query: observe that $w[i\,..\,i+k] = w[j\,..\,j+k]$ if and only if the common prefix of $w[i\,..\,m]$ and $w[j\,..\,m]$ is at least k. The first step in constructing a data structure for answering such queries is the LCP array: for each $i = 1, \ldots, m-1$ the $LCP[i]$ stores the length of the longest common prefix of $SA[i]$ and $SA[i+1]$. Given a suffix array, the LCP array can be constructed in linear time [25], however, the linear-time construction of suffix arrays can be in fact extended to return also the LCP array [24].

When the LCP array is supplied, the general longest prefix queries reduce to the range minimum queries: the longest common prefix of $SA[i]$ and $SA[j]$ (for $i < j$) is the minimum among $LCP[i], \ldots, LCP[j-1]$, and so it is enough to have a data structure that answers the queries about the minimum in the range in constant time. Such data structures in general case are known and in case of LCP arrays even simpler constructions were given [2]. The construction time is linear and query time is $\mathcal{O}(1)$ [2]. Hence, after a linear preprocessing, we can calculate the length of the longest common prefix of two substrings of a given string in $\mathcal{O}(1)$ time.

8.5.2 Several equations

The improved analysis assumes that we do not store a single equation, instead, we store several equations and look for substitutions that simultaneously satisfy all of them. Hence we have a collection $\mathcal{A}_i = \mathcal{B}_i$ of equations, for $i = 1, \ldots, m$, each of them is of the form described by (8.1); by $\mathcal{A} = \mathcal{B}$ we denote the whole system of the equations. In particular, each of those equations specifies the first and last letter of the solution, length of the a-prefix and suffix etc., exactly in the same way as it does for a single equation. If there is a conflict, as two equations give different answers regarding the first/last letter or the length of the a-prefix or b-suffix, then there is no solution at all. Still, we do not check the consistency of all those answers, instead, we use an arbitrary equation, say $\mathcal{A}_1 = \mathcal{B}_1$, to establish the first, last letter, etc., and as soon as we find out that there is a conflict, we stop the computation and terminate immediately.

The system of equations stored by OneVarWordEq is obtained by replacing one equation $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ (where $\mathcal{A}'_i, \mathcal{A}''_i, \mathcal{B}'_i, \mathcal{B}''_i \in (\Sigma \cup \{X\})^*$) with equivalent two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ (note that in general the latter two equation are not equivalent to the former one, however, we perform the replacement only when they are; moreover, we need to trim them so that they satisfy the form (8.1)).

The described way of splitting the equations implies a natural order on the equations in the system: when $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ is split to $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ then $\mathcal{A}'_i = \mathcal{B}'_i$ is before $\mathcal{A}''_i = \mathcal{B}''_i$ (moreover, they are both before/after each equation before/after which $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ was). This order is followed whenever we perform any operations on all words of the equations. We store a list of all equations, in this order.

We store each of the equations in the same way as described for a single equation in the previous phase, i.e. for an equation $\mathcal{A}_i = \mathcal{B}_i$ we store a list of pointers to words on one side and list of pointers to words on the other side. Additionally, the first word of $\mathcal{A}_i$ has a link to the last word of $\mathcal{A}_{i-1}$ and the last word of $\mathcal{A}_i$ similarly, the last word of $\mathcal{A}_i$ has a link to the first word of $\mathcal{A}_i$ and the first word of $\mathcal{A}_{i+1}$. We also say that A_i (B_j) is first or last if it is in any of the stored equations.

All operations on a single equation introduced in the previous sections (popping letters, cutting prefixes and suffixes, pair compression, blocks compression) generalise to a system of equations. The running times are addressed in detail later on. Concerning the properties, they are the same, we list those for which the generalisation or the proof are non-obvious: PreProc should ensure that there are only two crossing pairs. This is the case, as each X in every equation is replaced by the same aXb and $s(X)$ is the same for all equations, which is the main fact used in the proof of Lemma 8.14. Lemma 8.16 ensured that in each phase the length of the first and last word is decreased. Currently the first words in each equation may be different, however, the analysis in Lemma 8.16 applies to each of them.

8.5.3 Storing of an equation

To reduce the running time we store duplicates of short word only once. Recall that for each equation we store lists of pointers pointing to strings that are the explicit words in this equation. We store the long words in a natural way, i.e. each long word is represented by a separate string. The short words are stored more efficiently: if two short words in equations are equal we store only one string, to which both pointers point. In this way all identical short words are stored only once (though each of them has a separate pointer pointing to it); we call such a representation *succinct*.

We show that the compression can be performed on the succinct representation, without the need of reading the actual equation. This allows bounding the running time using the size of the succinct representation and not the equation.

We distinguish two types of short words: those that are substrings of long words (normal) and those that are not (overdue). We can charge the cost of processing the normal short words to the time of processing the long words. The overdue words can be removed from the equation after $\mathcal{O}(1)$ phases after becoming overdue, so their processing time is constant per $(\mathcal{A}, i)$-word (or $(\mathcal{B}, j)$-word).

The rest of this subsection is organised as follows:

- We first give precise details, how we store short and long words, see Section 8.5.3 and prove that we can perform compression using only succinct representation, see Lemma 8.21.

- We then define precisely the normal and overdue words, see Section 8.5.3 as well as show that we can identify new short and overdue words, see Lemma 8.23. Then we show that overdue words can be removed $\mathcal{O}(1)$ phases after becoming overdue, see Lemma 8.24 and 8.25.

- Lastly, in Section 8.5.3, we show that the whole compression time, summed over all phases is $\mathcal{O}(n)$. The analysis is done separately for long words, normal short words and overdue short words.

As observed in Lemma 8.19, as soon as the first or last word becomes short, the remaining running time is linear. Thus, when such a word becomes short, we drop our succinct representation and recreate out of it the simple representation. Such a recreation takes linear time.

Storing details

We give some more details about the storing: All long words are stored on two doubly-linked lists, one representing the long words on the left-hand sides and the other the long words on the right-hand sides. Those words are stored on the lists according to the initial order of the words in the input equation. Furthermore, for each long word we store additionally, whether it is a first or last word of some equation (note that a short word cannot be first or last). The short words are also organised as a list, the order on the list is irrelevant. Each short word has a list of its occurrences in the equations, the list points to the occurrences in the natural order (occurrences on the left-hand sides and on the right-hand sides are stored separately).

We say that such a representation is *succinct* and its size is the sum of lengths of words stored in it (so the sum of sizes of long words, perhaps with multiplicities, plus the sum of sizes of different short words). Note that we do *not* include the number of pointers from occurrences of short words. We later show that in this way we do not need to actually read the whole equation in order to compress it; it is enough to read the words in the succinct representation, see Lemma 8.22.

We now show that such a storage makes sense, i.e. that if two short words become equal, they remain equal in the following phases (note again that none of them are first, nor last).

Lemma 8.21. *Consider any explicit words A and B in the input equation. Suppose that during* OneVarWordEq *they were transformed to $A' = B'$, none of which is a first or last word in one of the equations. Then $A = B$ if and only if $A' = B'$.*

Proof. By induction on operation performed by OneVarWordEq. Since none of the A', B' is the first or last word in the equation, it means that during the whole OneVarWordEq they had X to the left and to the right. So whenever a letter was left-popped or right-popped from X, it was prepended or appended to both A and B; the same applies to cutting prefixes and suffixes. Compression is never applied to a crossing pair or a crossing block, so after it two strings are equal if and only if they were before the operation. The removal of letters (in order to preserve (8.1)) is applied only to first and last words, so it does not apply to words considered here. Partitioning the equation into subequations does not affect the equality of explicit words. □

We now show the main property of succinct representation: the compression (both pair and block) can be performed on succinct representation in linear time.

Lemma 8.22. *The compression in one phase of* OneVarWordEq *can be performed in time linear in size of the succinct representation.*

Proof. Let us recall what operations we need to perform and what changes are needed when comparing with the case of one equation, see Lemma 8.18 We comment on the case of pair compression, the case of blocks compression is done in a similar way.

Observe first, that from Lemma 8.21 it follows that if an explicit short word A occurs twice in the equations (both times not as a first, nor last word of the equation) it is changed during OneVarWordEq in the same way at both those instances. This justifies our approach of performing the operations on the words stored in the list of short words and not separately on each occurrence in the equations.

First, we perform the preprocessing, to this end we need to know the first (a) and last (b) letter, this is done by looking at the first and last word. Then we prepend b and append a to each word, except those that are first or last (first ones get only a and last ones only b). To this end we go through the list of long words and short words and append appropriate letters, note that each word stores an information, whether it is first or last, so we always know, whether to prepend or append.

Now we need to list the pairs that occur in the equation, again, this is done by going through the list. As each pair occurs in one of the words, the total size is proportional to the size of the succinct representation. Sorting then also is done in linear time (note that the size of the alphabet is at most the size of the succinct representation: each letter needs to occur somewhere).

To establish the crossing pair, it is enough to look at A_iXA_{i+1}, where A_i is any of the first words, after establishing this we filter out the crossing pair by going through the sorted list. Lastly, we perform the compression, using pointers to localise the occurrences of ab to be replaced. The compression of crossing pairs is done while reading the whole succinct representation, so also in linear time. □

Normal and overdue short words

The short words stored in the tables are of two types: normal and overdue. The *normal* words are substrings of the long words or A_0^2 and consequently the sum of their sizes is proportional to the size of the long words. A word becomes *overdue* if at the beginning of the phase it is not a substring of a long word nor A_0^2. It might be that it becomes a substring of such a word later, it does not stop to be an overdue word in such a case.

Since the normal words are of size $\mathcal{O}(K) = \mathcal{O}(1)$, the sum of lengths of normal words stored in short word list is at most $\mathcal{O}(1)$ larger than the sum of sizes of the long words. Hence the processing time of normal short words can be charged to the long words. For the overdue words the analysis is different: we show that after $\mathcal{O}(1)$ phases we can remove them from the equation (splitting the equations). Thus their processing time is $\mathcal{O}(1)$ per $(\mathcal{A}, i)$-word (or $(\mathcal{B}, j)$-word), so summed over all words it yields $\mathcal{O}(n_X + n_{\mathcal{B}}) = \mathcal{O}(n)$ in total.

The new overdue words can be identified in linear time: this is done by constructing a suffix array for a concatenation of long and short words occurring in the equations.

Lemma 8.23. *In time proportional to the size of succinct representation size we can identify the new overdue words.*

Proof. Consider all long words $A_0, \ldots, A_m$ (with or without multiplicities, it does not matter) and all short (not yet overdue) words $A'_1, \ldots A'_{m'}$, without multiplicities; in both cases this is just a listing of words stored in the representation (except for old overdue words). We construct a suffix array for the string

$$A_0^2\$A_1\$\ldots A_m\$A'_1\$\ldots A'_{m'}\# \ .$$

As it was already observed that the size of the alphabet is linear in the size of the succinct representation, inside the proof of Lemma 8.22, the construction of the suffix array can be done in linear time [24].

Now A'_i is a factor in some A_j (the case of A_0^2 is similar, it is omitted to streamline the presentation) if and only if for some suffix A''_j of A_j the strings $A''_j\$A_{j+1}\ldots A_m\$A'_1\$\ldots\$A'_{m'}\#$ and $A'_i\$\ldots\$A'_{m'}\#$ have a common prefix of length at least $|A'_i|$. In terms of the constructed suffix array, the entries for $A'_i\$\ldots\$A'_{m'}\#$ and $A''_j\$A_{j+1}\ldots\$A_m\$A'_1\$\ldots\$A'_{m'}\#$ should have a common prefix of length at least $|A'_i|$. Recall that the length of the longest common prefix of two suffixes stored at positions $p < p'$ in the suffix array is the minimum of $LCP[p]$, $LCP[p+1]$, …, $LCP[p'-1]$.

For fixed suffix $A'_i\$\ldots\$A'_{m'}\#$ we want to find $A''_j\$A_{j+1}\ldots\$A_m\$A'_1\$\ldots\$A'_{m'}\#$ (where A''_j is a suffix of some long word A_j) with which it has the longest common prefix. As the length of the common prefix of pth and p'th entry in a suffix array is $\min(LCP[p], LCP[p+1], \ldots, LCP[p'-1])$, this is is either the first previous or first next suffix of this form in the suffix array. Thus the appropriate computation can be done in linear time: we first go down in the suffix array, storing the last spotted entry corresponding to a suffix of some long A_j, calculating the LCP with consecutive suffixes and storing them for the suffixes of the form $A'_i\$\ldots\$A'_{m'}\#$. We then do the same going from the bottom

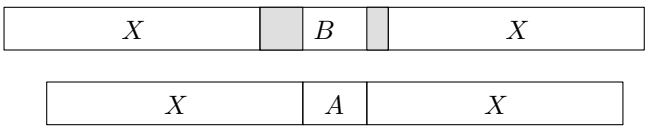

Figure 8.1: A is arranged against B. The periods of length at most $|B| - |A|$ are in ligther grey. Since $A \neq B$, at least one of them is non-empty.

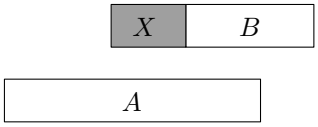

Figure 8.2: Subword of A_i is arranged against the whole $s(X)$.

of the suffix array. Lastly, we choose the larger from two stored values; for $A'_i\$ \dots \$A'_{m'}\#$ it is smaller than $|A'_i|$ if and only if A'_i just became an overdue word.

Concerning the running time, it linearly depends on the size of the succinct representation and alphabet size, which is also linear in size of succinct representation, as claimed. □

The main property of the overdue words is that they can be removed from the equations in $\mathcal{O}(1)$ phases after becoming overdue. This is shown by a series of lemmata.

First we need to define what does it mean that for solution word A in one side of the equation is at the same position as its copy on the other side of the equation: we say that for a substitution s the explicit word A_i (or its subword) is *arranged against* the explicit word B_j ($s(X)$ for some fixed occurrence of X) if the position within $s(\mathcal{A}_k)$ occupied by this explicit word A_i (or its subword) are within the positions occupied by explicit word B_j ($s(X)$, respectively) in $\mathcal{B}_k$.

Lemma 8.24. *Consider a short word A in a phase in which it becomes overdue. Then for each solution $s(X)$ either s is small or in every $s(\mathcal{A}_k) = s(\mathcal{B}_k)$ each explicit word A_i equal to A is arranged against another explicit word B_j equal to A.*

Proof. Consider an equation and a solution s such that in some $s(\mathcal{A}_k) = s(\mathcal{B}_k)$ an explicit word A_i (equal to an overdue word A) is not arranged against another explicit word equal to A. There are three cases:

A is arranged against $s(X)$ Note that in this case A is a substring of $s(X)$. Either $s(X)$ is a substring of A_0 or $s(X) = A_0^k A'_0$, where A'_0 is a prefix of A_0. In the former case A is a factor of A_0, which is a contradiction, in the latter it is a factor of A_0^{k+1}. As A_0 is long and A short, it follows that $|A| < |A_0|$ and so A is a factor of A_0^2, contradiction with the assumption that A is overdue.

A is arranged against some word Since A is an overdue word, this means that A_i is arranged against a short word B_j. Note that both A_i and B_j are preceded and succeeded by $s(X)$, since $A_i \neq B_j$ we conclude that $s(X)$ has a period at most $|B_j| - |A_i|$, see Fig. 8.1; in particular s is small.

Other case Since A_i is not arranged against any word, nor arranged against $s(X)$, it means that some substring of A_i is arranged against $s(X)$ and as A_i is preceded and succeeded by $s(X)$, this means that either $s(X)$ is shorter than A_i or it has a period at most $|A_i|$, see Figure 8.2 and 8.3, respectively. In both cases s is small. □

Observe that due to Theorem **??** and Lemma 8.24 the $(\mathcal{A}, i)$-words and $(\mathcal{B}, j)$-words that are overdue can be removed in $\mathcal{O}(1)$ phases after becoming overdue: suppose that A becomes an overdue word in phase ℓ. Any solution, in which an overdue word A is not arranged against another occurrence of A is small and so it is reported after $\mathcal{O}(1)$ phases. Consider an equation $\mathcal{A}_i = \mathcal{B}_i$ in which A occurs. Then the first occurrence of A in $\mathcal{A}_i$ and the first occurrence of A in $\mathcal{B}_j$ are arranged against each other for each solution s. In particular, we can write $\mathcal{A}_i = \mathcal{B}_i$ as $\mathcal{A}'_i X A X \mathcal{A}''_i = \mathcal{B}'_i X A X \mathcal{B}''_i$, where $\mathcal{A}_i$

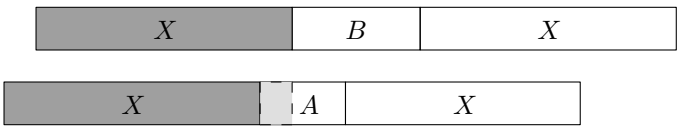

Figure 8.3: Subword of A_i is arranged against $s(X)$. The overlapping $s(X)$ are in in grey, the $s(X)$ has a period shorter than A_i, the period is depicted in lighter grey.

and $\mathcal{B}_i$ do not have A as an explicit word (recall that A is not the first, nor the last word in $\mathcal{A}_i = \mathcal{B}_i$). This equation is equivalent to two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$. This procedure can be applied recursively to $\mathcal{A}''_i = \mathcal{B}''_i$. In this way, all occurrences of A are removed and no solutions are lost in the process. There may be many overdue strings so the process is a little more complicated, however, as each word can be removed once during the whole algorithm, in total it takes $\mathcal{O}(n)$ time.

Lemma 8.25. *Consider the set of overdue words introduced in phase ℓ. Then in phase $\ell + \mathcal{O}(1)$ we can remove all occurrences of these overdue words from the equations.*

The obtained set of equations has the same set of solutions. The amortised time spend on removal of overdue words, over the whole run of OneVarWordEq*, is $\mathcal{O}(\#_X)$.*

Proof. Consider any word A that become overdue in phase ℓ and any solution s of this equation, such that in some $s(\mathcal{A}_i) = s(\mathcal{B}_i)$ the explicit word A is not arranged against another instance of the same explicit word. Then due to Lemma 8.24 the $s(X)$ is small. Consequently, from Theorem **??** this solution is reported before phase $\ell + c$, for some constant c. So any solution s' in phase $\ell + c$ corresponds to a solution s from phase ℓ that had each explicit word A arranged in each $s(\mathcal{A}_i) = s(\mathcal{B}_i)$ against another explicit word A. Since all operations in a phase either transform solution, implement the pair compression of implement the blocks compression for a solution $s(X)$, it follows that in phase $\ell+c$ the corresponding overdue words A' are arranged against each other in $s'(\mathcal{A}'_i) = s'(\mathcal{B}'_i)$. Moreover, by Lemma 8.21 each explicit word A' in this phase corresponds to an explicit word A in phase ℓ, i.e. there are no 'new' copies of A' (recall that hte fisrt and last words are long).

This observation allows removing all overdue words introduced in phase ℓ. Let C_1, C_2, ..., C_m (in phase $\ell + c$) correspond to all overdue words introduced in phase ℓ. By Lemma 8.23 we have already identified the overdue words. Using the list of short words, for each overdue word C, we have the list of pointers to occurrences of C in left-hand sides of the equations and right-hand sides of the equations, those lists are sorted according to the order of occurrences. In phase $\ell + c$ we go through those lists, if the first occurrences of A in the left-hand sides and right-hand sides are in different equations then the equations are not satisfiable, as this would contradict that in each solution both A is arranged against its copy. Otherwise, they are in the same equation $\mathcal{A}_i = \mathcal{B}_i$, which is of the form $\mathcal{A}'_i X A X \mathcal{A}''_i = \mathcal{B}'_i X A X \mathcal{B}''_i$, where $\mathcal{A}'_i$ and $\mathcal{B}'_i$ do not have any occurrence of A within them. We split $\mathcal{A}_i = \mathcal{B}_i$ into two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ and we trim them so that they are in the form described in (8.1). Clearly each solution of the new system of equation is also a solution of the old system, on the other hand, in any solution of the old system the copies of A were arranged against its copy, so the solution also satisfies the created equationss.

Note that as new equations are created, we need to reorganise the pointers from the first/last words in the equations, however, this is easily done in $\mathcal{O}(1)$ time. The overall cost can be charge to the removed X, which makes in total at most $\mathcal{O}(\#_X)$ cost. □

Compression running time

Lemma 8.26. *The running time of* OneVarWordEq*, except for time used to test the solutions, is $\mathcal{O}(n)$.*

Proof. By Lemma 8.22 the cost of compression is linear in terms of the size of the succinct representation by Lemma 8.23 in the same time bounds we can also identify the overdue words. Lastly, by Lemma 8.24 the total cost of removing the overdue words is $\mathcal{O}(n)$. So it is enough to show that the sum of sizes of the succinct representations summed over all phases is $\mathcal{O}(n)$.

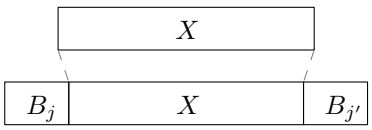

Figure 8.4: Let B_j and $B_{j'}$ both have their letters arranged against letters from fixed occurrence of X. Then the X separating them is a proper substring of another X, contradiction.

When the overdue words are excluded, the size of the succinct representation is proportional to the total length of long words. Since by Lemma 8.16 this sum of lengths decreases by a constant in each phase, the sum of those costs is linear in n.

Concerning the costs related to the overdue words: Note that an $(\mathcal{A}, i)$-word or $(\mathcal{B}, j)$-word is overdue for only $\mathcal{O}(1)$ phases, after which it is deleted from the equation see Lemma 8.25. So in $\mathcal{O}(1)$ phases it is charged $\mathcal{O}(K) = \mathcal{O}(1)$ cost, during the whole run of OneVarWordEq. Summing over all $(\mathcal{A}, i)$-words and $(\mathcal{A}, i)$-words yields $\mathcal{O}(n)$ time. □

8.5.4 Testing

We already know that thanks to appropriate storing the compression of the equations can be performed in linear time. It remains to explain how to test the solutions fast, i.e. how to perform TestSimpleSolution when all first and last words are still long.

Recall that TestSimpleSolution checks whether s,which is of the form $s(X) = a^\ell$ for some ℓ, is a solution by comparing $s(\mathcal{A}_i)$ and $s(\mathcal{B}_i)$ letter by letter, replacing X with a^ℓ on the fly. We say that in such a case a letter b in $s(\mathcal{A}_i)$ is *tested against* the corresponding letter in $s(\mathcal{B}_i)$. Note that during the testing we do not take advantage of the smaller size of the succinct representation, so we need to make a separate analysis. Consider two letters, from A_i and B_j, that are tested against each other. If one of A_i and B_j is long, this can be amortised against the length of the long word. The same applies when one of the words A_{i+1} or B_{j+1} is long. So the only problematic case is when all of those words are short. To deal with this case efficiently we distinguish between different test types, in which we exploit different properties of the solutions to speed up the tests. In the end, we show that the total time spent on testing is linear.

For a substitution s by a *mismatch* we denote the first position on which s is shown not be a solution, i.e. sides of the equation have different letters (we use a natural order on the equations); clearly, a solution has no mismatch. Furthermore, OneVarWordEq stops the testing as soon as it finds a mismatch, so in the rest of this section, if we use a name *test* for a comparison of letters, this means that the compared letters are before the mismatch (or that there is no mismatch at all).

There are two preliminary technical remarks: First we note that when testing a substitution s, for a fixed occurrence of X there is at most one explicit word whose letters are tested against letters from this occurrence of X.

Lemma 8.27. *Fix a tested substitution s and an occurrence of X in the equation. Then there is at most one explicit word whose letters are arranged against letters from this fixed occurrence of $s(X)$.*

Proof. Without loss of generality assume that X occurs within $\mathcal{A}_\ell$ in an equation $\mathcal{A}_\ell = \mathcal{B}_\ell$. Suppose that B_j and $B_{j'}$ (for $j' > j$) have their letters arranged against a letter from this fixed occurrence of $s(X)$, see Fig 8.4. But B_j and $B_{j'}$ are separated by at least one X in the equation, and whole this X is also arranged against this fixed occurrence of X, contradiction. □

As a second remark, observe that tests include not only explicit letters from $s(\mathcal{A}_\ell)$ and $s(\mathcal{B}_\ell)$ but also letters from $s(X)$. In the following we will focus on tests in which at least one letter comes from an explicit word. It is easy to show that the time spent on other tests is at most as large as time spent on those tests. This follows from the fact that such other tests boil down to comparison of long blocks of a and the previous test is of a different type, so we can account the comparison between two long blocks of a to the previous test. However, our fast testing procedures in some times makes a series of tests in $\mathcal{O}(1)$ time, so this argument can be made precise only after the explanation of the details of

various testing optimisations. For this reason the proof of Lemma 8.28 is delayed till the end of this section.

Lemma 8.28. *Suppose that we can perform all tests in which at least one letter comes from an explicit word in $\mathcal{O}(n)$ time. Then we can perform all test in $\mathcal{O}(n)$ time.*

Thus, in the following of this section we consider only the tests in which at least one letter comes from an explicit word.

Test types

Suppose that for a substitution s a letter from A_i is tested against a letter from $s(XB_j)$ or a letter from B_j is tested against a letter from $s(XA_i)$ (the special case, when there is no explicit word after X is explained later). We say that this test is:

- *protected:* if at least one of A_i, A_{i+1}, B_j, B_{j+1} is long;
- *failed:* if A_i, A_{i+1}, B_j and B_{j+1} are short and a mismatch for s is found till the end of A_{i+1} or B_{j+1};
- *aligned:* if $A_i = B_j$ and $A_{i+1} = B_{j+1}$, all of them are short and the first letter of A_i is tested against the first letter of B_j;
- *misaligned:* if all of A_i, A_{i+1}, B_j, B_{j+1} are short, $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ and this is not an aligned nor failed test;
- *periodical:* if $A_{i+1} = A_i$, $B_{j+1} = B_j$, all of them are short and this is not an aligned nor failed test.

So far this classification does not apply to the case, when a letter from A_i is tested against letter from X that is not followed by an explicit word. There are two cases:

- If A_i is not followed by X in the equation then A_i is a last word, in particular it is long. Therefore this test is protected.
- If A_i is followed by X then there is a mismatch till the end of A_iX, so this test is failed.

Observe that 'failed test' does not mean a mismatch, just a fact that soon there will be a mismatch. The protected, misaligned and failed tests are done in a letter-by-letter way, while the aligned and periodical tests are made in larger groups (in $\mathcal{O}(1)$ time per group, this of course means that we use some additional data structures).

It is easy to show that there are no other tests, see Lemma 8.29. We separately calculate the cost of each type of tests. As some tests are done in groups, we distinguish between number of tests of a particular type (which is the number of letter-to-letter comparisons) and the time spent on test of a particular type (which may be smaller, as group of tests are performed in $\mathcal{O}(1)$ time); the latter includes also the time needed to create and sustain the appropriate data structures.

For failed tests note that they take constant time per phase and we know that there are $\mathcal{O}(\log n)$ phases. For protected tests, we charge the cost of the protected test to the long word and only $\mathcal{O}(|C|)$ such tests can be charged to one long word C in a phase. On the other hand, each long word is shortened by a constant factor in a phase, see Lemma 8.16, and so this cost can be charged to those removed letters and thus the total cost of those tests (over the whole run of OneVarWordEq) is $\mathcal{O}(n)$.

In case of the misaligned tests, it can be shown that s in this case is small and that it is tested at the latest $\mathcal{O}(1)$ phases after the last of A_i, A_{i+1}, B_i, B_{i+1} becomes short, so this cost can be charged to, say, B_i becoming short and only $\mathcal{O}(1)$ such tests are charged to this B_i (over the whole run of the algorithm). Hence the total time of such tests is $\mathcal{O}(n)$.

For the aligned tests, consider the consecutive aligned tests, they correspond to comparison of $A_iXA_{i+1}\dots A_{i+k}X$ and $B_jXB_{j+1}\dots B_{j+k}X$, where $A_{i+\ell} = B_{j+\ell}$ for $\ell = 1, \dots, k$. So to perform them efficiently, it is enough to identify the maximal (syntactically) equal substrings of the equation

and from Lemma 8.21 it follows that this corresponds to the (syntactical) equality of substrings in the original equation. Such an equality can be tested in $\mathcal{O}(1)$ using a suffix array constructed for the input equation (and general lcp queries on it). To bound the total running time it is enough to notice that the previous test is either misaligned or protected. There are $\mathcal{O}(n)$ such tests in total, so the time spent on aligned tests is also linear.

For the periodical test suppose that we are to test the equality of (suffix of) $s((A_iX)^\ell)$ and (prefix of) $s(X(B_jX)^k)$. If $|A_i| = |B_j|$ then the test for A_{i+1} and B_{j+1} is the same as for A_i and B_j and so can be skipped. If $|A_i| > |B_j|$ then the common part of $s((A_iX)^\ell)$ and $s(X(B_jX)^k)$ have periods $|s(A_iX)|$ and $|s(B_jX)|$ and consequently has a period $|A_i| - |B_j| \leq K$. So it is enough to test first common $|A_i| - |B_j|$ letters and check whether $|s(A_iX)|$ and $|s(B_jX)|$ have period $|A_i| - |B_j|$, which can be checked in $\mathcal{O}(1)$ time.

This yields that the total time of testing is linear. The details are given in the next subsections.

We begin with showing that indeed each test is either failed, protected, aligned, misaligned or periodical.

Lemma 8.29. *Each test is either failed, protected, misaligned, aligned or periodical. Additionally, whenever a test in made, in $\mathcal{O}(1)$ time we can establish, what type of test this is.*

Proof. Without loss of generality, consider a test of a letter from A_i and from $s(XB_j)$. If any of A_{i+1}, B_{j+1}, A_i or B_j is long then it is protected (this includes the case in which some of A_{i+1}, B_j, B_{j+1} does not exist). Concerning the running time, for each explicit word we keep a flag, whether it is short or long. Furthermore, as each explicit word has a link to its successor and predecessor, we can establish whether any of A_{i+1}, B_{j+1}, A_i or B_j is long in $\mathcal{O}(1)$ time.

So consider the case in which all A_{i+1}, B_{j+1}, A_i and B_j (if they exist) are short, which also can be established in $\mathcal{O}(1)$ time. It might be that this test is failed (again, some of the words A_{i+1}, B_j, B_{j+1} may not exist), too see this we need to make some look-ahead tests, but this can be done in $\mathcal{O}(K)$ time (we do not treat those look-aheads as tests, so there is not recursion here).

Otherwise, if the first letter of A_i and B_j are tested against each other and $A_i = B_j$ and $A_{i+1} = B_{j+1}$ then the test is aligned (clearly this can be established in $\mathcal{O}(1)$ time using look-aheads). Otherwise, if $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ then it is misaligned (again, $\mathcal{O}(1)$ time for look-aheads). In the remaining case $A_{i+1} = A_i$ and $B_{j+1} = B_j$, so this is a periodical test. □

Failed tests

We show that in total there are $\mathcal{O}(\log n)$ failed tests. This follows from the fact that there are $\mathcal{O}(1)$ substitutions tested per phase and there are $\mathcal{O}(\log n)$ phases.

Lemma 8.30. *The number of all failed tests is $\mathcal{O}(\log n)$ over the whole run of* OneVarWordEq.

Proof. As noticed, there are $\mathcal{O}(1)$ substitutions tested per phase. Suppose that the mismatch is for the letter from A_i and a letter from XB_j (the case of XA_i and B_j is symmetrical). Then the failed tests include at least one letter from $XA_{i-1}XA_i$ or $XB_{j-1}XB_jX$, assuming they come from a short word. There are at most $4K$ failed tests that include a letter from A_{i-1}, A_i, B_{j-1}, B_j (as if the test is failed then in particular this explicit word is short). Concerning the tests including the occurrences of X in-between them, observe that by Lemma 8.27 each such X can have tests with at most one short word, so this gives additional $5K$ tests. Since $K = \mathcal{O}(1)$, we conclude that there are $\mathcal{O}(1)$ failed tests per phase and so $\mathcal{O}(\log n)$ failed tests in total, as there are $\mathcal{O}(\log n)$ phases, see Lemma 8.16. □

Protected tests

As already claimed, the total number of protected tests is linear in terms of length of long words: to show this it is enough to charge the cost of the protected test to the appropriate long word and see that a long word A can be charged only $|A|$ such tests for test including letters from A and $\mathcal{O}(1)$ letters from neighbouring short words, which yields $\mathcal{O}(|A|)$ tests. As the length of the long words drops by a constant factor, summing this up over all phases in which this explicit word is long yields $\mathcal{O}(n)$ tests in total.

Lemma 8.31. *In one phase the total number of protected tests is proportional to the length of the long words. In particular, there are $\mathcal{O}(n)$ such test during the whole run of* OneVarWordEq.

Proof. As observed in Lemma 8.28 we can consider only tests in which at least one letter comes from an explicit word. Suppose that a letter from A_i takes part in the protected test (the argument for a letter from B_j is similar, it is given later on) and it is tested against a letter from XB_j, then one of A_i, A_{i+1}, B_j, B_{j+1} is long, we charge the cost according to this order, i.e. we charge it to A_i if it is long, if A_i is not but A_{i+1} is long, we charge it to A_{i+1}, if not then to B_j if it is long and otherwise to B_{j+1}. The analysis and charging for a test of a letter from B_j is done in a symmetrical way (note that when the test includes two explicit letters, we charge it twice, but this is not a problem).

Now, fix some long word A_i, we estimate, how many protected tests can be charged to it. It can be charged with cost of tests that include its own letters, so $|A_i|$ tests. When A_{i-1} is short, it can also charge tests in which its letters take part. As it is short, it is at most $\mathcal{O}(K) = \mathcal{O}(1)$ such tests.

Also some words from $\mathcal{B}$ can charge the cost of tests to A_i, we can count only the test in which letters from A_i do not take part. This can happen in two situations: letters tested against XA_i and letters tested against XA_{i-1} (in which case we additionally assume that A_{i-1} is short). We have already accounted the tests made against A_{i-1} and A_i and by Lemma 8.27 for each occurrence of X there is at most one explicit word whose letters are tested against this occurrence of X. Those that were charged to A_i come from short words, so there are additionally at most $2K$ tests of this form.

So in total A_i is charged only $\mathcal{O}(|A_i|)$ in a phase. From Lemma 8.16 the sum of lengths of long words drops by a constant factor in each phase, and as in the input it is at most n, the total sum of number of protected tests is $\mathcal{O}(n)$. □

Misaligned tests

On the high level, in this section we want to show that if there is a misaligned test then the tested solution is small and use this fact for accounting the cost of such tests. However, this statement is trivial, as we test only solutions of the form a^k for some k, which are always small. To make this statement more meaningful, we generalise the notion of a misaligned test for arbitrary substitutions, not only the tested one. In this way two explicit words A_i and B_j can be misaligned for a substitution s. We show three properties of this notion:

M1 If there is a misaligned test for a substitution s for a letter from A_i against letter in XB_j or a letter from B_j against letter from XA_i then A_i and B_j are misaligned for s. This is shown in Lemma 8.32.

M2 If there are misaligned words A_i and B_j for a solution s then s is small, as shown in Lemma 8.33.

M3 If A_i and B_j are misaligned for s in a phase ℓ then s is reported in phase ℓ or the corresponding words A'_i and B'_j in phase $\ell + 1$ are also misaligned for the corresponding s', see Lemma 8.34.

Those properties are enough to improve the testing procedure so that one $(\mathcal{A}, i)$-word (or $(\mathcal{B}, j)$-word) takes part in only $\mathcal{O}(1)$ misaligned tests: suppose that A_i becomes small in phase ℓ. Then all solutions, for which it is misaligned with some B_j, are small by (M2). Hence, by Theorem **??**, all of those solutions are reported (in particular: tested) within the next c phases, for some constant c. Thus, if A_i takes part in a misaligned test (for s) in phase $\ell' > \ell + c$ then s is not a solution: by (M1) A_i and appropriate B_j are misaligned and by (M3) they were misaligned also in phase ℓ (for the corresponding solution s'), and solution s' was reported before phase ℓ', by (M2). Hence we can immediately terminate the test; therefore A_i can take part in misaligned tests in phases ℓ, $\ell + 1$, ..., $\ell + c$, i.e. $\mathcal{O}(1)$ ones. This plan is elaborated in this section, in particular, some technical details (omitted in the above description) are given.

We say that A_i and B_j that are blocks from two sides of one equations $\mathcal{A}_\ell = \mathcal{B}_\ell$ are *misaligned for a substitution s* if

- a mismatch for s is not found till the end of A_{i+1} or B_{j+1};
- all A_{i+1}, A_i, B_{j+1} and B_j are short;

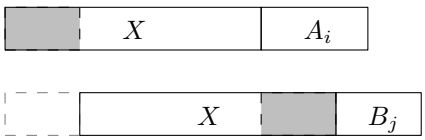

Figure 8.5: A letter from B_j is arranged against the letter from A_i. The period of $s(X)$ is in grey.

- either $A_i \neq A_{i+1}$ or $B_j \neq B_{j+1}$;
- it does not hold that $A_i = B_j$ and $A_{i+1} = B_{j+1}$ and the first letter of A_i is at the same position as the first letter of B_j under substitution s;
- the position of the first letter of A_i in $s(\mathcal{A}_\ell)$ is among the position of $s(XB_j)$ in $s(\mathcal{B}_\ell)$ or, symmetrically, the position of the first letter of B_j in $s(\mathcal{B}_\ell)$ is among the position of $s(XA_i)$ in $s(\mathcal{A}_\ell)$.

We show (M1), which shows that the definitions of misaligned blocks and misaligned tests are reformulations of each other.

Lemma 8.32. *If a letter from A_i is tested (for s) against a letter from XB_j and this test is misaligned then A_i and B_j are misaligned for s; similar statement holds for letters from B_j.*

Proof. This is just a reformulation of a definition (we consider only the case of letters from A_i, the argument for letters from B_j is symmetrical):

- Since this is not a failed test, there is no mismatch till the end of A_{i+1} and B_{j+1}.
- As this is not a protected test, all A_i, A_{i+1}, B_j and B_{j+1} are short.
- As this is a misaligned test, either $A_i \neq A_{i+1}$ or $B_j \neq B_{j+1}$.
- As this is not an aligned test, either $A_i \neq B_j$ or $A_{i+1} \neq B_{j+1}$ or the first letter of A_i is not at the same position as the first letter of B_j (both under s).
- By the choice of B_j, the first position of A_i under s is among the positions of XB_j (under s). □

We move to showing (M2). It follows by considering $s(XA_iXA_{i+1}X)$ and $s(XB_jXB_{j+1}X)$. The large amount of $s(X)$ in it allows showing the periodicity of fragments of $s(X)$ and in the end, that s is small.

Lemma 8.33. *When the A_i and B_j are misaligned for a solution s then s is small.*

Proof. Suppose that A_i and B_j are from an equation $\mathcal{A}_\ell = \mathcal{B}_\ell$. In the proof we consider only one of the symmetric cases, in which A_i is begins not later than B_j (i.e. the first letter of A_i is arranged against the letter from XB_j), the other case is shown similalry.

There are two main cases: either some of A_i, A_{i+1}, B_j and B_{j+1} has some of its letters arranged against an explicit word or all those words are arranged against (some occurrences) of X.

One of the words has some of its letters arranged against an explicit word.

We claim that in this case s has a period of length at most K, in particular, it is small. First of all observe that it is not possible that *each* of A_i, A_{i+1}, B_j and B_{j+1} has *all* of its letters arranged against letters of an explicit word: since A_i is arranged against XB_j this would imply that A_i is arranged against B_j (in particular, their first letters are at corresponding positions) and (as no mismatch is found till end of A_i and B_j) so $A_i = B_j$. Similarly, $A_{i+1} = B_{j+1}$. This contradicts the assumption that A_i and B_j are misaligned.

Thus, there is a word among A_i, A_{i+1}, B_j and B_{j+1}, say B_j, that is partially arranged against an explicit word and partially against X (note that this explicit word does not have to be among A_i, A_{i+1}, B_j and B_{j+1}), see Figure 8.5. As each explicit words is proceeded and succeeded by X, it follows that $s(X)$ has a period at most K.

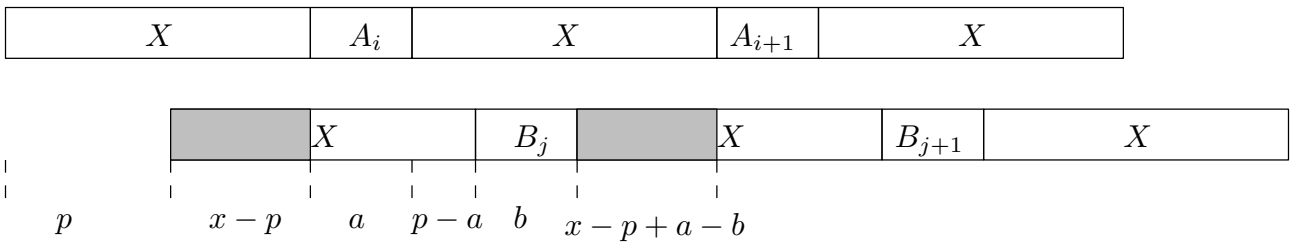

Figure 8.6: The letters of A_i, A_{i+1}, B_j and B_{j+1} are arranged against the letters from $s(X)$. The lengths of fragments of text are beneath the figure, between dashed lines. Comparing the positions of the first and second $s(X)$ yields that p is a period of $s(X)$, second and third that $x-p+a$ while the third and fourth that $p-a+b$ is. The borders of $s(X)$ corresponding to the first and third one are marked in grey.

All words have all their letters arranged against occurrences of X.

In the following we assume that letters from A_i, A_{i+1}, B_j and B_{j+1} are arranged against the letters from $s(X)$. Observe that due to Lemma 8.27 this means that whole A_i is arranged against $s(X)$ preceding B_j, the B_j against $s(X)$ preceding A_{i+1}, whole A_{i+1} against $s(X)$ preceding B_{j+1} and whole B_{j+1} against $s(X)$ succeeding A_{i+1}, see Fig. 8.6.

Let $a = |A_i|$, $b = |B_j|$ and $x = |s(X)|$, as in Fig. 8.6. There are three cases: $a > b$, $a < b$ and $a = b$, we consider them separately.

Consider first the case in which $a > b$, see Fig. 8.6. Let p denote the offset between the $s(X)$ preceding A_i and the one preceding B_j; then $s(X)$ has a period p. Similarly, when we consider the $s(X)$ succeeding A_i and the one succeeding B_j we obtain that the offset between them is $p-a+b$, which is also a period of $s(X)$. Those offsets correspond to borders (of $s(X)$) of lengths $x-p$ and $x-p+a-b$, see Fig. 8.6. Then the shorter border (of length $x-p$) is also a border of the longer one (of length $x-p+a-b$), hence the border of length $x-p+a-b$ has a period $a-b$, so it is of the form $w^k u$, where $|w| = a-b$ and $|u| < a-b$. Now, the prefix of $s(X)$ of length $x-p+a$ is of the form $w^k u'$, for some u' of length less than a (as this is a prefix of length $x-p+a-b$ extended by the following b letters). When we compare the positions of $s(X)$ preceding B_j and the one succeeding A_i we obtain that $s(X)$ has a period $x-p+a$ so the whole $s(X)$ is of the form $(w^k u')^\ell w'$, where w' is a prefix of $w^k u'$, hence s is small: w and u' are of length at most K, as w' is a prefix of $w^k u'$, either it is a prefix of w^k, so it is of the form $w^{k'} w''$ where w'' is a prefix of w, or it includes the whole w^k, so it is of the form $w^k u''$, where u'' is a prefix of u'.

Consider the symmetric case, in which $b > a$ and again use Fig. 8.6. The same argument as before shows that p and $p-a+b$ are periods of $s(X)$ and the corresponding borders are of length $x-p$ and $x-p+a-b$. Now, the shorter of them (of length $x-p+a-b$) is a border of longer of them (of length $x-p$), so the prefix of length $x-p$ of $s(X)$ has a period $b-a$, so it is of the form $w^k u$, where $|w| = b-a$ and $|u| < b-a$. Hence the prefix of length $x-p+a$ is of the form $w^k u'$ for some u' of length less than b. As in the previous case, $s(X)$ has a period $x-p+a$ and so the whole $s(X)$ is of the form $(w^k u')^\ell w'$, where w' is a prefix of $w^k u'$, hence s is small.

Consider now the last case, in which $|A_i| = |B_j|$. If $|A_{i+1}| \neq |A_i|$ then $|B_j| \neq |A_{i+1}|$ and we can repeat the same argument as above, with B_j and A_{i+1} taking the roles of A_i and B_j, which shows that s is small. So consider the case in which $|A_{i+1}| = |A_i|$. If $|B_j| \neq |B_{j+1}|$ then again, repeating the argument as above for A_{i+1} and B_{j+1} yields that s is small. So we are left with the case in which $|A_{i+1}| = |A_i| = |B_j| = |B_{j+1}|$. Then A_{i+1} is arranged against the same letters in $s(X)$ as A_i and B_{j+1} is arranged against the same letters in $s(X)$ as B_j. As there is no mismatch till the end of A_{i+1} and B_{j+1}, we conclude that $A_{i+1} = A_i$ and $B_{j+1} = B_j$ contradicting the assumption that A_i and B_j are misaligned, so this case is non-existing. □

We now show that if A_i and B_j are misaligned for s then they were (for a corresponding solution) in the previous phase (assuming that all involved words were short). This is an easy consequence of

the way explicit words are modified (we prepend and append the same letters and compress all explicit words in the same way).

Lemma 8.34. *Suppose that A_i and B_j are misaligned for a solution s. If at the previous phase all A'_{i+1}, A'_i, B'_{j+1} and B'_j were short then A'_i and B'_j were misaligned for the corresponding solution s'.*

Proof. We verify the conditions on misaligned words point by point:

- Since s' is a solution, there is no mismatch.
- By the assumption, all A'_{i+1}, A'_i, B'_{j+1} and B'_j are short.
- We know that either $A_i \neq A_{i+1}$ or $B_j \neq B_{j+1}$ and so by Lemma 8.21 either $A'_i \neq A'_{i+1}$ or $B'_j \neq B'_{j+1}$ (observe that none of them is the last nor first, as they are not in the next phase).
- Suppose that $A'_i = B'_j$, $A'_{i+1} = B'_{j+1}$ and under s' the first letters of A'_i and B'_j are arranged against each other. By Lemma 8.21 it follows that $A_i = B_j$, $A_{i+1} = B_{j+1}$. Observe that left-popping and right popping preserves the fact that the first letters of $(\mathcal{A}, i)$-word and $(\mathcal{B}, j)$-word are arranged against each other for s' (as $s(\mathcal{A})$ and $s'(\mathcal{A}')$ are the same words). As s' is a solution, the same applies to pair compression and block compression. Hence, the first letters of A_i and B_j are arranged against each other, contradiction with the assumption that A_i and B_j are misaligned.
- Suppose that the first letter of A_i is arranged against a letter from $s(XB_j)$. Consider, how A'_i and XB'_j under s' are transformed to A_i and XB_j under s. As in the above item, popping letters does not influence whether the first letter of $(\mathcal{A}, i)$-word is arranged against letter from $s(X)$ and $(\mathcal{B}, j)$-word (as $s(\mathcal{A})$ and $s'(\mathcal{A}')$ are the same words). Since s' is a solution, the same applies also to pair and block compression. So the position of the first letter of A_i is among the position of $s(XB_j)$ if and only if the first letter of A'_i is arranged against a letter from $s'(XB'_j)$.

  The case in which the position of the first letter of B_j is among the position of $s(XA_i)$ is shown in a symmetrical way. □

Now we are ready to give the improved procedure for testing and estimate the number of the misaligned tests in it.

Lemma 8.35. *There are $\mathcal{O}(n)$ misaligned tests during the whole run of* OneVarWordEq.

Proof. Consider a tested solution s and a misaligned test for a letter from A_i against a letter from XB_j (the case of test of letters from B_j tested against XA_i the argument is the same). Let ℓ be the number of the first phase, in which all $(\mathcal{A}, i)$-word, $(\mathcal{A}, i+1)$-word, $(\mathcal{B}, j)$-word and $(\mathcal{B}, j+1)$-word are short. We claim that this misaligned test happens between ℓ-th and $\ell + c$ phase, where c is the $\mathcal{O}(1)$ constant from Theorem **??**.

Let A'_i and B'_j be the corresponding words in the phase ℓ. Using induction on Lemma 8.34 it follows that A'_i and B'_j are misaligned for s'. Thus by Lemma 8.33 the s' is small and thus by Theorem **??** it is reported till phase $\ell + c$. So it can be tested only between phases ℓ and $\ell + c$, as claimed.

This allows an improvement to the testing algorithm: whenever (say in phase ℓ) a letter from A_i has a misaligned test against a letter from $s(XB_j)$ we can check (in $\mathcal{O}(1)$ time), in which turn ℓ' the last among $(\mathcal{A}, i)$-word, $(\mathcal{A}, i+1)$-word, $(\mathcal{B}, j)$-word and $(\mathcal{B}, j+1)$-word became small (it is enough to store for each explicit word the number of phase in which it became small). If $\ell' + c < \ell$ then we can terminate the test, as we know already that s is not a solution. Otherwise, we continue.

Concerning the estimation of the cost of the misaligned tests (in the setting as above), there are two cases:

- *The misaligned tests that lead to the rejection of s:* This can happen once per tested solution and there are $\mathcal{O}(\log n)$ tested solution in total ($\mathcal{O}(1)$ per phase and there are $\mathcal{O}(\log n)$ phases).

- *Other misaligned tests:* The cost of the test (of a letter from A_i tested against $s(XB_j)$) is charged to the last one among $(\mathcal{A}, i)$-word, $(\mathcal{A}, i+1)$-word, $(\mathcal{B}, j)$-word and $(\mathcal{B}, j+1)$-word that became short. By the argument above, this means that this word became short within the last c phases.

  Let us calculate, for a fixed $(\mathcal{A}, i)$-word (the argument for $(\mathcal{B}, j)$-word is symmetrical) how many misaligned tests of this kind can be charged to this word. They can be charged only within c phases after this word become short. In a fixed phase we test only a constant (i.e. 5) substitutions. For a fixed substitution, A_i can be charged the cost of tests in which letters from A_i or A_{i-1} are involved (providing that A_i or A_{i-1} is short), which is at most $2K$. They can be charged also the tests from letters from B_j that is aligned against X preceding A_{i-1} or X preceding A_i (providing that B_j as well as A_{i-1} are short). Note that there is only one B_j whose letter are aligned against X preceding A_{i-1} and one for X preceding A_i, see Lemma 8.27, so when they are short this gives additional $2K$ tests.

  This yields that one $(\mathcal{A}, i)$-word is charged $\mathcal{O}(K) = \mathcal{O}(1)$ tests in total. Summing over all words in the instance yields the claim of the lemma. □

Aligned tests

Suppose that we make an aligned test, without loss of generality consider the first such test in a sequence of aligned tests. Let it be between the first letter of A_i and the first letter in B_j (both of those words are short). For this A_i and B_j we want to perform the whole sequence of successive aligned tests at once, which corresponds of jumping to A_{i+k} and B_{j+k} within the same equation such that

- $A_{i+\ell} = B_{i+\ell}$ for $0 \leq \ell < k$ and
- $A_{i+k} \neq B_{j+k}$ or one of them is a last word or $A_{i+k}X$ or $B_{j+k}X$ ends one side of the equation.

Note that this corresponds to a syntactical equality of fragments of the equation, which, by Lemma 8.21, is equivalent to a syntactical equality of fragments of the original equation. We preprocess (in $\mathcal{O}(n)$ time) the input equation (building a suffix array equipped with a structure answering general lcp queries) so that in $\mathcal{O}(1)$ we can return such k as well as the links to A_{i+k} and B_{j+k}. In this way we perform all equality tests for $A_iXA_{i+1}X \dots A_{i+k-1}X = B_jXB_{j+1}X \dots XB_{j+k-1}X$ in $\mathcal{O}(1)$ time.

To simplify the considerations, when A_iX (B_jX) ends one side of the equation, we say that this A_i (B_j, respectively) is *almost last* word. Observe that in a given equation exactly one side has a last word and one an almost last word.

Lemma 8.36. *In $\mathcal{O}(n)$ we can build a data structure which given equal A_i and B_j in $\mathcal{O}(1)$ time returns the smallest $k \geq 1$ and links to A_{i+k} and B_{j+k} such that $A_{i+k} \neq B_{j+k}$ or one of A_{i+k}, B_{j+k} is a last word or one of A_{i+k}, B_{j+k} is an almost last word.*

Note that it might be that some of the equal words $A_{i+\ell} = B_{i+\ell}$ are long, and so their tests should be protected (also, the tests for some neighbouring words). So in this way we also make some free protected tests, but this is not a problem. Furthermore, the returned A_{i+k} and B_{j+k} are guaranteed to be in the same equation.

Proof. First of all observe that for A_i and B_j it is easy to find the last word in their equation as well as the almost last word of the equation: when we begin to read a particular equation, we have the link to both the last word and the almost last word of this equation and we can keep them during the testing of this equation. We also know the numbers of those words so we can also calculate the respective candidate for k. So it is left to calculate the minimal k such that $A_{i+k} \neq A_{j+k}$.

Let A'_i, B'_j etc. denote the corresponding original words of the input equation. Observe that by Lemma 8.21 it holds that $A'_{i+\ell} = B'_{j+\ell}$ if and only if $A_{i+\ell} = B_{j+\ell}$ as long as none of them is last or first word. Hence, it is enough to be able to answer such queries for the input equation: if the returned word is in another equation then we should return the last or almost last word instead.

To this end we build a suffix array [24] for the input equation, i.e. for a string $A'_1XA'_2X \dots A'_{n_X}XB'_1XB'_2X \dots B'_{n_X}\$$. Now, the lcp query for suffixes $A'_i \dots \$$ and $B'_j \dots \$$ returns the length of the

longest common prefix. We want to know what is the number of explicit words in the common prefix, which corresponds to the number of Xs in this common prefix. This information can be easily preprocessed and stored in the suffix array: for each position ℓ in $A'_1XA'_2X\dots A'_{n_X}XB'_1XB'_2X\dots B'_{n_X}\$$ we store, how many Xs are before it in the string and store this in the table *prefX*. Then when for a suffixes beginning at positions p and p' we get that their common prefix is of length ℓ, the $\mathit{prefX}[p+\ell]-\mathit{prefX}[p]$ is the number of Xs in the common prefix in such a case. If none of A_i, A_{i+1}, $\dots$, A_{i+k} nor B_j, B_{j+1}, $\dots$, B_{j+k} is the last word nor it ends the equation (i.e. they are all still in one equation) by Lemma 8.21 the k is the answer to our query (as $A_i = B_j$, $A_{i+1} = B_{j+1}$,... and $A_{i+k} \neq B_{j+k}$ and none of them is a last word, nor none of them ends the equation). To get the actual links to those words, at the beginning of the computation we make a table, which for each i returns the pointer to $(\mathcal{A}, i)$-word and $(\mathcal{B}, i)$-word. As we know i, j and k we can obtain the appropriate links in $\mathcal{O}(1)$ time. So it is left to compare the value of k with the value calculated for the last word and almost last word and choose the one with smaller k and the corresponding pointers. □

Using this data structure we perform the aligned tests is in the following way: whenever we make an aligned test (for the first letter of A_i and the first letter of B_j), we use this structure, obtain k and jump to the test of the first letter of A_{i+k} with the first letter of B_{j+k} and we proceed with testing from this place on. Concerning the cost, by easy case analysis it can be shown that the test right before the first of sequence of aligned tests (so the test for the last letters of A_{i-1} and B_{j-1}) is either protected or misaligned. There are only $\mathcal{O}(n)$ such tests (over the whole run of OneVarWordEq), so the time spend on aligned tests is $\mathcal{O}(n)$ as well.

Lemma 8.37. *The total cost aligned test as well as the usage of the needed data structure is $\mathcal{O}(n)$.*

Proof. We formalise the discussion above. In $\mathcal{O}(1)$ we get to know that this is an aligned test, see Lemma 8.29. Then in $\mathcal{O}(1)$, see Lemma 8.36, we get the smallest k such that $A_{i+k} \neq B_{j+k}$ or one of them is an almost last word for this equation or the last word for this equation. We then jump straight to the test for the first letter of A_{i+k} and B_{j+k}.

Consider A_{i-1} and B_{j-1} we show that the test for their last letters (so the test immediately before the first aligned one) is protected or misaligned. By Lemma 8.29 it is enough to show that it is not aligned, nor periodic, nor failed.

- If it were failed then also the test for the first letters of A_i and B_j would be failed.
- It cannot be aligned, as we chose A_i and B_j as the first in a series of aligned tests.
- If it were periodic, then $A_{i-1} = A_i$ and $B_{j-1} = B_j$ while by assumption $A_i = B_j$, which implies that this test is in fact aligned, which was already excluded.

Hence we can associate the $\mathcal{O}(1)$ cost of whole sequence of aligned test to the previous test, which is misaligned or protected. Clearly, one misaligned or protected test can be charged with only one sequence of aligned tests (as it is the immediate previous test). By Lemma 8.31 and 8.35 in total there are $\mathcal{O}(n)$ misaligned and protected tests. Thus in total all misaligned tests take $\mathcal{O}(n)$ time. □

Periodical tests

The general approach in case of periodical tests is similar as for the aligned tests: we would like to perform all consecutive periodical tests in $\mathcal{O}(K)$ time and show that the test right before this sequence of periodic tests is either protected or misaligned. As in case of aligned tests, the crucial part is the identification of a sequence of consecutive periodical tests. To identify them quickly, we keep for each short A_i the value k such that A_{i+k} is the first word that is different from A_i or is the last word or the almost last word (in the sense as in the previous section: A_{i+k} is almost last if $A_{i+k}X$ ends the side of the equation), as well as the link to this A_{i+k}. Those are easy to calculate at the beginning of each phase. Now when we perform a periodical test for a letter from A_i, we test letters from $s((AX)^k)$ against the letters from (suffix of) $s(X(BX)^\ell)$. If $|A| = |B|$ then both strings are periodic with period $|s(AX)|$ and their equality can be tested in $\mathcal{O}(|A|)$ time. If $|A| \neq |B|$ then we retrieve the values $k_\mathcal{A}$ and $k_\mathcal{B}$ which tell us what is repetition of AX and BX. If one of them is smaller than 3 we make

the test naively, in time $\mathcal{O}(|A| + |B|)$. If not, we exploit the fact that $s((BX)^\ell)$ has a period $|s(BX)|$ while $s((AX)^k)$ has a period $|s(AX)|$ and so their common fragment (if they are indeed equal) has a period $||s(AX)| - |s(BX)|| = ||A| - |B||$ (note that the outer '|' denote the absolute value). Hence we check, whether $s(AX)$ and $s(BX)$ have this period and check the common fragment of this length, which can be done in $\mathcal{O}(|A| + |B|)$ time. The converse implication holds as well: if $s(AX)$ and $s(BX)$ have period $||A| - |B||$ and the first $||A| - |B||$ tests are successful then all of them are. Concerning the overall running time, as in the case of aligned test, the test right before the first periodic test is either protected or misaligned, so as in the previous section it can be shown that the time spent on periodical tests is $\mathcal{O}(n)$ during the whole OneVarWordEq.

Lemma 8.38. *Performing all periodical tests and the required preprocessing takes in total $\mathcal{O}(n)$ time.*

Proof. Similarly as in the case of aligned tests, see Lemma 8.37, we can easily keep the value k and the link to A_{i+k} such that A_{i+k} is the last or almost last word in this equation, the same applies for B_{j+k}. Hence it is left to show how to calculate for each short A_i (and B_j) the k such that A_{i+k} is the first word that is different from A_i.

At the end of the phase we list all words A_i that become short in this phase, see Lemma 8.21, ordered from the left to the right (this is done anyway, when we identify the new short words). Note that this takes at most the time proportional to the length of all long words from the beginning of the phase, so $\mathcal{O}(n)$ in total. Consider any A_i on this list (the argument for B_j is identical), note that

- if $A_{i+1} \neq A_i$ then A_i should store $k = 1$ and a pointer to this A_{i+1};
- if $A_i = A_{i+1}$ then A_{i+1} also became short in this phase and so it is on the list and consequently A_i should store 1 more than A_{i+1} and the same pointer as A_{i+1}.

So we read the list from the right to the left, let A_i be an element on this list. Using the above condition, we can establish in constant time the value and pointer stored by A_i. This operation is performed once per $(\mathcal{A}, i)$-word, so in total takes $\mathcal{O}(n)$ time.

Consider a periodic test, without loss of generality suppose that a letter from A_i is tested against a letter from XB_j (in particular, A_i begins not later than B_j), let the k_A and k_B be stored by A_i and B_j; as this is a periodical test, both k_A and k_B are greater than 1. Among A_{i+k_A} and B_{j+k_B} consider the one which begins earlier under substitution s: this can be determined in $\mathcal{O}(1)$ by simply comparing the lengths, the length on the A-side of the equation is $k_A(|A_i| + |s(X)|)$ while B-side is $k_B(|B_j| + |s(X)|) + m$, where m is the remainder of $s(X)$ that is compared with A_i. Let k and ℓ be the smallest numbers such that the common part of $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+\ell-1}X)$ contain the common part of $s(A_iX \cdots XA_{i+k_A-1}X)$ and $s(B_jX \cdots XB_{j+k_B-1}X)$

A_{i+k} and $B_{j+\ell}$ be the Note that the test for the first letter of this word is not periodic, so when we jump to it we skip the whole sequence of periodic tests. We show that in $\mathcal{O}(1)$ time we can perform the tests for all letters before this word and that the test right before the first test for A_i is protected or misaligned.

Let $a = |A_i|$, $b = |B_j|$ and $x = |s(X)|$. First consider the simpler case in which $a = b$. Then the tests for A_{i+1}, ..., A_{i+k-1} are identical as for A_i, and so it is enough to perform just the test for A_i and B_j and then jump right to A_{i+k}.

So let us now consider the case in which $a > b$. Observe that when the whole $s((B_jX)^\ell)$ is within $s((A_iX)^3)$ then this can be tested in constant time in a naive way: the length of $s((A_iX)^3)$ is $3(a+x)$ while the length of $s((B_jX)^\ell)$ is $\ell(b+x)$. Hence $3(a+x) \geq \ell(b+x)$ and so $\ell \leq 3(a+x)/(b+x) \leq 3\max(a/b, x/x) \leq 3K$, because a/b is at most K. Thus all tests for $s((A_iX)^3)$ and $s((B_jX)^\ell)$ can be done in $\mathcal{O}(K) = \mathcal{O}(1)$ time.

So consider the remaining case, see Fig. 8.7 for an illustration, when $k > 3$. We claim that the tests for $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+\ell-1}X)$ are successful if and only if

- $s(A_iX)$ and $s(B_jX)$ have period $\gcd(a+x, b+x)$ *and*
- the first $\gcd(a+x, b+x)$ tests for $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+\ell-1}X)$ are successful.

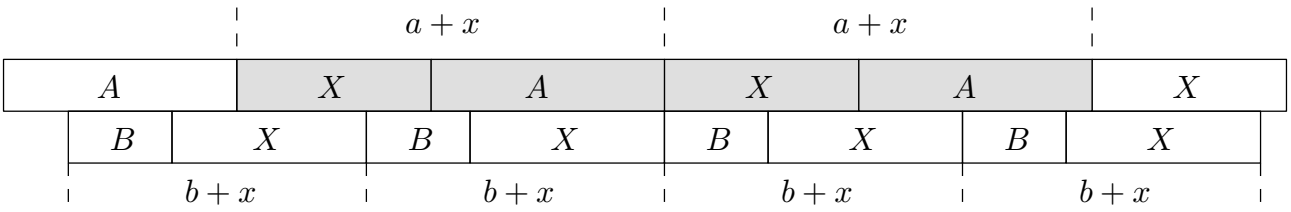

Figure 8.7: The case of $a > b$. The part of $s((XA_i)^2)$ that has a period $a + x$ and $b + x$ is in grey.

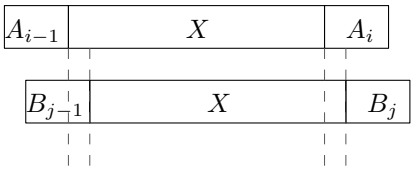

Figure 8.8: The test right before the first among the sequence of periodic tests. Since A_i begins not later than B_j, B_{j-1} ends not earlier than A_{i-1}.

$\Rightarrow$ First $s(XA_iXA_i)$ has period $x + a$. However, it is covered with $s((B_jX)^\ell)$, so it also has period $x + b$. Since $x + a + x + b < 2x + 2a$, it follows that also the $\gcd(x + a, x + b)$ is a period of $s(XA_iXA_i)$ and so also of $s(A_iX)$ and thus also $s(B_jX)$. The second item is obvious.
$\Leftarrow$ Since $s(A_iX)$ and $s(B_jX)$ have period $\gcd(a+x, b+x)$ also $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+\ell-1}X)$ have this period. As the first $\gcd(a+x, b+x)$ tests for $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+k-1}X)$ are successful, it follows that all the tests for their common part are.

So, to perform the test for $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+\ell-1}X)$ it is enough to: calculate $p = \gcd(a + x, b + x)$, test whether $s(A_iX)$, $s(B_jX)$ have period p and then perform the first p tests for $s(A_iX \cdots XA_{i+k-1}X)$ and $s(B_jX \cdots XB_{j+\ell-1}X)$. All of this can be done in $\mathcal{O}(1)$, since $p \leq a - b \leq K$ (note also that calculating p can be done in $\mathcal{O}(1)$, as $\gcd(x + a, x + b) = \gcd(a - b, x + b)$ and $a - b \leq K$).

The case with $b > a$ is similar: in the special subcase we consider whether $s((A_iX)^k)$ is within $s(X(B_jX)^3)$. If so then the tests can be done in $\mathcal{O}(K)$ time. If not, then we observe that the $s(XB_{j+1}XB_{j+2})$ is covered by $s((A_iX)^k)$. So it the tests are successful, it has period both $x + b$ as well as $x + a$, so it has period $\gcd(x + a, x + b)$. The rest of the argument is identical.

For the accounting, we would like to show that the test right before the first among the considered periodic tests is not periodic. Observe, that as A_i begins not later (under s) than B_j it means that the last letter of B_{j-1} is not earlier than the last letter of A_{i-1}, see Figure 8.8. So the previous test includes the last letter of B_{j-1}. It is enough to show that this test is not failed, periodic, nor aligned.

- *failed:* If it is failed then also the test for the letters in A_i are failed.
- *periodic:* If it is periodic then this contradicts our choice that the test for the first letter of A_i is the first in the sequence periodic tests.
- *aligned:* Since the first letter of A_i is arranged against XB_j, in this case the last letter of B_{j-1} needs to be arranged against the last letter of A_{i-1}. Then by the definition of the aligned test, $B_j = A_i$ and their first letters are at the same position. As by the assumption about the periodic tests we know that $A_{i+1} = A_i$ and $B_{j+1} = B_j$ we conclude that the test for the first letter of A_i is in fact aligned, contradiction.

Hence, by Lemma 8.29, the test for the last letter of B_{j-1} is either protected or misaligned. Using the same accounting as in Lemma 8.37 we conclude that we spent at most $\mathcal{O}(n)$ time on all periodic tests. □

Proof of Lemma 8.28

It is left to show that indeed we do not need to take into the account the time spent on comparing $s(X)$ with $s(X)$ on the other side of the equation.

proof of Lemma 8.28. Recall that we only test solutions of the form $s(X) = a^k$. Since we make the comparisons from left to the right in both $s(\mathcal{A}_\ell)$ and $s(\mathcal{B}_\ell)$ then when we begin comparing letters from one $s(X)$ with the other $s(X)$, we in fact compare some suffix a^ℓ of a^k with a^k. Then we can skip those a^ℓ letters in $\mathcal{O}(1)$ time. Consider the previous test, which needs to include at least one explicit letter. Whatever type of test it was or whatever group of tests it was in, some operations were performed and this took $\Omega(1)$ time. So we associate the cost of comparing $s(X)$ with $s(X)$ to the previous test, increasing the running time by at most a multiplicative constant. □

Exercises

Task 43 Consider an equation

$$A_0XA_1 \dots A_{n_X-1}XA_{n_X} = XB_1 \dots B_{n_X-1}X$$

with $A_0 \neq \epsilon \neq A_{n_X}$. Show that it has an equivalent equation in which $A'_{n_X} = \epsilon$ and $B'_{n_X} \neq \epsilon$.

Hint: First show that there is a system of equivalent equations, obtained by appropriate splitting, and then concatenate them in a different way.

Task 44 Show that if a word equation $Xp = qX$ is satisfiable then:

- p, q are conjugate and consequently also the primitive roots of p, q are conjugate, that is, there are u, v such that uv, vu are primitive and $p = (vu)^k$ and $q = (uv)^k$ for some $k \geq 1$;
- the set of solutions is $(uv)^*u$.

Given p, q the u, v can be calculated in linear time.

Task 45 Show that for a given a set of primitive words $P_1, \dots, P_k$ such that for each i $P_i^2 \sqsubseteq B_1A_0A_0$, in total time $\mathcal{O}(|B_1A_0A_0|)$ we can establish for all P_i from $P_1, \dots, P_k$ the P_i-prefix of $B_1A_0A_0$.

Task 46 Show that if a word equation

$$A_0XA_1 \dots A_{n_X-1}X = XB_1 \dots B_{n_X-1}XB_{n_X}$$

has an infinite family of solutions $s_i(X) = (uv)^iu$, then uv is the primitive root of A_0 (this more or less follows from the two first cases of our analysis) and, by symmetry, vu is the primitive root of B_{n_X}.

Moreover, this equation is equivalent to a system of equations (as above), in which each equation additionally satisfies:

$$\begin{aligned} A_0 &\in (uv)^+ \\ A_i, B_j &\in v(uv)^* \text{for } i > 0, j < n_X \\ B_{n_X} &\in (vu)^+ \end{aligned}$$

To be more precise, we can partition the above equation into such a system.

To this end consider the uv-prefixes of the sides. What happens, when they terminate?

Task 47 Show that if a word equation

$$A_0XA_1 \dots A_{n_X-1}X = XB_1 \dots B_{n_X-1}XB_{n_X}$$

has an infinite family of solutions $s_i(X) = (uv)^iu$ (for each i) then it has no other solution.

Use Task 46 and Task 44 and think big:

- use simplification from Task 46

- under those assumptions show that $XA_1X \dots A_{n_X-1}X, XB_1 \dots B_{n_X-1}X \in (uv)^*u$: treat the equation as an equation from Task 44
- conclude that without loss of generality $A_1, \dots, A_{n_X-1}, B_1, dots, B_{n_X-1} = v$
- then $A_1X \dots A_{n_X-1}X$ has period uv and Xv
- conclude that $X \in (uv)^*u$
- show that

. A proof by case inspection and periodicity is most likely very difficult. Laine and Plandowski [28] has such a proof, but it seems to have an error.

Task 48 Suppose that given a word w we can construct in $\mathcal{O}(n)$ time a structure such that given two indices i, j in $\mathcal{O}(1)$ time returns the length of the longest common prefix of words $w[i \,..\, n]$ and $w[j \,..\, n]$.

Explain how it can be used to verify in $\mathcal{O}(n + n_X \log n)$ time the $\mathcal{O}(\log n)$ candidate solutions, each of which is a prefix of A_0, defined as in (8.1).

Task 49 Show that 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 B_{n_X}.

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.

Task 50 Show that if compression-based algorithm (for word equations) at some point performs a a-blocks compression (and at least one block is replaced), then the word compressed to a from the initial instance is primitive.

For simplicity you may consider the algorithm running on a given word, with no variables.

Task 51 RSLP (run-length SLP) is an SLP in which we allow rules $X \to Y^k$, where k is a natral number. The rule size is considered constant.

Show that a compression-based algorithm for one-variable word equation reports a solution defind using an RSLP of size (and height) $\mathcal{O}(1)$ in $\mathcal{O}(1)$ rounds.

Task 52 Show that a one-variable word equations, whose constant words between the variables are defined using SLPs, can be solved in polynomial time.

This should be easy with compression-based algorithm (also tor RSLPs instead of SLPs), for word-combinatorics based one you need to use some (known) algorithms: pattern matching of SLP-defined pattern inside SLP-defined word can be done ia polynomial time. Computation of all primitive squares prefixing an SLP can be done in polynomial time.

Task 53 Show how to solve (in polynomial time) a 1-variable word equation which uses (inside constant words) expression of a form $(u^iu'v)^j$, where i, j are *parameters*, i.e. we want to have a description of all solutions, for all possible values of i, j. Here u' is a prefix of u.

If this helps, you can assume that $(u^iu'v)^j$ is a prefix of A_0.

Again, should be rather easy using compression-based approach. Requires some not-so difficult claims on primitivity for word-combinatorics based

Chapter 9

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 (i.e. each variable occurs at most twice) equations it is easy to give a (non-deterministic) linear-space algorithm, which preceded a general PSPACE algorithm.

Algorithm 12 Lentin/Plotkin/Siekman/Matiyasevich algorithm for word equations

```
1:  while The equation u = v is nontrivial do
2:      let u = αu', v = βv'
3:      if α = β then
4:          set u ← u', v ← v'
5:      else if α is a variable then
6:          if s(α) = ε then                          ▷ Non-deterministic guess
7:              remove α from the equation
8:          else if s(β) = ε then                     ▷ Non-deterministic guess
9:              remove β from the equation
10:         else if s(α) ≥ s(β) then                  ▷ Non-deterministic guess
11:             replace α in the equation with βα
12:         else                                      ▷ β is a variable
13:             replace β with αβ
14:     else if β is a variable then                  ▷ Symmetric actions
15:     else if α, β are different letters then
16:         reject
```

9.1 Analysis

It is easy to see that Algorithm 12 is sound — this follows straight from Lemma 2.3; it is not difficult to see that it is complete (when we make the choices according to some solution). What is not obvious, and in fact not true in general, is that it is terminating. However, for quadratic word equations the length of the equation does not increase: we introduce 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 [26], but the argument is heavy in terms of geometry, so it will not be presented here.

We say that (quadratic word equations) is *regular*, when each variable occurs at most once at each side. Satisfiability of quadratic regular word equations is in NP [9].

Exercises

Task 54 Run the algorithm for quadratic word equation on

$$abXcY = YcXba$$

Draw the resulting graph of equations considered by the algorithm.

Task 55 Show that the algorithm for quadratic equations in fact yields a description of all solutions of such na equation.

Task 56 Consider a restricted class of word equations satisfying the following two conditions: regular and

oriented If two variables X, Y occur on both sides of the equation then they appear in the same order on both sides (i.e. if X occurs to the left of Y on the left-hand side, the same happens on the right-hand side and vice-versa).

Show that satisfiability quadratic, regular, oriented word equations is in NP.

Task 57 Extend the algorithm for quadratic word equations so that it also allows regular constraints.

Chapter 10

Word equations with two variables

In this section be n we consistently mean the size of the input equation over two variables.

10.1 Parametrised words

This chapter is based on [13].

0 A 0-parametrised word is a language consisting of a single word p of length $\mathcal{O}(n)$; the size of this parametrised word is $|p|$.

1 A 1-parametrised word is a language $\{p^j q \,:\, j \geq 0\}$, where p, q are words, p is not a prefix of q; the size of this parametrised word is $|pq|$.

2 A 2-parametrised word is a language $\{(p^{j+a}q)^k p^j p' \,:\, j, k \geq 0\}$, where p, q, p' are words, $p' \sqsubseteq p$, p is not a prefix of $q$and $a \in \mathbb{N}$; the size of this parametrised word is $|pqp'|$.

10.2 Canonisation

We say that a word equation with two variables is in a *canonised* form if its sides begin with different variables (so one side with X and the other with Y) and end with different variables (so in case of quadratic words).

Lemma 10.1. *Given an equation over two variables it can be transformed into an equation in a canonised form or a superset of solutions for one of the variables which is a union of $\mathcal{O}(n)$ 1-parametrised words or 0-parametrised words can be found. If the equation is transformed then during this transformation variables are substituted with $X \leftarrow u_X X v_X$, $Y \leftarrow u_Y Y v_Y$ such that $u_X v_X u_Y v_Y \in \mathcal{O}(n)$ and among u_X, u_Y one is empty and among v_X, v_Y one is empty.*

Note that both cases can happen: we have a set of substitutions for a variable to test *and* a canonised equation.

Proof. We proceed similarly as in the case of quadratic equations:

- if sides of the equation begin with the same symbol (be it a letter or a variable) then we delete it;
- if the sides of the equation begin with different variable then we are done;
- if one side of the equation begins with X and the other with AY then we return a set of test substitutions for X: $\{A' \,:\, A' \sqsubseteq A\}$ and otherwise execute the substitution $X \leftarrow AX$. After removing the leading A from both sides of the equation we are done at this end of the equation.
- if one side of the equation begins with X and the other with AX then we return the following union of 1-parametrised words of possible substitutions for X: $\{A^j A' \,:\, A' \sqsubseteq A\}$.

- we perform symmetric actions at the end of the equation. If we deleted the equation and no parametrised solutions were proposed then this equation is always satisfied. If some were proposed then we are done. □

10.3 Simple systems of equations and their solutions

We are interested in simple systems that occur during the main reduction steps; those systems are of the following forms.

In the following subsections n denotes the length of the input equation and A, B, C, D are words, such that $|ABCD| \in \mathcal{O}(n)$.

10.3.1 $\mathcal{S}_1$

Assume additionally that CD is a primitive word. Then the system $\mathcal{S}_1$ is defined as

$$\begin{cases} YAX &= XBY \\ CDY &= YDC \end{cases} . \tag{$\mathcal{S}_1$}$$

Lemma 10.2. *Given a system of equations $\mathcal{S}_1$, in time $\mathcal{O}(n^2)$ we can find a superset of substitutions for X in all solutions, which is of a union of*

- *At most one 2-parametric word $\{(p^{j+a}q)^k p^j p' : j, k \geq 0\}$ where p is primitive and $0 \leq a \leq n$.*
- *A set of $\mathcal{O}(n)$ 1-parametric words $\{p^j q : j \geq 0\}$ with p primitive.*
- *At set of $\mathcal{O}(n^2)$ 0-parametric words.*

This representation is also a superset of substitutions for YAX.

10.3.2 $\mathcal{S}_2$

Let $|A| = |B| \leq |C| = |D|$. Then the system $\mathcal{S}_2$ is defined as

$$\begin{cases} YAX &= XBY \\ YCX &= XDY \end{cases} , \tag{$\mathcal{S}_2$}$$

and we are interested only in solutions for which $s(Y) < s(X)$.

Lemma 10.3. *Given a system of equations $\mathcal{S}_2$, in time $\mathcal{O}(n^2)$ we can find a superset of substitutions for X in all solutions with $|s(Y)| < |s(X)|$, which is of a union of*

- *At most one 2-parametric word $\{(p^{j+a}q)^k p^j p' : j, k \geq 0\}$ where p is primitive and $0 \leq a \leq n$.*
- *A set of $\mathcal{O}(n)$ 1-parametric words $\{p^j q : j \geq 0\}$ with p primitive.*
- *At set of $\mathcal{O}(n \log n)$ 0-parametric words.*

This representation is also a superset of substitutions for YAX.

10.3.3 $\mathcal{S}_3$

Let now $A \neq B$. Then the system $\mathcal{S}_3$ is defined as

$$YAX = XBY \ . \tag{$\mathcal{S}_3$}$$

System $\mathcal{S}_3$ is treated as a quadratic equation and the set of all its solutions can be found using this representation.

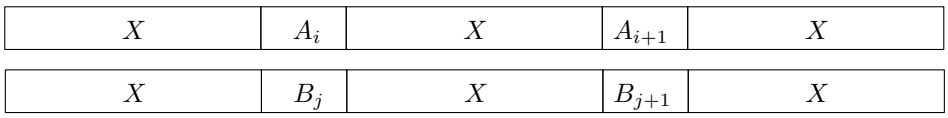

Figure 10.1: First case: each X matches an X.

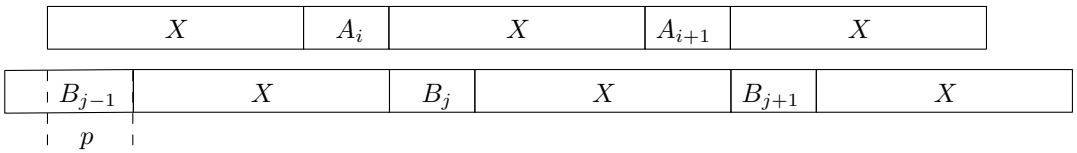

Figure 10.2: Second case: end of X matches B_{j-1}.

10.3.4 $\mathcal{S}_4$

Finally, the system $\mathcal{S}_4$ is defined as

$$YAX = XAY \ . \tag{$\mathcal{S}_4$}$$

Lemma 10.4. *Given a system of equations $\mathcal{S}_4$, in time $\mathcal{O}(n^2)$*

- *find a representation of its solutions of the form $X = (ZP)^j Z, Y = (ZP)^k Z$, where P is a word and Z is a variable.*
- *Find a superset of other solutions of form of $\mathcal{O}(n)$ 1-parametric words*

$$\{p^j q \ : \ j \geq 0\}$$

10.4 Singleton equations

Suppose that (after canonization) the two sides of the equations can be written as

$$XA_1XA_2X\cdots XA_kY\varphi(X,Y)YB_1XB_2X\cdots B_kX\psi(X,Y)$$

such that $|A_1A_2\cdots A_k| = |B_1B_2\cdots B_k|$.

Then $XA_1XA_2X\cdots XA_k$ and $B_1XB_2X\cdots B_kX$ (after substituting x for X) commute. In particular $(XA_1XA_2X\cdots XA_k)^2$ has an occurrence of $B_1XB_2X\cdots B_kX$.

There are some cases.

Every X in $B_1XB_2X\cdots B_kX$ matches an X above, so also each B_i matches some A_j, see Fig. 10.1. Note that there are at most k such possible matches. Then there are $P(X), Q(X)$ such that $P(X)Q(X)$ is primitie (as a word over $\Sigma \cup \{X\}$)

$$\begin{aligned} XA_1XA_2X\cdots XA_k &= (P(X)Q(X))^m \\ B_1XB_2X\cdots B_kX &= (Q(X)P(X))^\ell Y \qquad\qquad = (Q(X)P(X))^q Q(X) \end{aligned}$$

Some end of X is matched to an element of some A_i or B_j (and the other end is not) see Fig. 10.2 Then it has a short period p of at most n letters (candidates to be checked).

Both ends of some X are within some A_i or B_j, see Fig. 10.3/ Then X is short: it has at most n letters (candidates to be checked).

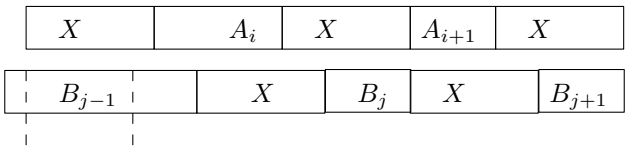

Figure 10.3: Second case: X is a substring of a constant word.

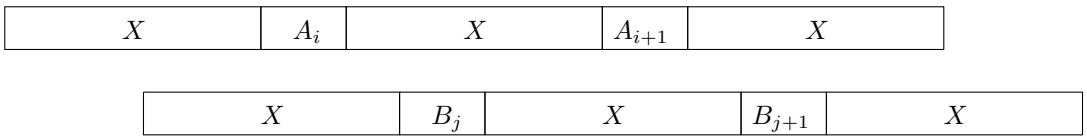

Figure 10.4: First case: each X matches an X.

The main case is that each A_i, B_j is matched into substring of X, see Fig, 10.4. Depending on lengths of A_i and B_j this yields a system $\mathcal{S}_1$ (when some A_i and B_j are of different length) or $\mathcal{S}_2$ (when all words are of the same length) or even $XA = YBX$, when all words in question are equal.

In the general case we can reduce a general system to something of similar form (If there are many B's than A's then we can take a prefix. The case of more As is a bit more complex.)

Exercises

Task 58 Solve a system $\mathcal{S}_2$ of word equations in two unknowns

$$YAX = XBY$$
$$YCX = XDY$$

where $(A, B) \neq (C, D)$. That is, present a simple superset of its solutions.

By symmetry you may assume that $|s(X)| > |s(Y)|$.

Consider $|s(Y)| < |s(X)| \leq |s(Y)||A|$ and $|s(X)| > |s(Y)||A|$ separately. Inb the second case substitute $X = YAZ$.

Task 59 Solve a system $\mathcal{S}_3$ of word equations in two unknowns

$$XAY = YBX$$

To this end consider it as a quadratic equation. What can you tell about its graph?

The solution set is described as morphism applied to some words. We want an exact description, not superset.

Task 60 Solve a system $\mathcal{S}_4$ of word equations in two unknowns:

$$XAYA = YAXA \ .$$

That is, give a reasonable superset of solution.

Chapter 11

One-variable equations in a free group

The problem of word equations in the free group was first investigated by Lyndon [39], who considered the restricted variant of one-variable equations. He showed that the solution set is a finite union of sets of the form

$$\{w_0 w_1^{i_1} w_2 w_3^{i_2} \cdots w_{2k-1}^{i_k} w_{2k} \ : \ i_1, \ldots, i_k \in \mathbb{Z}\} \ , \tag{11.1}$$

where $w_0, \ldots, w_{2k}$ are words over the generators of the free group, we call such sets k-parametric. In fact, it was first shown using combinatorial arguments that a superset of all solutions is of this form, and using algebraic methods the superset of all solutions is transformed into the actual set of all solutions. As a result, k depends on the equation and is a by-product of the algorithm rather than an explicitly given number. By using a more refined, though purely combinatorial, argument Appel [1] showed that there exists a superset of solutions that is a finite union of 1-parametric sets and that one can test for which values such words are indeed solutions. In principle, the proof can be readily used as an algorithm, but no reasonable bounds can be derived from it. Unfortunately, Appel's proof contains an error (see [5] for a discussion). A similar characterization was announced by Lorentz [37], but the proof was not supplied.

A polynomial-time algorithm solving the one-variable word equations (in the free group) was given by Bormotov, Gilman and Myasnikov [3]. In principle, their argument is similar to Appel, though simpler (and without errors), and extra care is taken to guarantee that testing takes polynomial time. The running time is high, though little effort was made to lower the exponent, we believe that simple improvements and better analysis should yield $\mathcal{O}(n^5)$ running time of their algorithm.

No polynomial-time algorithm for two-variable equations (in a free group) is known.

11.1 Preliminaries

To distinguish between the equality in a free group with involution (reversal) and equality in a free group, we will use '$=$' for the former and '$\approx$' for the latter.

Any equation in the free group is equivalent to an equation in which the right-hand side is ε, as $u \approx v$ is equivalent to $uv^{-1} \approx \varepsilon$, thus in the following we consider only equations in such a form. Moreover, $uv \approx \varepsilon$ is equivalent to $vu \approx \varepsilon$, which can be seen by multiplying by v from the left and v^{-1} from the right; hence we can assume that the equation begins with a variable. Let us fix the equation

$$X^{p_1} u_1 X^{p_2} u_2 \cdots u_{m-1} X^{p_m} u_m \approx \varepsilon \tag{11.2}$$

for the rest of the paper, each u_i is a reduced word in Σ^*, every p_i is 1 or -1 and there are no expressions $X\varepsilon\overline{X}$ nor $\overline{X}\varepsilon X$ in the equation. Clearly, m is the number of occurrences of the variable X in the equation, let $n = m + \sum_{i=1}^{m} |u_i|$ be the length of the equation. A reduced word $x \in \Sigma^*$ is a solution when $x^{p_1} u_1 x^{p_2} \cdots u_{m-1} x^{p_m} u_m \approx \varepsilon$.

11.2 Some combinatorial Lemmata

Lemma 11.1. *If $u_1 w u_2 = v_1 \overline{w} v_2$ and w is reduced then either $w = 1$ or $v_1 \overline{w} \sqsubseteq u_1$ or $u_1 w \sqsubseteq v_1$, i.e. w and $\overline{w}$ cannot overlap.*

Proof left as an exercise.

Lemma 11.2. *Let s be a cyclically reduced word. Let W be a set of words and $k = \sum_{w\in W} |w|$. Suppose that $s^{k_1}, \ldots, s^{k_p}$ are pairwise disjoint subwords of words in W and that $k_1, \ldots, k_p$ are pairwise different integers. Then $p \leq \sqrt{4k/|s| + 1}$ and if additionally $k \geq |s|$ then $p \leq \sqrt{5k/|s|}$.*

Proof left as an exercise.

11.2.1 Pseudosolutions

We want to specify some properties of reductions of solutions, instead of usual reduction sequences it is a bit more convenient to talk about pairings that they induce. Given a word $w[1\,..\,n] \in \Sigma^n$, $w \approx \varepsilon$ its partial reduction pairing (or simply partial pairing). is intuitively speaking, a pairing of indices of w corresponding to some reduction. Formally, it is a partial function $f : [1\,..\,n] \to [1\,..\,n]$ such that if $f(i) \neq \perp$ then $f(f(i)) = i$ (it is a pairing), $w[i] = \overline{w[f[i]]}$ (it pairs inverse letters) and either $f(i) \in \{i-1, i+1\}$ or $f(i) = j \neq i$ and f is defined on the whole interval $[\min(i,j)+1, \max(i,j)-1]$ and $f([\min(i,j)+1, \max(i,j)-1]) = [\min(i,j)+1, \max(i,j)-1]$ (so the pairing is well nested and corresponds to a sequence of reductions). A partial pairing is a *pairing* if it is a total function. When needed, we will draw partial reduction pairings as on Fig. 11.1–11.9, i.e. by connecting appropriate intervals of positions. Note that the reduction pairing is not unique, say $a\overline{a}a\overline{a}$ has two different reduction pairings.

It is easy to see that a reduction pairing induces to reduction sequence (perhaps more than one) and vice-versa, and so a word w has a reduction pairing if and only if $w \approx \varepsilon$.

Lemma 11.3. *A word w has a reduction pairing if and only if $w \approx \varepsilon$.*

Proof. We proceed by a simple induction: if w is reducible then either $w = a\overline{a}$ for some $a \in \Sigma$ and then it clearly has a reduction paring or $w = w_1 a\overline{a} w_2$ and $w_1 w_2 \approx \varepsilon$. Create the pairing for w by pairing those a and $\overline{a}$ and otherwise using the pairing for $w_1 w_2$, which is known to exist by the induction assumption (formally some renumbering of the indices is needed).

In the other direction, if w has a reduction pairing, consider $f(1) = i$. Then $w = a w_1 \overline{a} w_2$, such that w_1 and w_2 are paired inside. Thus by induction assumption both $w_1 \approx \varepsilon$ and $w_2 \approx \varepsilon$. Thus $w \approx a\overline{a} \approx \varepsilon$. □

Given a (not necessarily reducible) word $w = w_1 w_2 w_3 \in \Sigma^*$ we say that w_2 is a *pseudo-solution* for a partial reduction pairing f if f is defined on whole w_2. This is sometimes written as $w_1 \underline{w_2} w_3$ to make graphically clear, which subword is a pseudosolution. Note that we do allow that $w_2 \approx \varepsilon$, in which case it is a pseudo-solution, and we do allow that f pairs letters inside w_2.

The first fact to show is that for any pairing f if we factorize a reducible word then there is a pseudo-solution for some of its consecutive subwords. A variant of this Lemma was used Lyndon [39, Proposition 1], Appel [1, Proposition 1] and by Bormotov, Gilman and Myasnikov [3, Lemma 3] and it is attributed already to Nielsen [47].

Lemma 11.4. *Let $\varepsilon \approx s_0 u_1 s_1 u_2 \cdots s_{k-1} u_k s_k$ and f be its pairing. Then there is u_i that is a pseudo-solution of $u_{i-1} s_{i-1} \underline{u_i} s_i u_{i+1}$ (for f).*

Proof. If there is $u_i = \varepsilon$ then we are done. So consider the case that each $u_i \neq \varepsilon$. We maintain an interval of position I such that $f(I) \subseteq I$ and I contains at least one word u_i. Initially $I = [1\,..\,|w|]$.

Take any u_i within I and let $i_{\min}, i_{\max}$ be positions within u_i such that $f(i_{\min}) = \min f(u_i)$ and $f(i_{\max}) = \max f(u_i)$, i.e. $[f(i_{\min}), f(i_{\max})]$ is the smallest interval of positions such that u_i is a pseudosolution within it. If $f(i_{\min}), f(i_{\max}) \in u_{i-1} s_{i-1} u_i s_i u_{i+1}$ then we are done. If not, then by symmetry consider $f(i_{\min}) \notin u_{i-1} s_{i-1} u_i s_i u_{i+1}$. If $f(i_{\min})$ it is to the right of u_{i+1} then we take as the interval $[i_{\min}+1, f(i_{\min})-1]$: clearly it contains whole u_{i+1} and $f([i_{\min}+1, f(i_{\min})-1]) = [i_{\min}+1, f(i_{\min})-1]$ and it is smaller than I. If $f(i_{\min})$ it is to the left of u_{i-1} then we take $[i_{\min}-1, f(i_{\min})+1]$ and analyze it symmetrically. Thus at some point we will find the pseudo-solution. □

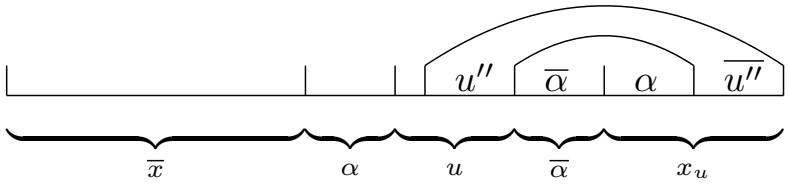

Figure 11.1: Pseudo-solution for the equation $\overline{x}\alpha u\overline{\alpha}\underline{x_u}$. Case when whole x_u is reduced within $\alpha u\overline{\alpha}$.

11.3 Superset of solutions

The previous characterization [3] essentially showed that a solution is either a $\mathcal{O}(1)$-represented word or of the form $u^i u' v'' v^j$ for some $i, j \in \mathbb{Z}$ and $u' \sqsubseteq u, v \sqsupseteq v''$ for some well defined u, v. As we intend to analyze those solutions using word combinatorics, it is useful to assume that u, v are cyclically reduced and primitive. Unfortunately, this cannot be extracted directly from the previous characterization, so we repeat the previous arguments taking some extra care.

Lemma 11.5 (cf. [19, Lemma 15])**.** *For a given equation* (11.2)*, in $\mathcal{O}(n^2)$ time one can compute a superset of solutions of the form*

$$S \cup \bigcup_{(\alpha u^I v^J \beta) \in W} \bigcup_{i,j \in \mathbb{Z}} \{\alpha u^{I(i)} v^{J(j)} \beta\}$$

where S is a set of $\mathcal{O}(1)$-represented words with $|S| = \mathcal{O}(n^2)$ and for each $0 \leq i \leq m-1$ there are numbers $\ell_i, \ell_i' \leq |u_i| + |u_{i+1}|$ such that W contains exactly $\ell_i \cdot \ell_i'$ parametric words satisfying

- *α, β, are $\mathcal{O}(1)$-represented, reduced and $|\alpha|, |\beta| \leq |u_i| + |u_{i+1}|$;*
- *u, v are 2-represented, cyclically reduced, primitive and $|u| = \ell_i$ and $|v| = \ell_i'$.*

The main principle of the proof of Lemma 11.5 is that when x is a solution of an equation (11.2), then after the substitution the obtained word is reducible and thus by Lemma 11.4, one of substituted x or $\overline{x}$ is a pseudosolution in $x^{p_h} u_h x^{p_{h+1}} u_{h+1} x^{p_{h+2}}$, where $p_h, p_{h+1}, p_{h+2} \in \{-1, 1\}$. Thus we analyze each possible triple p_h, p_{h+1}, p_{h+2} and show the possible form of the pseudosolution in corresponding case. Note that by symmetry we can consider $p_{h+1} = 1$.

We begin with some preliminary Lemmata.

Lemma 11.6. *Let $x_u \sqsubseteq x$ be a pseudo-solution of $\overline{x}\alpha u\overline{\alpha}\underline{x_u}$, where $x, \alpha u\overline{\alpha}$ are reduced and u is cyclically reduced. Then*

- *$x_u \sqsubseteq \alpha \overline{u}\, \overline{\alpha}$ or*
- *$x_u \approx \alpha u^i u'$ where $u' \sqsubseteq u$ and $i \in \mathbb{Z}$*

Proof. If $f(x_u) \subseteq \alpha u\overline{\alpha}$, see Fig. 11.1, then x_u is an inverse of some suffix of $\alpha u\overline{\alpha}$, so $x_u \sqsubseteq \alpha\overline{u}\,\overline{\alpha}$, as claimed.

In the remaining case observe that we may assume that $\alpha \sqsubseteq x_u$: consider the reduction pairing f, observe that either $\overline{x}\alpha$ reduces the whole α or $\overline{\alpha}x_u$ the whole $\overline{\alpha}$: if none of this happens then x_u reduces within $\overline{\alpha}x_u$, which was considered. But then in either case $\alpha \sqsubseteq x$ and so $\alpha \sqsubseteq x_u$ or $x_u \sqsubseteq \alpha$, the latter was already considered.

Let $x' \sqsubseteq x$ be the minimal prefix of x such that $\overline{x'}\alpha u\overline{\alpha}x_u \approx \varepsilon$. If $x' \sqsubseteq \alpha$ then again we end in the case such that $f(x_u) \subseteq \alpha u\overline{\alpha}$. So $\alpha \sqsubseteq x'$. As $\overline{x'}\alpha u\overline{\alpha}x_u \approx \varepsilon$ we can modify the pairing by first pairing the suffix $\overline{\alpha}$ of x' with α and the $\overline{\alpha}$ with the prefix α of x_u, see Fig. 11.2, and then extend to the rest of $\overline{x'}\alpha u\overline{\alpha}x_u$. Note that x is still a pseudosolution for such a modified pairing.

Observe now that it cannot be that letters in u are paired with both $\overline{x}$ and x_u, as this would imply that the first and last letter of u are paired with a corresponding letter of $\overline{x}$ and x, respectively. But then u would not be cyclically reduced.

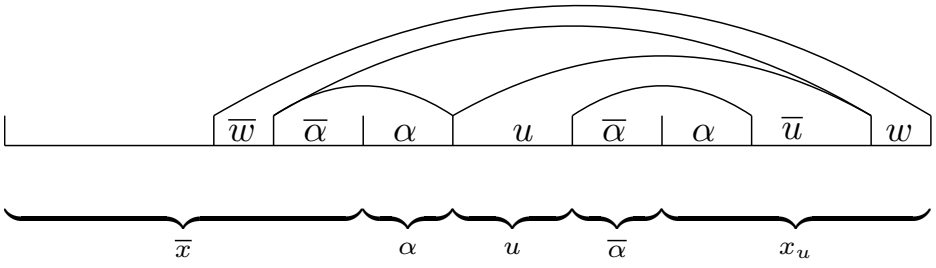

Figure 11.2: Pseudo-solution for the equation $\overline{x}\alpha u\overline{\alpha}\underline{x_u}$. Case when x_u is not reduced within $\alpha u\overline{\alpha}$ and u is paired with x_u.

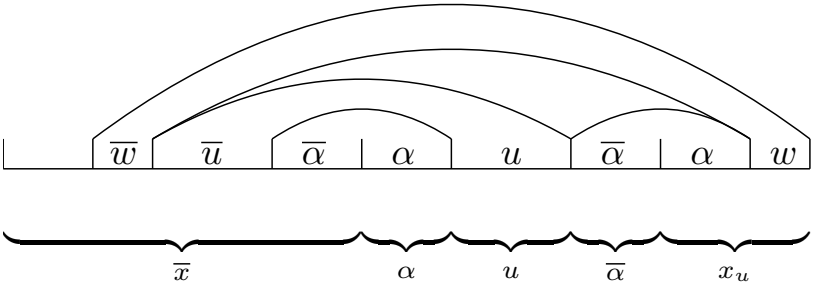

Figure 11.3: Pseudo-solution for the equation $\overline{x}\alpha u\overline{\alpha}\underline{x_u}$. Case when x_u is not reduced within $\alpha u\overline{\alpha}$ and u is not paired with x_u.

Consider first the case when (some) letters of u are paired with x_u. Then whole u is paired with x_u and the rest of x_u is paired with $\overline{x}$ (as otherwise $f(x_u) \subseteq u\overline{\alpha}$, which was already considered), see Fig. 11.2. Thus $x_u = \alpha\overline{u}w$ for some w and $\overline{x} \sqsupseteq \overline{w}\,\overline{\alpha}$, which implies $\alpha w \sqsubseteq x$. Comparing $x_u = \alpha\overline{u}w \sqsubseteq x$ with $\alpha w \sqsubseteq x$ we get that $w \sqsubseteq \overline{u}$ or $\overline{u}$ is period of w and so $x_u = \alpha\overline{u}^k\overline{u''}$, where $\overline{u''} \sqsubseteq \overline{u}$ and $k \geq 0$; this can be alternatively represented as $x_u \approx \alpha u^{-k-1}u'$, where $u = u'u''$.

The analysis for the case when some letter of u is paired with $\overline{x}$ is symmetric: let $x_u = \alpha w$. As some letter of x_u is paired with $\overline{x}$ then all letters in u are paired and so all of them are paired with $\overline{x}$, see Fig. 11.3, then $\overline{x} \sqsupseteq \overline{w}\,\overline{u}\,\overline{\alpha}$, which implies $\alpha uw \sqsubseteq x$, and $x_u = \alpha w \sqsubseteq x$, thus $w \sqsubseteq u$ or u is a period of w and so $x_u = \alpha u^k u'$ for some $u' \sqsubseteq u$ and $k \geq 0$. □

Lemma 11.7. *Let x_v be a pseudo-solution (for some partial pairing f) of $\underline{x_v}vx_ux_v$ but not in $\underline{x_v}vx_u$ (for the restriction of f). Then $x_ux_v = \overline{v''}\,\overline{v'}$ for some $v' \sqsubseteq v \sqsupseteq v''$.*

Proof. First, the whole x_u is reduced within $\mathrm{nf}(x_vvx_u)$: if not then there would be no further reduction in $\mathrm{nf}(x_vvx_u)x_v$, as x_ux_v is reduced, and so whole x_v reduces within x_vvx_u, which is forbidden by Lemma assumption. If also whole v is reduced within x_vvx_u then we are left with a prefix x_v' of x_v that should reduce with x_v, i.e. x_v' should reduce with x_v', which cannot happen. So not the whole v is reduced in $\mathrm{nf}(x_vvx_u)$, see Fig. 11.4. So $v = \delta\beta\overline{x_u}$, where δ is the maximal prefix that reduces with

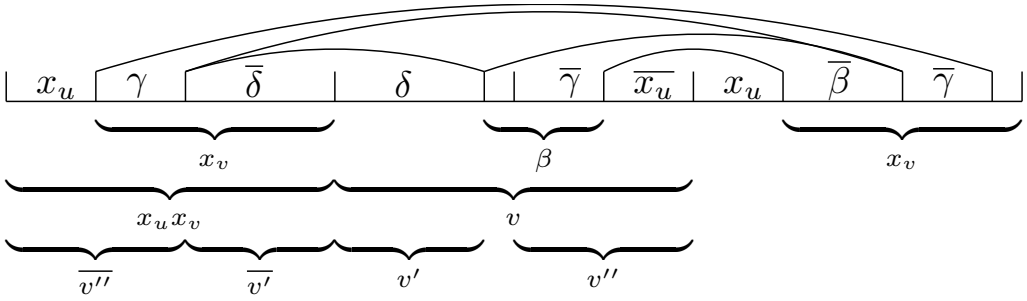

Figure 11.4: Pseudosolution for the equation $\underline{x_v}vx$. The case in which x_v reduces within $x_vvx_ux_v$ but not inside x_vvx_u. We take into account that not the whole v is reduced in $\mathrm{nf}(x_vvx_u)$.

the preceding x_v and $\overline{x_u}$ reduces with the following x_u. Then β reduces with the following x_v. Also, in the first x_v the remaining part (i.e. after reduction of δ), call it γ, reduces with the remaining part of the second x_v. So $x_v = \gamma\overline{\delta}$ and $\overline{\beta}\overline{\gamma} \sqsubseteq x_v$ and from Lemma 11.1 we get that $\gamma \sqsubseteq \overline{\beta}$ (or $\gamma = \varepsilon$, but this is covered by $\gamma \sqsubseteq \overline{\beta}$). It is left to observe that

$$x_u x_v = x_u \gamma \overline{\delta}$$

and $\overline{\gamma}\,\overline{x_u}$ is a suffix of v (as $\gamma \sqsubseteq \overline{\beta}$ implies $\beta \sqsupseteq \overline{\gamma}$) and δ is a prefix of v, so $x_u x_v = \overline{v''}\,\overline{v'}$ for some prefix v' $(= \gamma)$ of v and suffix v'' $(= \overline{\delta}\,\overline{x_u})$, as claimed. □

Lemma 11.8. *Let x be a pseudo-solution of $\overline{x}\alpha u\overline{\alpha}\underline{x}\beta v\overline{\beta}\overline{x}$, where $x, \alpha u\overline{\alpha}, \beta v\overline{\beta}$ are reduced and u, v are cyclically reduced. Then $x = x_u x_v$, where*

- *$x_u \sqsubseteq \alpha\overline{u}\,\overline{\alpha}$ or*
- *$x_u \approx \alpha u^i u'$ where $u' \sqsubseteq u$ and $i \in \mathbb{Z}$*

similarly

- *$\beta\overline{v}\,\overline{\beta} \sqsupseteq x_v$ or*
- *$x_v \approx v'' v^j \overline{\beta}$ where $v \sqsupseteq v''$ and $j \in \mathbb{Z}$.*

Proof. Fix a (partial) reduction pairing f such that whole middle x is paired. Define x_u, x_v such that $x = x_u x_v$, $f(x_u) \subseteq x^{-1}\alpha u\overline{\alpha}$ and $f(x_v) \subseteq \beta v\overline{\beta}x^{-1}$, i.e. as the prefix of x that is reduced to the left and suffix that is reduced to the right. Note that this is correct, as x is reduced and so no pairing is done inside it. Then Lemma 11.6 applied to $x\alpha u\overline{\alpha}x_u$ and $x_v\beta v\overline{\beta}x$ yields the claim (note that $\overline{u'}u^k \approx u''u^{k-1}$). □

Lemma 11.9. *Let x be a pseudo-solution of $\overline{x}\alpha u\overline{\alpha}\underline{x}vx$, where $x, \alpha u\overline{\alpha}, v$ are reduced and u is cyclically reduced. Then either*

- *$x = \overline{v''}\,\overline{v'}$, where $v' \sqsubseteq v \sqsupseteq v''$ or*
- *$x = x_u x_v$ where*
  - *$x_u \sqsubseteq \alpha\overline{u}\,\overline{\alpha}$ or*
  - *$x_u \approx \alpha u^i u'$ where $u' \sqsubseteq u$ and $i \in \mathbb{Z}$*

  and

  - *$\mathrm{nf}(\alpha u\overline{\alpha}\,\overline{v}) \sqsupseteq x_v$ or*
  - *$x_v \approx u''u^j\overline{\alpha}\,\overline{v}$, where $u \sqsupseteq u''$ and $j \in \mathbb{Z}$.*

Proof. Fix a (partial) reduction pairing f such that whole middle x is paired. Let x_u, x_v be such that $x = x_u x_v$ and $f(x_u) \subseteq x^{-1}u$ and $f(x_v) \subseteq vx$. Consider first the case in which $f(x_v) \not\subseteq vx_u$. Then Lemma 11.7 yields that $x = \overline{v''}\,\overline{v'}$, where $v' \sqsubseteq v \sqsupseteq v''$, as claimed. Thus we are left with the case when $f(x_v) \subseteq vx_u$, i.e. $\mathrm{nf}(\overline{x}_u\overline{v}) \sqsupseteq x_v$. Applying Lemma 11.6 to the $x\alpha u\overline{\alpha}x_u$ yields that the form of x_u is as claimed. Substituting the form of x_u to $\mathrm{nf}(\overline{x}_u\overline{v}) \sqsupseteq x_v$ yields the form of x_v. □

Lemma 11.10. *Let x be a pseudo-solution of $xu\underline{x}vx$, where x, u, v are reduced then either*

- *$x = \overline{v''}\,\overline{v'}$ or $x = \overline{u'}\,\overline{u''}$ or $x = \overline{u''}\,\overline{v'}$ or $x \approx \overline{u''}\,u^{\bullet\bullet}\overline{v}$ or $x \approx \overline{u}v^{\bullet}\overline{v'}$ where $v^{\bullet} \sqsubseteq v' \sqsubseteq v \sqsupseteq v''$, $u' \sqsubseteq u \sqsupseteq u'' \sqsupseteq u^{\bullet\bullet}$;*
- *$x = x_u x_v$, where*
  - *$x_u \sqsubseteq \alpha$ or $x_u = \alpha r_u^i r'_u$ for some $i \in \mathbb{N}$, where $r'_u \sqsubseteq r_u$ and $\overline{u}v = \alpha\overline{r}_u\overline{\alpha}$ and r_u is cyclically reduced;*

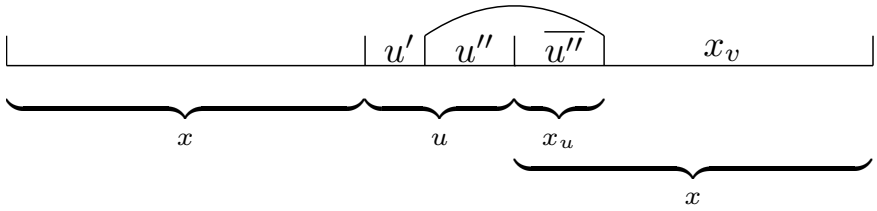

Figure 11.5: Pseudosolution for the equation $xu\underline{x_u}$; word u'' is maximal reducing with x_u. The case in which $x_u = \overline{u''}$.

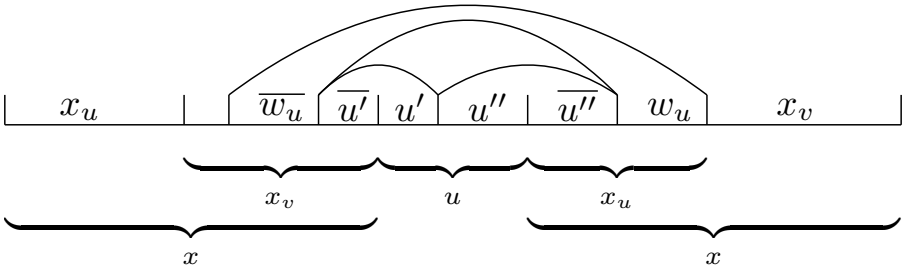

Figure 11.6: Pseudosolution for the equation $x_v u\underline{x_u}$; word u'' is maximal reducing with x_u. The case in which $x_u \neq \overline{u''}$ and so $\overline{u''} \sqsubseteq x_u$.

– *$\beta \sqsupseteq x_v$ or $x_v = r_v'' r_v^j \beta$ for some $j \in \mathbb{N}$ where $u\overline{v} = \overline{\beta} r_v \beta$ and $r_v \sqsupseteq r_v''$ and w_v is cyclically reduced;*

Proof. Fix a partial reduction pairing f such that whole middle x is paired. Define x_u and x_v such that $x = x_u x_v$ and $f(x_u) \subseteq xu$ and $f(x_v) \subseteq vx$. Consider $f(x_u) \subseteq xu = x_u x_v u$; if $f(x_u) \not\subseteq x_v u$ then by Lemma 11.7 we get that $x = \overline{u'}\,\overline{u''}$, where $u' \sqsubseteq u \sqsupseteq u''$. Similarly if $f(x_v) \not\subseteq v x_u$ then by Lemma 11.7 we get that $x = \overline{v''}\,\overline{v'}$, where $v' \sqsubseteq v \sqsupseteq v''$. So in the following we may assume that $f(x_u) \subseteq x_v u$ and $f(x_v) \subseteq v x_u$.

Let $u = u'u''$ where $f(u'') \subseteq x_u$ is maximal with this property. Either $x_u = \overline{u''}$, see Fig. 11.5, or $\overline{u''} \sqsubseteq x_u$, see Fig. 11.6. In the latter case, as not whole x_u is paired with u'', some of its letters need to be paired with the preceding x_v and so the whole u' is paired with this x_v as well, see Fig. 11.6, in particular, $x_v \sqsupseteq \overline{u'}$. Then $x_u = \overline{u''} w_u$ for some reduced $w_u \neq \varepsilon$ and $x_v \sqsupseteq \overline{w_u}\,\overline{u'}$.

Similarly, define $v = v'v''$ where $f(v') \subseteq x_v$ is maximal of this property. Either $x_v = \overline{v'}$ or $x_v = w_v \overline{v'}$ and $\overline{v''}\,\overline{w_v} \sqsubseteq x_u$ for some reduced $w_v \neq \varepsilon$, see Fig. 11.7.

Taking into account the form of x_u and x_v, there are in total four subcases:

First, if $x_u = \overline{u''}$ and $x_v = \overline{v'}$ then this is of the desired form in the first.

Consider the case when $x_u = \overline{u''}$ and $x_v = w_v \overline{v'}$: this case is left as an exercise: the claim is that $x = x_u x_v \approx \overline{u''} u^{\bullet\bullet} \overline{v}$ for some $u \sqsupseteq u'' \sqsupseteq u^{\bullet\bullet}$, as listed in the first point. (and $\overline{v''}\,\overline{w_v} \sqsubseteq x_u = \overline{u''}$); in particular, $x = \overline{u''} w_v \overline{v'}$, see Fig. 11.8.

The case of $x_u = \overline{u''} w_u$ and $x_v = \overline{v'}$ is symmetric to the above.

Now let us consider the main case: when $x_u = \overline{u''} w_u$ and $x_v = w_v \overline{v'}$, see Fig. 11.9. In particular,

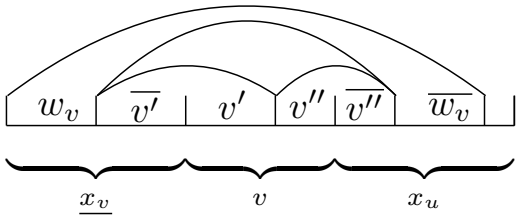

Figure 11.7: Pseudosolution for the equation $\underline{x_v} v x_u$; word v' is maximal reducing with x_v. The case in which $x_v \neq \overline{v'}$ and so $x_v \sqsupseteq \overline{v'}$.

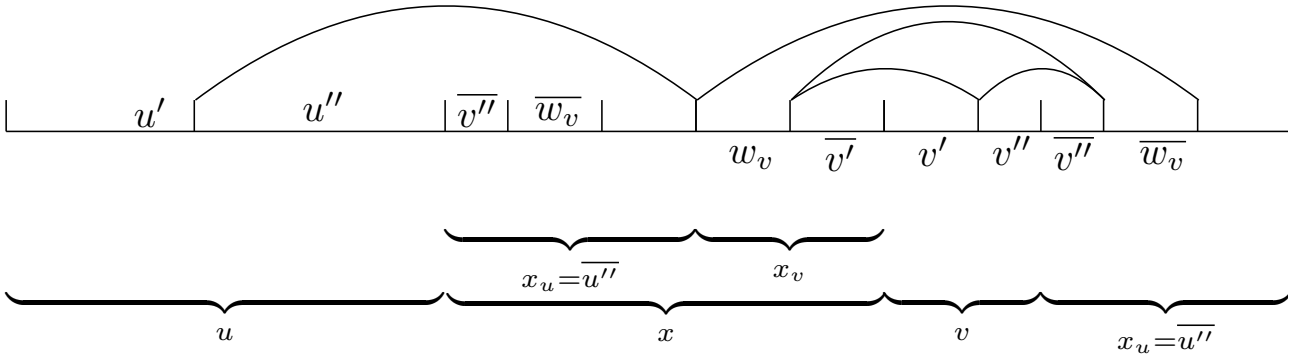

Figure 11.8: Pseudosolution for the equation $x_v u \underline{x} v x_u$. The case in which $x_u = \overline{u''}$, $x_v = w_v\overline{v'}$ and $\overline{v''}\,\overline{w_v} \sqsubseteq \overline{u''}$.

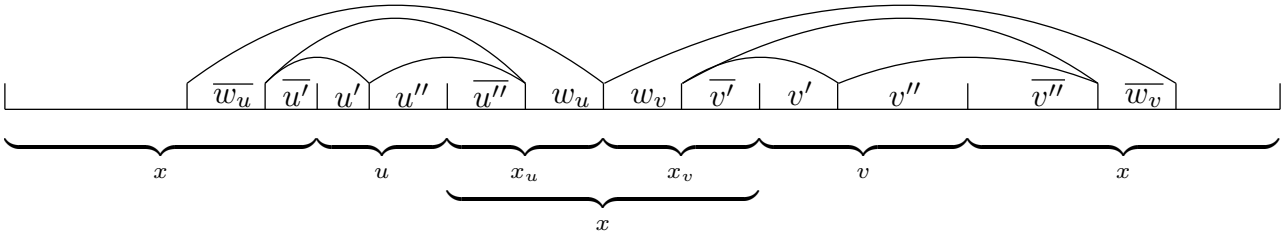

Figure 11.9: Pseudosolution for the equation $xu\underline{x}vx$. The main case: $x_u = \overline{u''}w_u$ and $x_v = w_v\overline{v'}$.

$x = \overline{u''}w_u w_v \overline{v'}$. Note that by case assumption also $x_v \sqsupseteq \overline{w_u}\,\overline{u'}$ and $\overline{v''}\,\overline{w_v} \sqsubseteq x_u$, see Fig. 11.10:

$$\overline{v''}\,\overline{w_v} \sqsubseteq \overline{u''}w_u \quad w_v\overline{v'} \sqsupseteq \overline{w_u}\,\overline{u'}$$

Let us analyze $x_u = \overline{u''}w_u$. In the following, it is convenient to consider not only the prefix relation on normal forms. Note that in general $s \sqsubseteq t$ does not imply $\mathrm{nf}(s) \sqsubseteq \mathrm{nf}(t)$ and $\mathrm{nf}(s) \sqsubseteq \mathrm{nf}(t)$ does not imply $\mathrm{nf}(s's) \sqsubseteq \mathrm{nf}(s't)$. However, if $\mathrm{nf}(s) \sqsubseteq \mathrm{nf}(t)$ and the reductions in us leading to $\mathrm{nf}(us)$ do not reduce whole s then $\mathrm{nf}(us) \sqsubseteq \mathrm{nf}(ut)$ (left as an exercise).

Note that $w_v\overline{v'} \sqsupseteq \overline{w_u}\,\overline{u'} \implies u'w_u \sqsubseteq v'\overline{w_v}$:

$$\begin{aligned}
\overline{v''}\,\overline{w_v} &\sqsubseteq \overline{u''}w_u && \\
v'v''\overline{v''}\,\overline{w_v} &\sqsubseteq v\overline{u''}w_u && \text{Multiply by } v = v'v'' \\
\mathrm{nf}(v'\overline{w_v}) &\sqsubseteq \mathrm{nf}(v\overline{u''}w_u) && \text{Reduce left-hand side} \\
v'\overline{w_v} &\sqsubseteq \mathrm{nf}(v\overline{u''}w_u) && \mathrm{nf}(v'\overline{w_v}) = v'\overline{w_v} \\
u'w_u &\sqsubseteq \mathrm{nf}(v\overline{u''}w_u) && \text{Transitivity} \\
\mathrm{nf}(\overline{u''}\,\overline{u'}u'w_u) &\sqsubseteq \mathrm{nf}(\overline{u}v\overline{u''}w_u) && \text{Multiply by } \overline{u} = \overline{u''}\,\overline{u'} \\
\mathrm{nf}(\overline{u''}w_u) &\sqsubseteq \mathrm{nf}(\overline{u}v\overline{u''}w_u) && \text{Reduce left-hand side} \\
x_u &\sqsubseteq \mathrm{nf}(\overline{u}vx_u) && x_u = \overline{u''}w_u = \mathrm{nf}(\overline{u''}w_u)
\end{aligned}$$

Multiplying by v is legal: $\overline{v''}\,\overline{w_v}$ is irreducible and after the multiplication the remaining $v'\overline{w_v}$ is also irreducible. Similarly, after the multiplication by $\overline{u}$ the $\overline{u''}w_u$ is irreducible, so the multiplication is legal.

Let $\mathrm{nf}(\overline{u}v) = \alpha r_u \overline{\alpha}$, where r_u is cyclically reduced. Observe that in $\alpha r_u \overline{\alpha} x_u$ we can reduce at most half of letters in $\alpha r_u \overline{\alpha}$, as otherwise $\mathrm{nf}(\alpha r_u \overline{\alpha} x_u)$ has less letters than x_u, which is its prefix. Hence $\mathrm{nf}(\alpha r_u \overline{\alpha} x_u)$ begins with αa, where a is the first letter of r_u. If $x_u \sqsubseteq \alpha$ then we are done. Otherwise $x_u = \alpha a x''_u$ and so $\alpha r_u \overline{\alpha} x_u \approx \alpha r_u a x''_u$ and $a x''_u \sqsubseteq \mathrm{nf}(r_u a x''_u)$. Note that $r_u a x''_u$ is reduced: if there is a reduction in $r_u a$ then the last letter of r_u is $\overline{a}$ and a is the first letter of r_u, contradiction, as r_u is

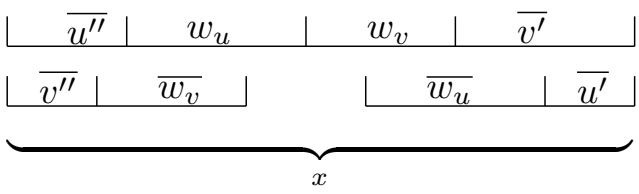

Figure 11.10: Pseudosolution for the equation $xu\underline{x}vx$. The main case: $x_u = \overline{u''}w_u$ and $x_v = w_v\overline{v'}$. The depiction of prefixes and suffixes of x.

cyclically reduced. Hence $ax''_u \sqsubseteq r_u a x''_u$ and so $ax''_u = r_u^i r'_u$ for some $i \geq 0$ and $r'_u \sqsubseteq r_u$, and so $x_u = \alpha r_u^i r'_u$, as claimed.

A similar analysis applies to x_v. □

Lemma 11.11. *If s, t are cyclically reduced or st is cyclically reduced and $k_1, k_2, \ldots, k_{2\ell}, k_{2\ell+1}$, for $\ell \geq 1$, are non-zero integers such that*

$$s^{k_1} t^{k_2} \cdots s^{k_{2\ell-1}} t^{k_{2\ell}} \approx \varepsilon \quad \text{or} \quad s^{k_1} t^{k_2} \cdots s^{k_{2\ell-1}} t^{k_{2\ell}} s^{k_{2\ell+1}} \approx \varepsilon \tag{11.3}$$

then s, t are powers of the same word.

In particular, if s, t are not powers of the same word then the mapping $S \mapsto s$, $T \mapsto t$ defines an isomorphism between the subgroup generated by s, t and a free group generated by S, T.

The proof almost follows from a known theorem that a subgroup of a free group is a free group. To be more precise, the proof shows that using a set of finite generators we can either obtain a trivial word or those generators are free.

11.4 Data structure

Words appearing naturally in our proofs and algorithms are concatenations of a constant number of subwords (or their involutions) of the input equation. We say that a word w is *k-represented*, if w is given as $w = (U\overline{U})[b_1 \,..\, e_1] \cdots (U\overline{U})[b_k \,..\, e_k]$, where $U = u_1 \cdots u_m$ is the concatenation of all words from the equation (11.2). A parametric word $s_0 t_1^{\varphi_1} s_1 \cdots s_{\ell-1} t_\ell^{\varphi_\ell} s_\ell$ is k-represented, when $s_0, t_1, s_1, \ldots, t_\ell, s_\ell$ are $k_0, \ldots, k_{2\ell}$ represented and $k = \sum_{i=0}^{2\ell} k_i$.

Intuitively, all basic operations that we perform on (parametric) words that are k and ℓ represented can be performed in $\mathcal{O}(k + \ell)$ time. As k, ℓ are usually small constants this amounts to $\mathcal{O}(1)$ time.

We use standard data structures, like suffix arrays [24] and structures for answering longest common prefix queries on them [25]. As a result, we can answer all basic queries (like normal form, longest common prefix, power prefix, etc.) about words in the equation in $\mathcal{O}(1)$ time; note that this is the place in which we essentially use that we can perform operations on $\mathcal{O}(\log n)$-size numbers in $\mathcal{O}(1)$ time. As an example of usage, we can test whether a word is a solution in $\mathcal{O}(m)$ time:

Lemma 11.12. *For the equation* (11.2) *we can construct a data structure, which given two words s, t that are k and ℓ represented, we can:*

- *compute the longest prefix of s that has period p in $\mathcal{O}(k)$ time,*
- *compute the s-power prefix and suffix of t in $\mathcal{O}(k + \ell)$ time,*
- *compute* $\mathrm{nf}(st)$ *in $\mathcal{O}(k + \ell)$ time.*

Lemma 11.13. *Given a word $\alpha u^i v^j \beta$, where α, β, u, v are $\mathcal{O}(1)$-represented, α, β are reduced and u, v are cyclically reduced and primitive and i, j are a pair of integer numbers, we can test whether $\alpha u^i v^j \beta$ is a solution of* (11.2) *in $\mathcal{O}(m)$ time.*

11.5 Restricting the superset of solutions

By Lemma 11.5, we know the form of possible solutions, and by Lemma 11.13 we can test a single candidate solution in $\mathcal{O}(m)$ time. In particular, all solutions from the set S in Lemma 11.5 can be tested in $\mathcal{O}(n^2m)$ time, as desired. The other solutions are instances of parametric words the form $\alpha u^I v^J \beta$ for well-defined α, u, v, β. The next step is to bound, for fixed α, u, v, β, the set of values (i, j) such that $\alpha u^I v^J \beta(i, j)$ could be a solution; this is the main result of the paper.

Idea Suppose we want to find out which words of the form u^i are a solution of (11.2). We substitute u^I to the equation and treat its left-hand side as a parametric word w depending on I. If substituting $I = i$ leads to a trivial word, then it is known that some u-power cancels within the neighboring u-powers (actually, a variant of this fact was used to characterize the superset of solutions [39, 1, 3], and it is attributed already to Nielsen [47]), more formally:

We want to use Lemma 11.4 to claim that some u-parametric powers need to reduce, however, as there can be powers of u as constants, this makes the analysis problematic: as an example, consider an equation $au^I u^\ell \overline{a} \approx \varepsilon$, if $I = i$ is a solution and we set $s_0 = a, u_1 = u^i, s_1 = u^\ell \overline{a}$ (so that u_1 corresponds to u^I) then Lemma 11.4 guarantees that u^i cancels within u^ℓ, i.e. $0 \geq i \geq -\ell$, even though $I = -\ell$ is the only solution. This is caused by u-powers next to u-parametric power, which makes our application of the Lemma 11.4 nearly useless. To fix this, in $au^i u^\ell \overline{a}$ we set $s_0 = a, u_1 = u^{i+\ell}, s_1 = \overline{a}$, and then Lemma 11.4 yields $i = -\ell$. On the level of the parametric word this corresponds to considering $au^{I+\ell}\overline{a} \approx au^I u^\ell \overline{a}$, i.e. we include u-powers into the u-parametric power next to them.

This is formalized as follows: A parametric word w is *u-reduced* when u is cyclically reduced, primitive and w does not have a subword of the form:

- u^φ for a constant integer expression φ;
- $a\overline{a}$ for some letter a (so w is reduced);
- $u^\varphi u^\psi$ for some (non-constant) integer expressions φ, ψ;
- $uu^\varphi, \overline{u}u^\varphi, u^\varphi u, u^\varphi \overline{u}$ for some (non-constant) integer expression φ.

Note that we do not forbid subwords that are powers of u, we forbid parametric subwords that are in fact subwords, i.e. have constant exponents.

Given a parametric word w we can u-reduce it to obtain a parametric word that is equal (in the free group) and u-reduced by a simple greedy procedure, i.e. replacing a parametric power with a constant integer expression as exponent with a power or reduction or joining two u-powers into one (the running time for specific applications is analyzed separately at appropriate places). When we replace, say uu^φ with $u^{\varphi+1}$, then we say that letters in u were u-reduced to $u^{\varphi+1}$. Note that there are different u-reduced equivalent parametric words, so the output of u-reduction is not unique, this has no effect on the algorithm, though.

If a parametric word w (with all exponents depending on one variable) is u-reduced then from Lemma 11.4 we infer that $w(i) \approx \varepsilon$ implies $|\varphi(i)| \leq 3$ for some parametric power u^φ in w:

Lemma 11.14. *Let $w = w_0 u^{\varphi_1} w_1 \cdots u^{\varphi_k} w_k$ be a u-reduced parametric word, where $w_0, \ldots, w_k$ are words and $\varphi_1, \ldots, \varphi_k$ are integer expressions, all depending on exactly one and same variable. If $w(i) \approx \varepsilon$ then there is φ_ℓ such that $|\varphi_\ell(i)| \leq 3$. In particular, $w(i) \approx \varepsilon$ for each i if and only if $w = \varepsilon$.*

Proof. We use Lemma 11.4 for a factorization with $s_j = w_j$ and $u_j = u^{\varphi_j(i)}$. Then for some ℓ we have that $u^{\varphi_\ell(i)}$ reduces within $u^{\varphi_{\ell-1}(i)} w_{\ell-1} u^{\varphi_\ell(i)} w_\ell u^{\varphi_{\ell+1}(i)}$. Suppose that $|\varphi_\ell(i)| \geq 4$, say $\varphi_\ell(i) > 0$, the other case is shown in the same way. Consider $u^{\varphi_{\ell-1}(i)} w_{\ell-1} u^2 u^{\varphi_\ell(i)-4} u^2 w_\ell u^{\varphi_{\ell+1}(i)}$. Then it can be shown that the reductions in $u^{\varphi_{\ell-1}(i)} w_{\ell-1} u^2$ and $u^2 w_\ell u^{\varphi_{\ell+1}(i)}$ are both of length less than $2|u|$, thus not the whole $u^2 u^{w_\ell - 4} u^2$ is reduced, contradiction.

For the last claim, if w contains u-parametric powers, then clearly there is a finite set of is such that $w(i) \approx \varepsilon$. If it does not, then as it is u-reduced, it is also reduced, and so $w \approx \varepsilon$ implies $w = \varepsilon$. □

As φ_ℓ in Lemma 11.14 is a non-constant integer expression then there are at most 7 values of i such that $|\varphi_\ell(i)| \leq 3$. Hence it is enough to find appropriate i values. Clearly, there are at most m integer expressions in w (as this is the number of variables). We can give better estimations, though: if the expression is not of the form kI then it "used" at least $|u|$ letters from the equation. So there are $n/|u|$ different expressions and the ones of the form kI; as $|ki| \leq 3$ implies $|i| \leq 3$, there are $7(1 + n/|u|)$ candidates for i in total. Lastly, when the solution depends on two variables, it can be shown that all obtained parametric powers have coefficient ± 1, which allow even better estimations: a parametric power $I + c$ uses at least $c|u|$ letters from the equation and so it can be shown that at most $\mathcal{O}(\sqrt{n/|u|})$ different integer expressions can be formed in such a case.

The actual solution is of the form $\alpha u^I v^J \beta$. Firstly, the presence of α, β make estimations harder, as their letters can also be used in the u- and v-reductions. Secondly, there are two parameters, which makes a simple usage of Lemma 11.14 impossible. However, if $w(i, j) \approx \varepsilon$ then $w(I, j) \approx \varepsilon$ depends on one variable, so Lemma 11.14 is applicable to it. The analysis yields that we can restrict the possible value of i or j or (i, j); note that this is non-obvious, as there are infinitely many $w(I, j)$s. A similar analysis can be made for $w(i, J)$, and combining those two yields a set of pairs to be tested as well as $\mathcal{O}(1)$ individual is and js that should be tested separately. But for a fixed i (j) we can substitute it to the equation and use Lemma 11.14 for J (I, respectively).

11.5.1 Restricting the set of (i, j)

Fix some $0 \leq i_0 \leq m - 1$ and the corresponding u_{i_0}, u_{i_0+1} in the equation (11.2). Using Lemma 11.5 we construct a parametric word $\alpha u^I v^J \beta$, with α, u, v, β depending on u_{i_0}, u_{i_0+1} as well as exponents $p_{i_0}, p_{i_0+1}, p_{i_0+2}$. We substitute $X = \alpha u^I v^J \beta$ to the equation (11.2), obtaining a parametric word on the left-hand side. We are to find values $(i, j) \in \mathbb{Z}^2$ for which the value of the obtained parametric word is equivalent to ε, thus we call such an (i, j) a solution. We want to find a suitable set of pairs (i, j) and test each one individually, using Lemma 11.13.

The analysis depends on the relation between u and v: i.e. whether $u \in \{v, \overline{v}\}$, $u \not\sim v$ or $u \sim v$.

$u \not\sim v$

Due to symmetry, we consider the case when $|v| \geq |u|$, note that it could be that $|u| = |v|$. We rotate the left-hand side of the equation so that it begins and ends with a parametric power: we rotate $\alpha u^I v^J \beta w = \varepsilon$ to $v^J \beta w \alpha u^I = \varepsilon$ or $\overline{\beta} \overline{v}^J \overline{u}^I \overline{\alpha} w = \varepsilon$ to $\overline{u}^I \overline{\alpha} w \overline{\beta} \overline{v}^J = \varepsilon$, depending on the form of the equation. The equation after the rotation is equisatisfiable to the previous one.

We call each parametric word beginning with v^J or $\overline{u}^I$ and ending with u^I or $\overline{v}^J$ and no parametric power inside a *fragment*. The parametric word after the rotation is a concatenation of m fragments. We use the name h-th fragment to refer to the one corresponding to u_h (so h-th from the left); let f_h denote the word that is left from h-th fragment after removing the leading and ending parametric power; note that f_h is of one of the forms $\beta u_h \alpha$, $\beta u_h \overline{\beta}$, $\overline{\alpha} u_h \alpha$, $\overline{\alpha} u_h \overline{\beta}$. For u^I we call the preceding α the associated word, the same name is used to β succeeding v^J, $\overline{\alpha}$ succeeding $\overline{u}^I$ and $\overline{\beta}$ preceding $\overline{v}^J$. To simplify, we will call it a word associated with the parametric power.

We now preprocess the equation, by replacing the left-hand side with an equivalent parametric word (i.e. equal according to $\approx$). As a first step, we replace each f_h with $\mathrm{nf}(f_h)$. Next, observe that if w is the power of u then $\overline{u}^I w u^I \approx w$ and similarly $v^J w' \overline{v}^J \approx w'$ for w' being a power of v. In the second step we check each fragment separately, and if possible, replace it as described above. For fragments that remained unchanged in the second step, we use previous names, i.e. if h-th fragment $v^J \,\mathrm{nf}(f_h) u^I$ was not replaced then we still write it as $v^J \,\mathrm{nf}(f_h) u^I$ and call it h-th fragment. A *trivial fragment* is a maximal subword obtained as concatenations of words obtained due to replacements in the second step.

Lemma 11.15. *The preprocessing can be performed in $\mathcal{O}(m)$ time. Afterwards the parametric word is $\mathcal{O}(m)$-represented and it is a concatenation of fragments and trivial fragments.*

Each trivial fragment is obtained by replacing some h-th, $h + 1$-st, ..., $h + k$-th fragments by $\mathrm{nf}(f_h \cdots f_{k+h})$ moreover $|\,\mathrm{nf}(f_h \cdots f_{k+h})| \leq \sum_{i=h}^{h+k} |u_i|$; if $k > 0$ then such a trivial factor is not a power of u nor v.

We now perform the u-reduction (note that the v^J is not touched) and afterwards the v-reduction. Let the obtained equation be of the form

$$W \approx \varepsilon \ , \tag{11.4}$$

where W is a parametric word. In the following, we are looking for (i,j)s such that $w(i,j) \approx \varepsilon$, and so we simply call (i,j) a solution (of (11.4)).

Lemma 11.16. *For $u \not\sim v$ we can perform the u-reduction and v-reduction after the preprocessing in $\mathcal{O}(m)$ time; the obtained parametric word is u-reduced. No two parametric powers are replaced by one during the u-reduction and v-reduction, in particular, for a given parametric power u^φ (v^ψ) in* (11.4) *the φ (ψ) has a coefficient of the variable equal to ± 1 and the only letters that are u-reduced (v-reduced) to this power come either from the associated fragment of u^I or $\overline{u}^I$ (v^J or $\overline{v}^J$) and the letters from the adjacent trivial fragment (assuming that there is an adjacent trivial fragment).*

Note that the claim that no two parametric powers are replaced by one is not obvious—in principle, it could be that after the preprocessing a trivial fragment is a power of u (or v) and then it is wholly u-reduced, which can lead to two adjacent parametric powers of u, which are then replaced with one. However, this cannot happen, as such a trivial fragment is of the form $u^{k_1}v^{k_2}\cdots$ for some $0 < |k_1|, |k_2|, \ldots$ and such a word cannot be a power of u nor v when $u \not\sim v$, as the subgroup generated by u, v is a free group.

We now estimate, how many different u-parametric expressions are there after the reductions. When we want to distinguish between occurrences of parametric powers with the same exponent (say, two occurrences of u^{I+1} counted separately) then we write about parametric powers and when we want to treat it as one, then we talk about exponents. We provide two estimations, one focuses on parametric powers and the other on exponents.

Lemma 11.17. *There is a set S of $\mathcal{O}(1)$ size of integer expressions such that there are $\mathcal{O}(n/|u|)$ occurrences of u-parametric powers in W from* (11.4) *whose exponents are not in S and $\mathcal{O}(n/|v|)$ occurrences of v-parametric powers whose exponents are not in S. The set S can be computed and the parametric powers identified in $\mathcal{O}(m + n/|u|)$ time.*

The Lemma considers, whether the parametric power used some letters from the trivial fragment or its associated fragment had u_h of length at least $|u|$. If so, then it is in the $\mathcal{O}(n/|u|)$ parametric powers, as one such power uses at least $|u|$ letters of the input equation (this requires some argument for the trivial fragments) and otherwise is can be shown that there are only $\mathcal{O}(1)$ possible exponents: say, when we consider the longest suffix of $\mathrm{nf}(\beta u_h \alpha)$ that is a u-power, where $|u_h| < |u|$, then there is a constant number of possibilities how this suffix is formed (fully within α, within $\mathrm{nf}(u_h\alpha)$, uses some letters of β) and in each case the fact that $|u_h| < |u|$ means that there are only $\mathcal{O}(1)$ different u_hs that can be used; note that we need the primitivity of u here. Concerning the algorithm, note that we can distinguish between these two cases during the preprocessing and mark the appropriate powers.

The next lemma provides a better estimation for the number of different exponents, it essentially uses the fact that all exponents have coefficients at variables ± 1: as there are only two possible coefficients, we can focus on the constants. Now, to have a constant $|c|$, we have to use a power u^c from W and to have k *different* constants one has to use k different powers and so from Lemma 11.2 we conclude that $k = \mathcal{O}(|W|/|u|)$. In general, W can be of quadratic length, as we introduce m copies of α and β into it; the resulting bound is too weak for our purposes. To improve the bound, consider that when the u-power suffix of, say, $\beta u_h \alpha$, is u^k. It can be shown that there are k_α, k_u, k_β such that $|k - k_\alpha - k_u - k_\beta| \leq 2$ and u^{k_u}, u^{k_β} are maximal u-powers in u_h, β and u^{k_α} is the u-power suffix of α. Using Lemma 11.2, this yields that there are $\mathcal{O}(\sqrt{n/|u|})$ different possible values of k_u (over all u_h), $\mathcal{O}(\sqrt{|\beta|/|u|}) = \mathcal{O}(\sqrt{|u_{i_0}u_{i_0+1}|/|u|})$ of k_β and k_α is fixed, so there are at most $\mathcal{O}(\sqrt{n/|u|} \cdot \sqrt{|u_{i_0}u_{i_0+1}|/|u|}) = \mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|/|u|})$ possible values of k.

The actual argument is more involved, as it is also possible that the u-parametric power includes letters from the trivial fragments, which requires some extra arguments, nevertheless the general approach is similar.

Lemma 11.18. *After the u-reduction and v-reduction there are $\mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|}/|u|)$ different integer expressions as exponents in parametric powers of u and $\mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|}/|v|)$ of v in the equation. The (sorted) lists of such expressions can be computed in $\mathcal{O}(m+n/|u|)$ and $\mathcal{O}(m+n/|v|)$ time, respectively.*

We can use Lemma 11.4 together with bounds on the number of different exponents in parametric powers from Lemma 11.18 to limit the possible candidates (i, j) for a solution. However, these bounds are either on i or on j. And as soon as we fix, say, $J = j$ and substitute it to W, the obtained parametric word $W(I, j)$ (or $W(i, J)$) is more complex than W, in particular, we do not have the bounds of Lemma 11.18 for it, so the set of possible candidates for i for a given $W(I, j)$ is linear, which is too much for the desired running time.

Instead, we analyze (as a mental experiment) $W(I, j)$: Fix $j \in \mathbb{Z}$ such that $W(i, j) \approx \varepsilon$ for some i. Compute $W(I, j)$, u-reduce it, call the resulting parametric word $W_{J=j}$. If $W_{J=j} = \varepsilon$, then clearly for each i the (i, j) is a solution of (11.4) (and vice-versa, see Lemma 11.14). It can be shown that in this case for some v^ψ in $W_{J=j}$ it holds that $|\psi(j)| < 6$: at least some two u-parametric powers in W should be merged in $W_{J=j}$, in W they are separated by a v-parametric power, say v^ψ. All letters of $v^{\psi(j)}$ are u-reduced, then standard arguments using periodicity show that $|\psi(j)| < 6$ so we can compute all candidates for such js and test for each one whether indeed $W_{J=j} = \varepsilon$, this is formally stated in Lemma 11.21.

If $W_{J=j}$ depends on I then from Lemma 11.14 for some of the (new) u-parametric powers u^φ it holds that $|\varphi(i)| < 6$. Consider, how this φ was created. It could be that it is (almost) unaffected by the second u-reduction and so it is (almost) one of the u-parametric powers in W, see Lemma 11.22 for precise formulation and sketch of proof, in which case we can use Lemma 11.18. Intuitively, u^φ is affected if the whole two parametric powers in W were used to create u^φ. Then it can be shown that some v-parametric power v^ψ from W turned into v-power $v^{\psi(j)}$ satisfies $|\psi(j)| < 6$ and is u-reduced to u^φ, the argument is as before, when $W_{J=j} \approx \varepsilon$. Moreover, this occurrence of v^ψ also determines u^φ; hence the choice of ψ determines $\mathcal{O}(1)$ candidates for j, uniquely identifies φ and i satisfies $|\varphi(i)| < 6$, i.e. there are $\mathcal{O}(1)$ candidates for (i, j). Then Lemma 11.17 is applied to this v^ψ: if it is one of $n/|v|$ occurrences of v-parametric powers then we get $\mathcal{O}(1)$ candidates for (i, j) (for this ψ), so $\mathcal{O}(n/|v|)$ in total, over all choices of such ψ. Otherwise, ψ it is one of $\mathcal{O}(1)$ integer expressions (Lemma 11.17) and so j is from $\mathcal{O}(1)$-size set and we can compute and consider $W_{J=j}$ for each one of them separately.

A similar analysis applies also to $i \in \mathbb{Z}$ substituted for I. The results are formalized in the Lemma 11.19 below, its proof is spread across a couple of Lemmata.

Lemma 11.19. *Given equation (11.4) we can compute in $\mathcal{O}(mn/|u|)$ time sets $S_I, S_J, S_{\mathbb{Z},J} \subseteq \mathbb{Z}$ and $S_{I,J} \subseteq \mathbb{Z}^2$, where $|S_I| = \mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|}/|u|)$, $|S_J| = \mathcal{O}(1)$, $|S_{\mathbb{Z},J}|, |S_{I,J}| = \mathcal{O}(n/|u|)$, such that: if (i, j) is a solution of (11.4) then at least one of the following holds:*

- *$i \in S_I$ or*
- *$j \in S_J$ or*
- *$j \in S_{\mathbb{Z},J}$ and for each i' the (i', j) is a solution or*
- *$(i, j) \in S_{I,J}$.*

Similarly, given equation (11.4) we can compute in $\mathcal{O}(mn/|v|)$ time sets $S'_I, S'_J, S'_{I,\mathbb{Z}} \subseteq \mathbb{Z}$ and $S'_{I,J} \subseteq \mathbb{Z}^2$, where $|S'_I| = \mathcal{O}(1)$, $|S'_J| = \mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|}/|v|)$ $|S'_{I,\mathbb{Z}}|, |S'_{I,J}| = \mathcal{O}(n/|v|)$ such that at if (i, j) is a solution of (11.4) then least one of the following holds:

- *$i \in S'_I$ or;*
- *$i \in S'_{I,\mathbb{Z}}$ and for each $j' \in \mathbb{Z}$ the (i, j') is a solution or;*
- *$j \in S'_J$ or;*
- *$(i, j) \in S'_{I,J}$.*

As noted above, the main distinction is whether the u^φ in $W_{J=j}$ was "affected" or not during the second u-reduction. Let us formalize this. Given an occurrence of a parametric power u^φ in $W_{J=j}$ consider the largest subword w of W such that each letter in $w(I,j)$ is either reduced or u-reduced to this u^φ; note that this may depend on the order of reductions, we fix an arbitrary order. We say that parametric powers in w are *merged* to u^φ. We extend this notion also to the case when $W_{J=j} = \varepsilon$, in which case $W = w$ and every parametric power is merged to the same parametric power u^0. A similar notion is defined also for parametric powers of v. Note that a parametric power is not merged to two different parametric powers u^φ and $u^{\varphi'}$.

Lemma 11.20. *For any parametric power in W there is at most one parametric power in $W_{J=j}$ to which it was merged; the same holds for $W_{I=i}$.*

We say that a u-parametric power u^φ in $W_{J=j}$ *was affected* by substitution $J = j$ if

- more than one parametric power was merged to u^φ *or*
- for the unique u-parametric power $u^{\varphi'}$ merged to u^φ there is a v-parametric power $v^{\psi'}$ such that $|\psi'(j)| < 6$ and there is no u-parametric power between $u^{\varphi'}$ and $v^{\psi'}$.

The intuition behind the first condition is that when we merge two u-powers then we create a completely new parametric power, for the second condition, when $|\psi'(j)| < 6$ then $v^{\psi'(j)}$ no longer behaves like $v^{\psi'}$ and can either be wholly merged to a u-power or be canceled by a trivial fragment, which can also lead to a large modification of the neighbouring u-parametric power. Note that the second condition could be made more restrictive, but the current formulation is good enough for our purposes.

We first investigate the case, when the parametric power was affected by a substitution.

Lemma 11.21. *In $\mathcal{O}(mn/|v|)$ time we can compute and sort sets $S_J, S_{E,J}$, where $|S_J| = \mathcal{O}(1)$ and $|S_{E,J}| = \mathcal{O}(n/|v|)$, such that for each occurrence of a u-parametric power u^φ in $W_{J=j}$ affected by the substitution $J = j$ either $j \in S_J$ or $(\varphi, j) \in S_{E,J}$.*

Similarly, in time $\mathcal{O}(mn/|u|)$ we can compute and sort sets $S'_I, S_{I,E}$, where $|S'_I| = \mathcal{O}(1)$ and $|S_{I,E}| = \mathcal{O}(n/|u|)$, such that for each occurrence of a v-parametric power v^ψ in $W_{I=i}$ affected by the substitution $I = i$ either $i \in S'_I$ or $(i, \psi) \in S_{I,E}$.

The sketch of the argument was given above Lemma 11.19. Concerning the running time, the appropriate exponents are identified during the u-reduction and v-reduction, which are performed in given times using the data structure.

We now consider the case when u^φ was not affected. Essentially, we claim that u^φ is almost the same as some $u^{\varphi'}$ in W. The difference is that it can u-reduce letters from v-parametric powers that become v-powers. However, as such v-power is not wholly merged (as it is not affected), only its proper suffix or prefix can be u-reduced and by primitivity and by case assumption $u \not\sim v$ and $|v| \geq |u|$, this suffix is of length at most $|v| + |u|$. Thus, while in principle there are infinitely many possibilities for $v^\psi(j)$ when $j \in \mathbb{Z}$, it is enough to consider a constant number of different candidates (roughly: $\overline{v}^2, \overline{v}, \varepsilon, v, v^2$) and we can procure all of them so that an analysis similar to the one in Lemma 11.18 can be carried out: essentially we replace a fragment $v^J f_h u^I$ with 5 "fragments" $v^c f_h u^I$ for $c \in \{-2, -1, 0, 1, 2\}$. In this argument, we used the assumption that $|v| \geq |u|$ (the u-reduction is of length at most $|v| + |u| \leq 2|v|$), but it turns out that in the case v-parametric powers the argument is even simpler: the v-reduced prefix of u-parametric power is of length at most $2|v|$, so the v-parametric power is modified by an additive $\mathcal{O}(1)$ summand.

Lemma 11.22. *We can compute and sort in $\mathcal{O}(m + n/|u|)$ time a set of $\mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|}/|u|)$ integer expressions E such that for every j if u^φ is a parametric power in $W_{J=j}$ not affected by substitution $J = j$ then $\varphi \in E$.*

A similar set of $\mathcal{O}(\sqrt{n|u_{i_0}u_{i_0+1}|}/|v|)$ integer expressions can be computed for the not affected v-parametric powers after the second v-reduction in $\mathcal{O}(m + n/|v|)$ time.

Note that the second u (or v) reduction is performed only for some chosen values of i and j, and not for each possible one.

Lemmata 11.17, 11.18, 11.21 and 11.22 are enough to prove Lemma 11.19, by a simple case distinction, as described in text preceding Lemma 11.17.

What is left to show is how to compute candidate solutions, when one of I, J, say J, is already fixed, as in the claim of Lemma 11.19. The analysis is similar as in the case of two parameters, however, we cannot guarantee that after the u-reduction the coefficient at the u-parametric powers are ± 1. On the positive side, as there is only one integer variable, we can apply Lemma 11.14 directly. The additional logarithmic in the running time is due to sorting, which now cannot be done using counting sort, as the involved numbers may be large.

Lemma 11.23. *For any given j in $\mathcal{O}(m)$ time we can decide, whether for each $i \in \mathbb{Z}$ the $\alpha u^i v^j \beta$ is a solution of (11.2) and if not then in $\mathcal{O}(m + n \log m/|u|)$ time compute a superset (of size $\mathcal{O}(n/|u|)$) of is such that $\alpha u^i v^j \beta$ is a solution.*

A similar claim holds for any fixed i (with superset size $\mathcal{O}(n/|v|)$ and running time $\mathcal{O}(m + n \log m/|v|)$).

Exercises

Task 61 Show that if $u_1 w u_2 = v_1 \overline{w} v_2$ and w is reduced then either $w = 1$ or $v_1 \overline{w} \sqsubseteq u_1$ or $u_1 w \sqsubseteq v_1$, i.e. w and $\overline{w}$ cannot overlap.

Task 62 Consider the case of x being a pseudo-solution of $xu\underline{x}vx$, where $x_u = \overline{u''}$ and $x_v = w_v \overline{v'}$: (and $\overline{v''}\,\overline{w_v} \sqsubseteq x_u = \overline{u''}$); in particular, $x = \overline{u''} w_v \overline{v'}$.

Show that $x = x_u x_v = \mathrm{nf}(\overline{u''} u^{\bullet\bullet} \overline{v})$ for some $u \sqsupseteq u'' \sqsupseteq u^{\bullet\bullet}$; note that there is a reduction in $\overline{u''} u^{\bullet\bullet}$

Task 63 Let $x_u = \overline{u''} w_u$, $x_v = w_v \overline{v'}$ (in particular, $x = \overline{u''} w_u w_v \overline{v'}$) and also $x_v \sqsupseteq \overline{w_u}\,\overline{u'}$ and $\overline{v''}\,\overline{w_v} \sqsubseteq x_u$. All above are in nf; this corresponds to the main case of pseudo-solution of $xu\underline{x}vx$.

Justify the argument (especially the steps involving nf)

$$\begin{aligned}
\overline{v''}\,\overline{w_v} &\sqsubseteq \overline{u''} w_u \\
v' v'' \overline{v''}\,\overline{w_v} &\sqsubseteq v \overline{u''} w_u \\
\mathrm{nf}(v' \overline{w_v}) &\sqsubseteq \mathrm{nf}(v \overline{u''} w_u) \\
v' \overline{w_v} &\sqsubseteq \mathrm{nf}(v \overline{u''} w_u) \\
u' w_u &\sqsubseteq \mathrm{nf}(v \overline{u''} w_u) \\
\mathrm{nf}(\overline{u''}\,\overline{u'} u' w_u) &\sqsubseteq \mathrm{nf}(\overline{u} v \overline{u''} w_u) \\
\mathrm{nf}(\overline{u''} w_u) &\sqsubseteq \mathrm{nf}(\overline{u} v \overline{u''} w_u) \\
x_u &\sqsubseteq \mathrm{nf}(\overline{u} v x_u)
\end{aligned}$$

Task 64 Let s be a cyclically reduced word. Let W be a set of words and $k = \sum_{w \in W} |w|$. Suppose that $s^{k_1}, \ldots, s^{k_p}$ are pairwise disjoint subwords of words in W and that $k_1, \ldots, k_p$ are pairwise different integers. Show that $p \leq \sqrt{4k/|s| + 1}$.

Chapter 12

Approaches to dimension

In this chapter we will try (and fail to great extent) to define a notion of dimension for systems of word equations, in a way similar to linear algebra of (algebraic or analytic) geometry.

12.1 Independent systems of equations

This section is based on [21], though with a different construction (original one seems to have a problem with correctness).

Definition 12.1. Two systems of equations $\mathcal{E}$ and $\mathcal{E}'$ (perhaps infinite) are *equivalent* if: for each substitution s it is a solution of $\mathcal{E}$ if and only if it is a solution of $\mathcal{E}'$ (note that the number of variables can be infinite).

Example 12.1. Consider the following system E of equations:

$$e_i : xy^iz = zy^ix, \quad i = 1, 2, \ldots$$

The equations $e_1 : xyz = zyx$ and $e_2 : xy^2z = zy^2x$ are independent, since $x = a$, $y = b$, and $z = aba$ is a solution of the first one but not of the second one, and $x = a$, $y = b$, and $z = abba$ is a solution of the second one but not of the first one.

The system E is equivalent to its subsystem $E' = \{e_1, e_2\}$. For this, we can suppose by symmetry that $|x| > |z|$ in a solution of e_1. Therefore, $x = zw_1 = w_2z$ for some words w_1 and w_2.

From e_1 and e_2, we obtain:

$$w_1y = yw_2,$$
$$w_1y^2 = y^2w_2,$$

and, consequently,

$$yw_2y = w_1y^2 = y^2w_2,$$

i.e., $yw_2 = w_2y$, and similarly $w_1y = yw_1$. Since $|w_1| = |w_2|$, we have $w_1 = w_2$, and therefore, for $i \geq 2$,

$$xy^iz = zw_1y^iz = zy^iw_1z = zy^ix$$

as required.

We prove that every system of equations in a free semigroup over a finite set of variables has an equivalent finite subsystem. Note that this system may include constants (from a finite alphabet).

Theorem 12.2 (Ehrenfeucht's Conjecture)**.** *Every infinite system of word equations $\mathcal{E}$ in a finite number of variables over a finite alphabet has a finite subsystem $\mathcal{E}' \subseteq \mathcal{E}$ that is equivalent to $\mathcal{E}$.*

It is easy to see that the assumption on finite number of variables and on finite alphabet are essential (left as an exercise).

We will reduce this problem to a one on polynomials (with coefficients in $\mathbb{Z}$) and use Hilbert's theorem: The proof first encodes the word equations as equations between matrices, see Task 8. Those can be reduced to polynomials. Solutions of word equations correspond to ideals of polynomials, and for that we know that polynomials with integer coefficients have finitely generated ideals (we will simplify the presentation and avoid the name of ideal later on).

Theorem 12.3 (Hilbert's basis theorem)**.** *Let P_i, for $i > 1$, be polynomials in $\mathbb{Z}[\vec{X}]$. There exists a finite subset $P_1, P_2, \ldots, P_t$ of these polynomials such that every P_i can be expressed as a linear combination*

$$P_i = \sum_{j=1}^{t} Q_{ij} P_j,$$

where $Q_{ij} \in \mathbb{Z}[\vec{X}]$.

We will use it in the following form.

Theorem 12.4. *Let $\{P_i = 0 : i > 1\}$ be a system of polynomial equations, where $P_i \in \mathbb{Z}[\vec{X}]$. There exists an equivalent finite subsystem $\{P_i = 0 : i = 1, 2, \ldots, \ell\}$.*

For simplicity we will use two-letter alphabet, but this is not restrictive (exercise).

Now, recall the construction of Task 8: The mapping is defined as

$$\varphi(a) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \text{ and } \varphi(b) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} .$$

and it is extended to Σ^* as a homomorphism.

Theorem 12.5. *φ is an isomorphism between Σ^* with concatenation and matrices with natural coefficients and determinant 1 with multiplication.*

Now, for each variable X we introduce for integer variables $x_{11}, x_{12}, x_{21}, x_{22}$ with the intention that they represent a matrix

$$\varphi(X) = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$$

We rewrite the system of equations, by replacing every constant c with $\varphi(c)$ and X with $\varphi(X)$, i.e. $u = v$ is replaced with $\varphi(u) = \varphi(v)$, or, equivalently, with

$$\varphi(u) - \varphi(v) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Now, after computing the left-hand side, we get a polynomial (in several variables). So the satisfiability of the original system of equations is equivalent to the satisfiability of the system of polynomial equations in natural numbers. We now add new polynomial to ensure that each integer variable is non-negative (task, reduces to number theory) and apply Theorem 12.4. This yields a finite subsystem of polynomial equations and we consider the corresponding (finite) subsystem of word equations, which is equivalent to the input one.

Remark. Note that we proved a combinatorial result for semigroups, i.e. noncommutative structure with one operation by reducing it to a result for commutative rings, i.e. commutattive structure with two operations.

12.2 Defect Theorem

Definition 12.6. A finite set $A \subseteq \Sigma^*$ of words is a *code*, if every word $w \in \Sigma^*$ has at most one factorization, i.e.

$$\forall w \neq w' \in A \; wA^* \cap w'A^* = \emptyset \ .$$

Similarly, a finite $A \subseteq \Sigma^+$ is an ω-code, if the above holds for all $w \in \Sigma^{\mathbb{N}}$, that is,

$$\forall w \neq w' \in A \; wA^{\mathbb{N}} \cap w'A^{\mathbb{N}} = \emptyset \ .$$

Finally, $A \subseteq \Sigma^+$ is a prefix code, if

$$\forall w \neq w' \in A \; w \cap w'\Sigma^* = \emptyset \ .$$

It is easy to show that a prefix code is an ω-code, and an ω-code is a code (exercise).

Theorem 12.7 (Defect Theorem)**.** *If a set of words $A \subseteq \Sigma^*$ is a not a code then there is $B \subseteq \Sigma^*$ such that $|B| < |A|$ and $A \subseteq B^*$.*

The Theorem is in fact stronger (it restricts the form of B), but it requires some technical tools and definitions.

Possible applications: results similar and stronger than the periodicity Lemma:

Example 12.2. Suppose that $u, v \in \Sigma^*$ satisfy a nontrivial equation (that is a one which does not reduce to $\epsilon = \epsilon$ after removing of the same prefixes/suffixes from both of them). Then there is a word $w \in \Sigma^*$ such that $u, v \in w^*$.

For instance, this implies the Periodicity Lemma, with the equation being $uv = vu$. But it works for any other equation, say $uuvv = vvuu$.

12.2.1 Semigroups and bases

In the following we consider subgroups without neutral element, so in case of subsets of Σ^*: without ϵ.

For a subsemigroup $A \subseteq \Sigma^+$ its *base* is a minimal (in terms of inclusion) subset B such that $A = B^+ \cup$. It is easy to see that B consists exactly of those elements of A that cannot be represented as a concatenation of two other elements of A. Hence

$$B(S) = S \setminus S^2$$

In particular, the base is unique.

The *rank* of S is defined as

$$\mathrm{rank}(S) = |B(S)|.$$

A semigroup A is *free*, when it is isomorphic ot Σ^+ for some finite Σ.

A subsemigroup S of Σ^+ is free if its base $B(S)$ is a code (exercise).

S is free if and only if it satisfies the following (Schützenberger's) condition (more difficult exercise):

$$\forall u \in \Sigma^+ : Su \cap S \neq \emptyset \quad \text{and} \quad uS \cap S \neq \emptyset \implies u \in S. \tag{12.1}$$

A subsemigroup S of Σ^+ is said to be *right unitary*, if its base is a prefix code.

S is right unitary if and only if

$$\forall u \in S, v \in \Sigma^+ : uv \in S \implies v \in S. \tag{12.2}$$

A subsemigroup S of Σ^+ is called *ω-free*, if its base is an ω-code. It can be shown that S is an ω-free subsemigroup if and only if it is free and satisfies

$$\forall u \in S, s \in S^{\mathbb{N}}, w \in \Sigma^+ : uw \in S, ws \in S^{\mathbb{N}} \implies w \in S. \tag{12.3}$$

Those characterizations allow:

Theorem 12.8. *Any nonempty intersection of free (resp. right unitary) subsemigroups of Σ^+ is free (resp. right unitary).*

For any subset $A \subseteq \Sigma^+$, there exists the least free subsemigroup $F(A)$ containing A, there exists the least right unitary subsemigroup $P(A)$ containing A, and the least ω-free subsemigroup containing A, namely

$$F(A) = \{S \,:\, A \subseteq S \subseteq \Sigma^+, S \text{ is free}\},$$
$$P(A) = \{S \,:\, A \subseteq S \subseteq \Sigma^+, S \text{ is right unitary}\},$$
$$W(A) = \{S \,:\, A \subseteq S \subseteq \Sigma^+, S \text{ is } \omega\text{-free}\}.$$

These semigroups are called the *free hull*, the *prefix hull* and the ω-free hull of A, respectively.

In the following we give an "algorithm" which transforms a given set (generating a semigroup) into a base of free hull/prefix hull/ω-free hull. The algorithm is greedy and computes the closure according to conditions (12.1)–(12.3) (and removes the non-needed generators).

Let A be a finite subset of Σ^+, define the sets of words that violate conditions (12.1)–(12.3):

$$C_f(A) = \{(u, v) \in A \times A \,:\, u \neq v, uA^* \cap vA^* \neq \emptyset\},$$
$$C_p(A) = \{(u, v) \in A \times A \,:\, u \cap v\Sigma^+ \neq \emptyset \text{ or } v \cap u\Sigma^+ \neq \emptyset\},$$
$$C_\omega(A) = \{(u, v) \in A \times A \,:\, u \neq v, uA^{\mathbb{N}} \cap vA^{\mathbb{N}} \neq \emptyset\}.$$

By (12.1)–(12.3): A is a prefix code (resp. a code or an ω-code) if and only if $C_p(A) = \emptyset$ (resp. $C_f(A) = \emptyset$ or $C_\omega(A) = \emptyset$). Notice that

$$C_f(A) \subseteq C_\omega(A) \subseteq C_p(A)$$

for all finite A.

Let $\bullet \in \{p, f, \omega\}$.

Algorithm 13 $P_\bullet$ computing a base of appropriate hull of $A \subseteq \Sigma^+$

```
while C•(A) ≠ ∅ do
    let (u, v) ∈ C$(A_j), where u = vw
    if w ∈ A then
        A ← A \ v
    else
        A ← (A \ u) ∪ w
return A
```

It is easy to see that the sum of lengths of elements in A only decreases, so the procedure will terminate.

We want to show that the elements added to A by $P_\bullet$ are indeed in appropriate hull.

Lemma 12.9. *Let $(u, v) \in C_\bullet(A_j)$, where $u = vw$. Then w is in appropriate hull of A.*

Proof. For $\bullet = p$ observe that by (12.2), when $vw, v \in P(A)$ then also $w \in P(A)$.

For $\bullet = f$ since $(u, v) \in C_f(A)$, we have words $u', v' \in A^*$ such that

$$uu', vv' \in A^+ \subseteq P(A)$$

Then

$$wu' = v', vw = u \in A^+ \subseteq P(A)$$

and by (12.1), also $w \in P(A)$.

The argument for $\bullet = \omega$ is as in the second case, with condition (12.3). □

Hence the appropriate hull of A remain the same during the whole procedure. Since we terminate in a base, this yields that the procedure returns the base of appropriate hull.

From the procedures $P_\bullet$ it follows that the base elements of $P(A)$, $W(A)$, and $F(A)$ are suffixes of words of A. In the case $\bullet = f$, the reverses of the words in A can be considered so that the elements of $F(A)$ are also prefixes of words of A. Note that the base is defined uniquely!

Theorem 12.10. *Let $A \subseteq \Sigma^+$ be a finite set. Then*

1. $\mathrm{rank}(F(A)) \leq card(A)$ *with equality if and only if A is a code.*
2. $\mathrm{rank}(P(A)) \leq card(A)$.
3. $\mathrm{rank}(W(A)) \leq card(A)$ *with equality if and only if A is an ω-code.*

Proof. The inequality is clear, as the procedure can only decrease the size of A. We show that if A is not a code then at some point the procedure will remove an element.

So suppose that A is not a code, $u, v \in A_j$ with $us = vt$ and $u = vw$ for some $s, t \in A^*$ and $w \in \Sigma^+$. Clearly, if (u, v) is not chosen by the procedure, then exactly the same equation still holds (note that new A, call it A', satisfies $A^+ \subseteq A'^+$). If c was chosen in a different pair then the equation still holds and is still no-trivial (it has different first elements). If u was chosen in a different pair, but split into $v'w'$ where $v' \neq v$ then again the equation has different first elements. If $v' = v$ then this is in fact the same pair (u, v). If w is removed then we are done. If not, then consider that the old equation

$$us = vt \implies vws = vt \implies ws = t$$

yet t cannot begin with element w it is not in A and the new w are always after v).

The argument is the same in the third case. □

Exercises

Task 65- Write a polynomial with integer coefficients (in several variables $x, x_1, \ldots$) such that for each natural $n \geq 0$ there is an integer solution $(n, n_1, \ldots)$ and for each solution $(n, n_1, \ldots)$ we have $n \geq 0$.

This is used to guarantee that the solutions in the proof of Ehrenfeucht's conjecture are indeed positive.

Hint: Lagrange's four-square theorem.

Task 66 Show that a semigroup $S \subseteq \Sigma^*$ is right-unitary (its base is a prefix code) if and only if

$$\forall u \in S, v \in \Sigma^+ : uv \in S \implies v \in S.$$

Do not use the algorithm for computing the base (as its uses this condition).

Task 67(2 points) Show that a semigroup $S \subseteq \Sigma^*$ is free (its base is a code) if and only if

$$\forall u \in \Sigma^+ : Su \cap S \neq \emptyset \quad \text{and} \quad uS \cap S \neq \emptyset \implies u \in S.$$

Do not use the algorithm for computing the base (as its uses this condition).

Task 68 Show that a base $B(S)$ of a semigroup S is a code if and only if S isomorphic to Γ^* for some finite Γ.

Task 69 Give a polynomial-time algorithm for verifying, whether (finite) $A \subseteq \Sigma^*$ is a code.

Hint: Automata; fixing two words that should give a counterexample may help.

Task 70 Give a polynomial-time algorithm for verifying, whether (finite) $A \subseteq \Sigma^*$ is an ω-code.

Hint: Automata over infinite strings are one option; using some periodicity is another. Or you can use automata over finite words directly.

Task 71 Show that the base of smallest semigroup containing A, so $B(F(A))$, consists of suffixes of some words from A. Show that they are in fact borders of words from A.

Hint: Easy for suffixes. For borders: use a trick to show that they are prefixes and then the fact that the base is unique.

Chapter 13

Equations without constants and related topics

13.1 Lyndon-Schützenberge Theorem

The presentation is based on [15].

The main goal is to prove the following Theorem, originally stated for the free groups [38], so in a slightly stronger form.

Theorem 13.1 (Lyndon-Schützenberge)**.** *Let $m, n, k \geq 2$ be natural numbers. Then all solutions of the equation*

$$X^m Y^n = Z^k$$

are periodic, i.e. for each solution s there is a word w_s such that $s(X), s(Y), s(Z) \in w_s^*$ for each solution s.

We will show the theorem in a less general case of word equations.

Definition 13.2. A word w is *bordered*, if there exists $\epsilon \neq v \neq w$ such that $w = vw'' = w'v$ for some w', w''.

Lemma 13.3. *A word w is bordered if and only if there exists v, w' with $\epsilon \neq v \neq w$ such that $w = vw'v$.*

A word w is bordered if and only if there exists a word u with $|u| < |w|$ and a natural number k such that w is a subword of u^k.

A simple proof is left as an exercise.

Lemma 13.4. *If w is a conjugate to a^m then there are words p, q such that $a = pq$ and $w = (qp)^m$.*

Definition 13.5. Let $\leq$ denote a linear-order on letters, which is extended to strings as a lexicographical order.

A primitive word w is a *Lyndon word* (according to the order $\leq$), if it is the lexicographically minimal one among the cyclic shifts of w.

Lemma 13.6. *Lyndon word is not bordered.*

A simple proof is left as an exercise.

Lemma 13.7. *Let u, v be primitive words and w a word such that $|w| \leq |v|$. If they satisfy an equation*

$$u^m = v^k w$$

for some $k, m \geq 2$ then either

- *$u = v$ and $w \in \{\epsilon, v\}$* or

- *$m = k = 2$ and there exist p, q such that their primitive roots are different and there are natural s, ℓ such that $u = (pq)^{s+\ell}p(pq)^{\ell}$, $v = (pq)^{s+\ell}p$ and $w = (qp)^{\ell}(pq)^{\ell}$.*

Proof. If $u = v$ then clearly $w \in \{\epsilon, v\}$. Moreover, if $w \in \{\epsilon, v\}$ then u^m is equal to v^k or v^{k+1} and so has periods u, v and thus from periodicity Lemma we obtain that v, u are powers of the same primitive words, so the assumption yields that $v = u$.

So it is left to consider the case in which $w \notin \{\epsilon, v\}$. First, observe that both u and v are periods of v^k, so if $|u| + |v| \leq \left|v^k\right|$ then they are both the periods of it which, together with the primitivity of u, v, leads to a conclusion that $u = v$. Thus we can assume that

$$|u| + |v| > \left|v^k\right| \ .$$

Then on one hand we have

$$\begin{aligned} m|u| &= |u^m| \\ &= \left|v^k w\right| \\ &\leq (k+1)|v| \end{aligned}$$

and on the other

$$\begin{aligned} |u| &> \left|v^k\right| - |v| \\ &= (k-1)|v| \end{aligned}$$

Multiplying the second inequality by m and comparing them we obtain

$$(k+1)|v| > m(k-1)|v|$$

After simplification

$$2 > (m-1)(k-1) \ ,$$

which can hold only when $m = k = 2$.

Thus we arrive at the desired equation

$$u^2 = v^2 w \ .$$

Since $u \neq v$ we have that $u = vv'$ for some $v' \sqsubseteq v$, let also $v = v'v''$. Then

$$u = v'v''v'$$

and on the other hand, from the second copy of u

$$u = v''w$$

So $|w| = 2\,|v'|$ and so looking at the two expressions for w we get that

$$w = w'v'$$

for some $w' \sqsubseteq w$. Looking again at u we finally obtain

$$u = v'v''v' = v''w'v'$$

After removing the suffix v' we see that

$$v'v'' = v''w'$$

Now from Lemma 8.2 this means that there exist p, q such that $v' = (pq)^{\ell}$, $w' = (qp)^{\ell}$ and $v'' = (pq)^s p$ for some natural numbers ℓ, s. This yields the required form of v, u, w. Proving that the primitive roots of p, q are different, that $u \neq v$ and $w \notin \{\epsilon, v\}$ is left as an easy exercise. □

Lemma 13.8. *Let u, v be primitive words and $v' \sqsubset v$ be a prefix of v. If they satisfy an equation*

$$u^m = v^k v'$$

for some $k, m \geq 2$ then $u = v$ and $v' = \epsilon$.

A similar statement holds when v' is a suffix and the equation is

$$u^m = v' v^k$$

Proof.
- If it does not hold that $m = k = 2$ then this follows directly from Lemma 13.7.
- If $u = v$ (and $k = m = 2$) then equation $u^2 = v^2 v'$ implies that $v' = \epsilon$.
- If $k = m = 2$ and $u \neq v$ then from Lemma 13.7 we obtain the form of v, u, v' and it is easy to verify that this form implies that v' is not a prefix of v, contradiction.

Thus $u = v$ and $v' = \epsilon$. □

Now we are ready to move to the main proof

proof of Theorem 13.1. Let a, b, c be words satisfying an equation

$$a^m b^n = c^k$$

for some $m, n, k \geq 2$. We intend to show that there exist w such that $a, b, c \in w^*$.

If any of those words is not primitive then we can replace it with its primitive root (and increase the power appropriately).

If $\epsilon \in \{a, b, c\}$ then the claim is clear: if $c = \epsilon$ then also $a = b = c = \epsilon$ and we are done. If, say, $a = \epsilon$ then we get $b^n = c^k$ and from Lemma 13.8 it follows that $b = c$; the proof is the same for $b = \epsilon$.

If $a = b$ then again we use Lemma 13.8 to conclude that $a = b = c$. If $a = c$ then we can cancel out powers of a and obtain $b^n = c^{k-m}$: if $k - m \leq 1$ then the claim is clear. If $k - m \geq 2$ then again we use Lemma 13.8 to conclude that $a = b = c$. The proof is analogous for $b = c$.

Thus we are left with the main case, when a, b, c are pairwise different and non-empty. Let $a^m = c^s c'$ and $b^n = c'' c^{k-s-1}$ for some c', c'' such that $c'c'' = c$. If $s \geq 2$ or $k - s - 1 \geq 2$ then we can use Lemma 13.8 and conclude that $a = c$ or $b = c$, which ends the argument. Thus assume $s \leq 1$ and $k - s - 1 \leq 1$, thus $k \leq 3$ (and by the assumption $k \geq 2$). Moreover, if $k = 3$ then from $k - s - 1 \leq 1$ we obtain that $s = 1$. Let us consider the two cases: $k = 3, s = 1$ and $k = 2$ separately.

First, let $k = 3$ and $s = 1$. Since $a^m = cc'$ and $b^n = c''c$ then in particular $|a|, |b| < |c|$. Look at

$$c^2 = c'c''c'c'' = a^m c'' = c' b^n$$

note that the overlap of a^m and b^n in this representation is $c''c'$, i.e. it is of length $|c|$. Now, each conjugate of c occurs in cc and so it appears either in a^m or in b^n. But then, from Lemma 13.3, we obtain that each conjugate of c is bordered. But this cannot be, as c is primitive and so some of its conjugates is a Lyndon word and such a word cannot be bordered, see Lemma 13.6.

So let us investigate the case $k = 2$. Thus we are looking at the equation

$$a^m b^n = c^2$$

Without loss of generality we can take a, b, c such that $|c|$ is the smallest possible.

If $|a^n| = |b^m|$ then $a^n = b^m$ and so from Lemma 13.6 we get that $a = b$ and we are done. So consider the case when $|a^n| > |b^m|$, which implies $|a^n| > |c|$ (the other case is symmetric, or we can make it by reversing both sides of the equation). Then $a^m = cc'$ for some $c' \sqsubseteq c$, and $c'c = (c')^2 b^n$. Thus a^m and $(c')^2 b^n$ are conjugate. From Lemma 13.4 we obtain that there are p, q such that $(qp)^m = (c')^2 b^n$. Note that this is also an equation of the type we are investigating. If $m = 2$ then we have an equation

$$(c')^2 b^n = (qp)^2$$

and $|qp| = |pq| = |a| < |c|$ and this is a contradiction with the choice of c.

So we are left with the case $m \geq 3$, but we already proved that this implies that b, c', qp are all powers of the same word. But as b is a primitive word, they are all powers of b. Also, qp is conjugate to $pq = a$ and it also is primitive, so we get $b = qp$ and by assumption $b = c''$. Also $c' \in (qp)^*$ and so $c = c'c'' \in (qp)^*$, contradiction, as it is primitive. □

13.2 Assigning numerical values technique

This Section is based on [55].

In this section we say that a solution s is *periodic* if there is a word w such that for each variable X we have $s(X) \in w^*$.

Consider a system of the form

$$X_0^k = X_1^k X_2^k \cdots X_n^k \text{ for } k = k_1, k_2, k_3 \tag{13.1}$$

with $0 < k_1 < k_2 < k_3$ being natural numbers.

Theorem 13.9. *A system* (13.1) *has only trivial solutions.*

Note that it is easy to define systems of this form with only two equations and they have nonperiodic solutions.

We do not specify the alphabet, yet if (13.1) has a nonperiodic solution, it has one over the binary alphabet.

Lemma 13.10. *If the system* (13.1) *has a nonperiodic solution then it has a nonperiodic solution over a binary alphabet.*

A simple proof is left as an exercise.

In the following, we use Γ rather than Σ to denote the finite alphabet and identify Γ with a subset of $\mathbb{R}$. Given a word $w = a_1 a_2 \dots a_\ell$ by $\sum(w)$ we denote $a_1 + a_2 + \ldots + a_\ell$. Also, by $\mathrm{psw}_q(w)$ (partial sum word) we denote a sequence $q + a_1, q + a_1 + a_2, \ldots, q + \sum(w)$ and think of if as a piecewise linear function that goes through the points $(0, q), (1, q + a_1), (2, q + a_1 + a_2), \ldots$. If no lower index is used then by default it is 0, i.e. $\mathrm{psw}(w) = \mathrm{psw}_0(w)$ The idea of the lower index is that $\mathrm{psw}(ww') = \mathrm{psw}(w)\,\mathrm{psw}_{\sum(w)}(w')$, the piecewise linear functions are concatenated in a natural way.

We sometimes normalise the solution: w is called a 0-word if $\sum(w) = 0$. We usually assume that $s(X_0)$ is a 0-word. This can be always achieved.

Lemma 13.11. *Given a solution s to a system of word equations* (13.1) *by changing the alphabet Γ we obtain a different solution such that $s(X_0)$ is a 0-word.*

Again, a simple proof is left as an exercise.

As a first step towards the main proof we show that if a solution is length-minimal among the non-periodic solutions then not all of $s(X_0)$, $s(X_1)$, ... are 0-words.

Lemma 13.12. *Let s be a length-minimal among the nonperiodic solutionso of the system of equations* (13.1)*. Then not all among $s(X_0)$, $s(X_1)$,..., $s(X_n)$ are 0-words.*

Proof. Suppose not. Consider the factorisations of $s(X_0^{k_1})$, $s(X_0^{k_2})$ and $s(X_0^{k_3})$ into 0-words that cannot be further factorised into 0-words. Let $V = \{v_1, \ldots, v_\ell\}$ be the set of all obtained such words. Then $s(X_0)$ is a concatenation of some words from this set. We claim that this is the same for each $s(X_i)$, which is claimed by induction: suppose that each of $s(X_0)$, $s(X_1)$, ..., $s(X_i)$ factorises into words from V. Look at the first occurrence of $s(X_{i+1})$ in $s(X_1^k X_2^k \cdots)$. The corresponding word on the left hand side factorizes into words form V. Also the prefix factorizes into words from V. So also $s(X_{i+1})$ factorizes into them, as it is a 0-word.

Observe that $s(X_0)$ is not a power of one of elements from V: in such a case it would be a periodic solution and by easy induction also all $s(X_i)$ would be powers of the same word. Thus some of used words from V is of length graeater than 1. Make a new solution over the alphabet V, which is equal to the factorisation of each $s(X_i)$ into elements of V. This is a shorter solution and it is nonperiodic. □

We now are ready to prove Theorem 13.9

proof of Theorem 13.9. Take the shortest non-periodic solution of the system of word equations (13.1). By Lemma 13.11 we can assume that $\sum(s(X_0)) = 0$ and by Lemma 13.12 for some other variable X_i we have that $\sum(s(X_i)) \neq 0$.

Divide the variables into two groups: good and bad. A variable X_i is bad, if $\sum(s(X_i)) = 0 = \sum(s(X_1X_2\cdots X_{i-1}))$ and it is good otherwise. Fix a number a and let us count, how many times it occurs in $\mathrm{psw}(s(X_0)^k)$. If a occurs m_0 times in $\mathrm{psw}(s(X_0))$ then it occurs km_0 times in $\mathrm{psw}(s(X_0)^k)$, as $s(X_0)$ is a 0-word and so each copy of $\mathrm{psw}(s(X_0))$ begins with an offset 0.

The same argument applies to each of the bad variables (with a different value of m, of course). Let us now fix a as the maximal value of that occurs in $\mathrm{psw}(s(X_1)^k s(X_2)^k \cdots s(X_n)^k)$ (over all $k \in \{k_1, k_2, k_3\}$) and comes from some good variable. For this fixed a it occurs km times on the left hand side and has km' occurrences that come from bad variables on the right-hand side. Thus it has $k(m - m') > 0$ occurrences that come from good variables. To obtain a contradiction we show that for each good variable X_i it has at least as many occurrences that come from $s(X_i)^{k_1}$ as from $s(X_i)^{k_2}$, which cannot be, as they should all sum (over all good variables) to, respectively, $k_1(m - m') < k_2(m - m')$.

For this fixed i let $s = \sum(s(X_1 \cdots X_{i-1}))$, then $s \neq 0$ or $\sum(s(X_i)) \neq 0$, as this is a good variable. Define $w_i = \mathrm{psw}_{k_i s}(s(X_i^{k_i}))$. This is the part of the right-hand side that corresponds to the input of X_i in the equation for k_i.

We consider some cases. If $\sum(s(X_i)) = 0$ then w_1, w_2, w_3 are all 0-words and they begin at height $k_1 s, k_2 s, k_3 s$, which are different. Thus w_2 cannot have any occurrences of a, as w_1 or w_3 is above it and a is at least the maximal value in them. In particular a has not more occurrences in w_2 then in w_1.

If $s = 0$ then $\sum(s(X_i)) \neq 0$ and so all $w_1, \ldots, w_3$ begin at the same point (0) and they all either increase or decrease with each copy of $s(X)_i$. If they increase then a has no occurrence in w_2, as the last copy of $s(X_i)$ in w_3 is higher. If they decrease then the maximal value is attained in the first copy of $s(X_i)$ and it is identical in w_1 and w_2.

The other cases are shown in a similar fashion (sometimes we need to consider the sign of $s + \sum(s(X_i))$). □

Exercises

Task 72 Show that a Lyndon word is not bordered.

Task 73 Give an algorithm that, given a word w and an order on the alphabet, computes the Lyndon word conjugate to w.

Task 74 Show that a word w is bordered if and only if it is a subword of u^k for some word u, where $|u| < |w|$, and some $k \geq 2$.

Task 75 Show that if the system (13.1) has a nonperiodic solution then it has a nonperiodic solution over a binary alphabet.

Task 76 Show that given a solution s to a system of word equations (13.1) by changing the alphabet $\Gamma \subseteq \mathbb{R}$ we obtain a different solution such that $s(X_0)$ is a 0-word.

Task 77 Show that the set of minimal 0-sum words are a prefix code.

Task 78 Show the remaining cases of the proof for Theorem 13.9 not shown in the lecture. The claim is that the number of occurrences for k_2 is not more than the number of occurrences for k_1.

We have covered the case when $\sum(s(X_i)) = 0$ and $\sum(s(X_1 \cdots X_{i-1})) = 0$.

Chapter 14

Parametrasability of solutions

This section is based on [8].

Hmelevskii showed that a solution of 3-variable constant-free equations are finitely parametrisable and that this does not hold for 4-variable equations (without constants). We show a modern, simplified proof of this result.

To be more precise we show that solutions of an equation

$$xyz = zvx$$

is not finitely parametrisable.

14.1 Parametric Equations

We define word parameters and numerical parameters as parameters whose values are words over the alphabet Σ, and nonnegative integers, respectively. Let Γ be a new alphabet. A parametric word over Γ is defined inductively as follows:

1. Every letter in Γ is a parametric word.

2. If Ψ is a parametric word, and k is a numerical parameter and c is a natural constant, then Ψ^{ck} is a parametric word.

3. If Ψ_1 and Ψ_2 are parametric words, then also $\Psi_1\Psi_2$ is a parametric word, where $\Psi_1\Psi_2$ is obtained by concatenating Ψ_1 and Ψ_2.

Note that a parameter can in principle be copied several times.

Given a parametric word Ψ, every assignment φ of values in Σ^* to the letters of Γ, and of values in $\mathbb{N}$ to the numerical parameters, defines a unique word in Σ^*, called the value of Ψ under φ, and is denoted by $\varphi(\Psi)$.

An equation over Σ and with n unknowns is parametrisable if there exists a finite number of n-tuples of parametric words $F_1, \ldots, F_k$ over an alphabet Γ such that the solution sets coincides with the set of values of $F_1, ldots, F_k$ (over all assignments φ and of numerical parameters).

Lemma 14.1. *Let $u = v$ be a constant-free equation with n unknowns over the alphabet Σ with $|\Sigma| \geq 2$ and $(T_1, \ldots, T_n)$ be a parametric solution. Let $(V_1, \ldots, V_n) \in (\Delta^*)^n$ be the n-tuple obtained from $(T_1, \ldots, T_n)$ by assigning fixed values to all numerical parameters. Then, $(V_1, \ldots, V_n)$ is a solution of the equation $u = v$ over Δ.*

Easy prof is left as an exercise.

14.2 Fibonacci Words and Parametric Equations

Define the Fibonacci as (note, there is an offset in indices by 1 with respect to usual definition, this was left to be consistent in the presentation)

$$\begin{aligned} F_0 &= a, \\ F_1 &= ab, \\ F_n &= F_{n-1}F_{n-2}, \quad \text{for all } n \geq 2. \end{aligned}$$

$$\mathrm{suf}_2(F_n) = \begin{cases} ab, & \text{if } n \text{ is odd,} \\ ba, & \text{if } n \text{ is even,} \end{cases}$$

for all $n \geq 1$.

Lemma 14.2. *Fibonacci words are 4-th power free, i.e. they do not contain a subword of a form w^4.*

Proof left as an exercise.

Consider the words G_n obtained from F_n by removing the last two letters.

$$G_n = F_n\Sigma^{-2}$$

14.3 Result

Theorem 14.3. *The set of solutions of the equation $xyz = zvx$ over an alphabet with at least two distinct letters is not parametrisable.*

Proof. Assume for the sake of contradiction that $xyz = zvx$ is parametrisable.

Then we have a finite number of 4-tuples of parametric words (T_1, T_2, T_3, T_4), such that all valuation of them give exactly the set of solutions of $xyz = zvx$. In particular, each T_i uses a finite number of different letters from Γ and parameters. Let (V_1, V_2, V_3, V_4) be the 4-tuple obtained from (T_1, T_2, T_3, T_4) by assigning fixed values to each numerical parameter, which are solutions of the equations over Γ, i.e. $V_1V_2V_3 = V_3V_4V_1$. We can use the notions of prefix, suffix, alphabet etc. as before.

By symmetry, we can assume that $|V_1| \geq |V_3|$.

We prove now that for such (V_1, V_2, V_3, V_4) we have

$$V_2 = V_4 \quad \text{or} \quad \mathrm{Alph}(V_1V_3) \subseteq \mathrm{Alph}(V_2V_4),$$

where $\mathrm{Alph}(V_2V_4)$ denotes the set of letters used in V_2V_4. There are three cases, depending on length of V_1

If $|V_1| = |V_3|$ then $V_1 = V_3$ and $V_2 = V_4$, as claimed.

If $|V_3| < |V_1| \leq |V_3V_4| = |V_2V_3|$. Then $V_1 = V_3P$ and $V_1 = QV_3$ for some $P, W \in \Delta^+$ are a of V_4 and suffix of V_2, respectively. Thus, $QV_3 = V_3P$, which implies

$$Q = SW, \quad P = WS, \quad V_3 = (SW)^iS, \quad V_1 = (SW)^{i+1}S,$$

Then $\alpha \in \mathrm{Alph}(V_1V_3) = \mathrm{Alph}(SW) \subseteq \mathrm{Alph}(V_2V_4)$.

If $|V_1| > |V_3V_4|$. We can write $V_1 = V_3V_4P$ and $V_1 = QV_2V_3$, with $P, Q \in \Delta^+$. Substituting these relations into the identity $V_1V_2V_3 = V_3V_4V_1$, we obtain that $P = Q$. So, in this case (V_1, V_2, V_3, V_4) is a solution of the equation $xyz = zvx$ over Δ if and only if

$$PV_2V_3 = V_3V_4P \quad \text{and} \quad V_1 = PV_2V_3,$$

i.e., (P, V_2, V_3, V_4) is a solution of the equation $xyz = zvx$ over Δ and $V_1 = PV_2V_3$. If $V_1 = P$, i.e. $V_2V_3 = 1$, then $V_2 = V_3 = V_4 = \epsilon$, in particular $V_2 = V_4$. Otherwise (i.e. $V_1 \neq P$), we reduce the solution (V_1, V_2, V_3, V_4) to (P, V_2, V_3, V_4), with $|P| < |V_1|$, and $V_1 = PV_2V_3$. We can iterate this step

until $|P| \leq |V_3V_4|$, which means that we can apply one of the cases above. So, for (P, V_2, V_3, V_4) we have

$$V_2 = V_4 \quad \text{or} \quad \text{Alph}(PV_3) \subseteq \text{Alph}(V_2V_4).$$

But, since $V_1 = PV_2V_3$, this implies that for the solution (V_1, V_2, V_3, V_4) we have

$$V_2 = V_4 \quad \text{or} \quad \text{Alph}(V_1V_3) \subseteq \text{Alph}(V_2V_4).$$

Observe that for the words G_k the tuple

$$(G_k, ab, G_{k-1}, ba)$$

is a solution for the equation $xyz = zvx$ over Σ for every odd k. (Exercise)

Consider now an assignment τ and a parametric solution (T_1, T_2, T_3, T_4) such that

$$(\tau(T_1), \tau(T_2), \tau(T_3), \tau(T_4)) = (G_k, ab, G_{k-1}, ba)$$

for some odd index k. We prove now that the length of $\tau(T_1)$ is bounded by a constant.

First, since every G_k is a prefix of the Fibonacci word, which is 4-free, and $\tau(T_1) = G_k$ for some odd k, we must have that every power of a factor in $\tau(T_1)$ is less than 4.

Second, consider the 4-tuple (V_1, V_2, V_3, V_4) over Δ obtained from the parametric solution (T_1, T_2, T_3, T_4) by substituting every numerical parameter with its value $\tau(\alpha)$. Since $\tau(T_i) = \tau(V_i)$ for every $1 \leq i \leq 4$, we obtain the following relations:

$$G_k = \tau(V_1), \quad G_{k-1} = \tau(V_3), \quad ab = \tau(V_2), \quad ba = \tau(V_4). \tag{14.1}$$

Notice now that the values of formulas V_2 and V_4 under the assignment τ must be ab and ba, respectively, so $V_2 \neq V_4$. Thus, from (14.1) for any $\alpha \in \text{Alph}(V_2V_4)$ we have $|\tau(\alpha)| \leq 2$. But $\text{Alph}(V_1V_3) \subseteq \text{Alph}(V_2V_4)$. and so the same claim holds also for $\alpha \in \text{Alph}(V_1V_3)$. Thus $|\tau(V_1)|$ and $|\tau(V_3)|$ are both bounded by a constant.

So, what we obtained is that for any numerical parameter which appears in T_1, $\tau(\alpha) < 4$, and for any word parameter α which appears in T_1, $|\tau(\alpha)| \leq 2$. Consequently, $|\tau(T_1)|$ is bounded by some positive constant, i.e., we cannot generate arbitrarily large solutions (G_k, ab, G_{k-1}, ba), with k odd. But this is a contradiction since words G_k can be arbitrarily large. □

Exercises

Task 79 Let $u = v$ be a constant-free equation with n unknowns over the alphabet Σ with $|\Sigma| \geq 2$ and $(T_1, \ldots, T_n)$ be a parametric solution. Let $(V_1, \ldots, V_n) \in (\Delta^*)^n$ be the n-tuple obtained from $(T_1, \ldots, T_n)$ by assigning fixed values to all numerical parameters. Show that $(V_1, \ldots, V_n)$ is a solution of the equation $u = v$ over Δ.

Task 80 Show that

$$G_kabG_{k-1} = G_{k-1}baG_k$$

For odd k.

To this end show that

$$G_k = F_{k-1}G_{k-2}$$
$$G_k = F_{k-2}G_{k-1}$$

The first follows from definition and the second follows by easy inductive proof.

Task 81 Consider a parametric word of length n (we count each occurrence of a natural parameter as length 1). Upper-bound the maximal length of a word that can be obtained from it by substituting the letters with word over Σ and natural parameters with values, assuming that

- the obtained word is 4-th power free (i.e. does not have a subsword of a form w^4)
- we can substitute for each letter in Γ a word of length at most 2.

Note, we *can* stack exponents, say $((a^i)^j)^k$ is legal (and has length 4 as a parametric word).

Chapter 15

Infinite Alphabets

We now consider word equations over infinite alphabets. This more or less trivially reduces to the finite alphabet case when only the equations are allowed; things gets different when we allow regular constraints, though one needs an appropriate definition of "regular". This can be formalised using notions of symbolic/parametric/register automata.

15.1 Models of automata over infinite alphabets

We do not assume that the alphabet Σ is finite.

Satisfiability of equations over such an alphabet trivially reduce to the case of finite alphabets: if there is a solution, then there is one over the alphabet of letters present in the equation (and perhaps one more letter).

The situation changes, when we allow "regular constraints," but we need to define, what is a regular constraint.

There is no single canonical model of regular languages in this case.

15.1.1 Symbolic automata

We assume that the (infinite) alphabet comes with some logic and we label the transition with a formula ("guard") with one free variable. In most cases we assume that the formula is from some restricted fragment (say, no quantifiers or no alternation of quantifiers, perhaps some more restriction on the form, like it should be in some normal form). As a least requirement, we should be able to decide for a symbol a, whether $\varphi(a)$ holds and whether there is at all a such that $\varphi(a)$ holds.

By $\mathcal{T}(x)$ we will denote the set of allowed formulas. Then the transition function should satisfy

$$\delta \subseteq Q \times \mathcal{T}(x) \times Q$$

a we can move in the automaton from q to q' using transition $\varphi(x)$ by a symbol a when $\varphi(a)$ holds.

Otherwise, the definition of the automaton and the accepted language is the same.

As usual we denote the language recognized by the automaton by $L(\mathcal{A})$.

15.1.2 Symbolic automata

In parametric automaton we allow the automaton to guess some values before the start of the computation and use them later on (those are the parameters). Hence $\mathcal{T}(\vec{y}, x)$ has $p+1$ free variables. For a fixed sequence of parameters instantiations (by $\vec{p}$) we can move in the automaton from q to q' using transition $\varphi(\vec{y}, x)$ by a symbol a when $\varphi(\vec{p}, a)$ holds.

We denote the resulting language by $L_{\vec{p}}(\mathcal{A})$ and finally

$$L(\mathcal{A}) = \bigcup_{\vec{p}} L_{\vec{p}}(\mathcal{A})$$

For instance: such an automaton can recognize language of the form

$$\bigcup_{a\in\Sigma} a^*$$

it guesses a at the beginning and the tests each of the following letter for an equality with a.

If T includes the (linear) order relation It can recognize a language of words in which the maximum occurs twice: it guesses the maximum and then counts that it occurs twice and that nothing larger than the guessed elements occurs.

More details are given in later Lemma 15.6.

15.1.3 Register automata

The register automaton has k registers, in which it can store some of the values read during the computation; it can also update those values, but only to the letters just read.

Note that in most cases when one talks about register automata, we assume that the only allow tests for equality, i.e. the logic is restricted to Boolean combinations of equality relation (on registers and the read value).

Formally, a confiuration on an autmaton is $(q, \vec{r})$, where each $r_i \in \Sigma \cup \bot$, where $\bot$ represents the register whose value was not set yet (they are not initilised). Then the transition function satisfies

$$(p, \varphi(\vec{y}, x)q, \vec{b}) \subseteq Q \times \mathcal{T}(\vec{y}, \vec{x}) \times Q \times \{0, \ldots, k\}^k$$

where $\vec{b}$ defines, whether we should update the register

Then a transition from $(p, \vec{r})$ to $q, \vec{r'}$ by transition $(q, \varphi(\vec{y}, x)\, vecb)$ and letter a is possible, when $\varphi(\vec{r}, a)$ holds and

$$r'_i = \begin{cases} r_{b_i} & \text{when } b_i > 0 \\ a & \text{when } b_i = 0 \end{cases}$$

To simplify the definition, we assume that register automaton does not compare letters to $\bot$ and does not introduce $\bot$ (such actions can be simulated in the state: it can store the information, which registers are in fact empty).

We think that the number of registers is small (1, 2, ..., in most cases can be treated as a constant).

For instance, when the logic allows the usage or order, we can verify that the strict is strictly (or weakly) increasing: we test that the read letter is larger than the register and update the register to the read letter.

15.2 Word equations with symbolic automata

Here (and in case of register automata) we assume that the theory does not allow quantification.

Consider a set Φ of all atoms in all guards in the regular constraints together with the set of formulas $\{x = c\}$ over all letters $c \in \Sigma$ that appear in all equations; and the negations of both types of formulas.

The type $\Delta(a)$ of $a \in \Sigma$ is the set of formulas in Φ satisfied by a, i.e. $\{\varphi \in \Phi : \varphi(a) \text{ holds}\}$. Clearly there are at most exponentially many different types. A type t is realizable, when $t = \Delta_\pi(a)$ and it is realized by a.

If the constraints are satisfiable then they are satisfiable over a subset $\Sigma_0 \subseteq_{\text{fin}} \Sigma$, where Σ_0 is created by taking (arbitrarily) one element of a realizable type. Note that for each constant c in the equations there is a formula "$x = c$" in Φ, in particular $\Delta(c)$ is realizable (only by c) and so $c \in \Sigma_0$.

Lemma 15.1. *Given a system of constraints let $\Sigma_0 \subseteq \Sigma$ be obtained by choosing (arbitrarily) for each realizable type a single element of this type. Then the set of constraints is satisfiable over Σ if and only if they are satisfiable over Σ_0. To be more precise, there is a letter-to-letter homomorphism $\psi : \Sigma^* \to \Sigma_0^*$ such that if s is a solution of a system of constraints then $\psi \circ s$ is also a solution.*

Proof. If the constraints are satisfiable over Σ_0 then they are clearly satisfiable over Σ (a larger set), as the same assignment works.

If the constraints are satisfiable for Σ, then we change the assignment. For shortness of notation, for a type t by a_t we denote the chosen letter of this type in Σ_0, i.e. $a_t \in \Sigma_0, \Delta(a_t) = t$. Given an assignment satisfying all constraints we replace each symbol a with $a_{\Delta(a)}$. We claim that such assignment still satisfies all constraints.

For the regular constraint, suppose that $s(X) \in L(\mathcal{A})$, let $s'(X)$ be the assignment value after the replacement. Then $s(X) \in L(\mathcal{A})$: the corresponding letters of $s(X)$ and $s'(X)$ are of the same type, so they satisfy the same guards in $\mathcal{A}$ and so an accepting path for $s(X)$ yields the same accepting path for $s'(X)$ and vice versa.

For the equations, first observe that $|s(X)| = |s'(X)|$, as the latter was obtained from the former by a letter-to-letter replacement. Consider an equation $L = R$. If the corresponding letters in $s(L), s(R)$ were both obtained from the variables, then they were replaced at both sides with the same letters. If symbols at both sides come from the constants, then they are clearly not changed (and still equal). If one side comes from a constant in the equation, say c, and the other from the variable, say X, then in $s(X)$ at the corresponding position $s(X)$ has c. As c is a constant in the equation, the "$x = c$" is an atom in Φ and so it is in the type $\Delta(c)$ and so c is the unique letter (in whole Σ) with this type and so c in $s(X)$ is replaced with c in $s'(X)$ and so the equation is still satisfied. □

Once the domain is restricted to a finite set (Σ_0), the equations and regular constraints reduce to word equations with regular constraints: treat Σ_0 as a finite alphabet, for a symbolic automaton $\mathcal{A} = (\Sigma, Q, \Delta, q_0, F)$ create an NFA $\mathcal{A}' = (\Sigma_0, Q, \Delta', q_0, F)$, over the alphabet Σ_0, with the same set of states Q, same starting state q_0 and accepting states F and the relation defined as $(q, a, q') \in \Delta'$ if and only if there is $(q, \varphi(x), q') \in \Delta$ such that $\varphi(a)$ holds, i.e. we can move from q to q' by a in $\mathcal{A}'$ if and only if we can make this move in $\mathcal{A}$. Clearly, from the construction

Lemma 15.2. *Let Σ_0 be a set from Lemma 15.1, $\mathcal{A}$ be a symbolic automaton and $\mathcal{A}'$ the automaton as constructed above. Then*

$$L(\mathcal{A}) \cap \Sigma_0^* = L(\mathcal{A}') \ .$$

We can rewrite the parametric automata-constraints with regular constraints and consider equations over the finite alphabet D_π. From Lemma 15.1 and Lemma 15.2 it follows that the original constraints have a solution if and only if the constructed system of constraints has a solution. The extra detail is how to store the letters from Σ_0. We can identify letters with their type and store it as a bitvector (which formula is satisfied in the type). This allows to use the transitions. As in the case of regular constraints, we do not need to actually store all letters, when a letter is introduced we need to be able to verify, whether its type is realizable.

Note that there is an extra complexity in checking, whether a type is realizable.

Theorem 15.3. *Assuming that $\mathcal{T}$ is quantifier-free and is closed under complementation and conjunction, the word equations over infinite alphabets with constraints given by symbolic automata can be verified in* PSPACE.

15.3 Word equations with parametric automata

Observe that once the parameter assignment $\pi : \vec{p} \to \Sigma$ is fixed, the rest of the construction for parametric automata is exactly the same as in the case of symbolic automata: the type $\Delta_\pi(a)$ of a (under assignment π) is the set of formulas in Φ satisfied by a, i.e. $\{\varphi \in \Phi \ : \ \varphi(\pi(\vec{p}), a) \text{ holds}\}$.

Lemma 15.4. *Given a system of constraints and a parameter assignment π let $\Sigma_\pi \subseteq D$ be obtained by choosing (arbitrarily) for each realizable type a single element of this type. Then the set of constraints is satisfiable (for π) over Σ if and only if they are satisfiable (for π) over Σ_π. To be more precise, there is a letter-to-letter homomorphism $\psi : \Sigma^* \to \Sigma_\pi^*$ such that if s is a solution of a system of constraints then $\psi \circ s$ is also a solution.*

Also, the construction of the automaton is similar.

Lemma 15.5. *Given an assignment of parameters π let Σ_π be a set from Lemma 15.4, $\mathcal{A}$ be a parametric automaton and $\mathcal{A}'$ the automaton as constructed above. Then*

$$L_\pi(\mathcal{A}) \cap \Sigma_\pi^* = L(\mathcal{A}') \ .$$

It turns out that we do not need the actual π, it is enough to know which types are realisable for it, which translates to an exponential-size formula. We will use letter τ to denote subset of Φ; the idea is that $\tau = \{\Delta_\pi(a) \ : \ a \in D\} \subseteq 2^\Phi$ and if different π, π' give the same sets of realizable types, then they both yield a satisfying assignment or both not. Hence it is enough to focus on τ and not on actual π.

Lemma 15.6. *Given a system of equations and regular constraints we can non-deterministically reduce them to a formula of a form*

$$\exists \vec{p} \in \Sigma^+ \bigwedge_{t \in \tau} \exists_{t \in \tau} a_t \in \Sigma \bigwedge_{\varphi \in t} \varphi(\vec{p}, a_t) \ , \tag{15.1}$$

where $\tau \subseteq 2^\Phi$ is of at most exponential size, and a system of word equations with regular constraints of linear size and over an $|\tau|$-size alphabet, using auxiliary $\mathcal{O}(n|\tau|)$ space. The solution of the latter word equations (for which also (15.1) *holds) are solutions of the original system, by appropriate identifications of symbols.*

Proof. We guess the set τ of types of the assignment of parameters π, i.e. $\tau = \{\Delta_\pi(a) \ : \ a \in \Sigma\}$. Note that as Φ has linearly many atoms and $\tau \subseteq 2^\Phi$, then $|\tau|$ may be of exponential size, in general. The (15.1) verifies the guess: we validate whether there are values of $\vec{p}$ such that for each type $t \in \tau$ there is a value a such that $\Delta_\pi(a) = t$.

Let Σ_π be a set having one symbol per every type in τ, as in Lemma 15.4; note that this includes all constants in the equations constraints. The algorithm will not have access to particular values, instead we store each $t \in \tau$, say as a bitvector describing which atoms in Φ this letter satisfies. In particular, $|\Sigma_\pi| = |\tau|$ and it is at most exponential. In the following we will consider only solutions over Σ_π.

For each $a \in \Sigma_\pi$ we can validate, which transitions in $\mathcal{A}$ it can take: the transition is labelled by a guard which is a conjunction of atoms from Φ and either each such atom is in $\Delta_\pi(a)$ or not. Hence we can treat $\mathcal{A}$ as an NFA for Σ_π. We do not need to construct nor store it, we can use $\mathcal{A}$: when we want to make a transition by $\varphi(\vec{p}, a)$ we look up, whether each atom of φ is in $\Delta_\pi(a)$ or not. Similarly, the constraint $\mathcal{A}(X)$ is restricted to $X \in L_\pi(\mathcal{A})$ and for $X \in \Sigma_\pi^*$ this is a usual regular constraint.

We treat equations constraints as word equations over alphabet Σ_π.

Concerning the correctness of the reduction: if the system of word equations (with regular constraints) is satisfiable and the formula (15.1) is also satisfiable, then there is a satisfying assignment s over Σ_π and Σ_π^* in particular, there is an assignment of parameters for which there are letters of the given types (note that in principle it could be that s induces more types, i.e. there is a value a such that $\Delta_s(a) \notin \tau$ and so it is not represented in Σ_π, but this is fine: enlarging the alphabet cannot invalidate a solution), i.e. the transitions for a_t in the automata after the reduction are the same as in the corresponding parametric automata for the assignment π, this is guaranteed by the satisfiability of (15.1) and the way we construct the instance, see Lemma 15.5.

On the other hand, when there is a solution of the input constraints, there is one for some assignment of parameters π. Hence, by Lemma 15.4, there is a solution over Σ_π. The algorithm guesses $\tau = \{\Delta_\pi(a) \ : \ a \in \Sigma\}$ and (15.1) is true for it. Then by Lemma 15.4 there is a solution over Σ_π as constructed in the reduction and by Lemma 15.5 the regular constraints define the same subsets of Σ_π^* both when interpreted as parametric automata and NFAs. □

Theorem 15.7. *If theory T is in* PSPACE *then sequence constraints are in* EXPSPACE.

15.4 Undecidability for register automata constraints

First, observe that we can assume that we have a finite number of chosen constants, called a, b, c, among letters in Σ: to this end we introduce variables a, b, c — one for each constant, and regular

constraint $L = \{xy \ : \ x, y \in \Sigma, x \neq y, |x| = |y| = 1\}$, which can be clearly realized by a register automaton, and writing constraints $\mathcal{A}(ab), \mathcal{A}(ac), \mathcal{A}(bc)$; formally this is realized by new sequence variables.

We give a construction, which results in X, X', X'' satisfying the following conditions:

1. $X'' = aX$ and X'' begins with a and has all letters different;
2. $X' = (aX)^{|X|}$.

In the following we will use simple variants of this construction to encode (positive) integer arithmetic with addition and multiplication, with X as above representing an integer variable with value $|X|$.

For a variable X introduce a variable X'' and write a constraint $X'' = aX$ and an automaton constraint that X'' has no other occurrence of a. Write constraint $X''X' = X'X''$. This implies that there is $aw \in \Sigma^+$ such that $X'' = (aw)^k$, $X' = (aw)^\ell$ for some k, ℓ. Since X'' has only a single a, $k = 1$ and so $X' = (aX)^\ell$.

We construct another register automaton and put a constraint on $X'a$ (formally this requires a new variable); there is a special case when $X' = a$, which is trivially handled separately; hence in the following we assume that $|w| > 0$. The register automaton reads the first letter of $X'a = (aX)^k a$, i.e. a, and stores it in the register r_1. Then it enters a loop: it reads a value, stores it in r_2 and scans the input until it finds another occurrence of value from r_2. If in the meantime it finds no letter a (stored in r_1), then it rejects. After finding another copy of r_2 it goes to the next element, stores it in r_2 and continues the loop (the r_1 is not altered).

The loop ends when after finding the copy of r_2 the next letter is a, in which case we accept (and reject in all other cases).

Lemma 15.8. *The construction above yields X, X', X'' satisfying conditions 1, 2.*

Proof. By the construction we have that $X = w$ with a not being a letter in w. It was argued that $X' = (aw)^\ell$ for some ℓ.

We call the automaton action between storing for the i-th time the letter in r_2 and finding its next occurrence an i-th pass.

Suppose that aw consists of different letters. By simple induction in i-th pass we store the i-th letter of w in the register and read till i-th letter in the $i+1$-st copy of w (and store the next letter). Hence, in w-th pass we reach the last letter of $|w|$-th copy of aw, which is followed by a, so we accept after reading $(aw)^{|w|}a$, as claimed. We reject in each other case, so in particular for other powers $(aw)^n a$, where $n \neq |w|$.

If $X = w$ and w contains a letter twice, say at positions $i < j$ then at the i-th pass we will reject, as there is no a between those positions. □

Lemma 15.9. *Using register automata constraints and sequence constraints we can enforce that substitution for X, Y, Z has all letters different and $|X| + |Y| = |Z|$*

Simply write $XY = Z$ and use the constraint from Lemma 15.8 to Z, which in particular implies that all letters in X, Y are different (and other than a).

We can use a variant of construction from Lemma 15.8 to simulate multiplication: roughly, we use three variables X, Y, Z and consider an equation $(aXbYcZ)W = W(aXbYcZ)$ and make three types of checks in parallel. For Z we use the same construction as before, for Y we "reset" after each $|Y|$ full passes and for X we make one pass for each full pass of Y. This will ensure that $|Z| = |X| \cdot |Y|$.

Lemma 15.10. *Using register automata constraints and sequence constraints we can enforce that substitution for X, Y, Z has all letters different and $|X| \cdot |Y| = |Z|$*

Proof. The proof is similar as in Lemma 15.8: Choose constant a, b, c, take new variables X', Y', Z', W, using regular constraints enforce that a, b, c do not appear in X, Y, Z. Impose a sequence constraint

$$(aXbYcZ)W = W(aXbYcZ)$$

as in Lemma 15.8 this implies that $W = (aXbYcZ)^k$ for some k.

We construct a register automaton, which has three "components", working in parallel; each is similar to the automaton from Lemma 15.8. Formally the constraint is on sequence $abcWa$, so that constants a, b, c are known to it and there is an a-terminator at the end.

The first component works as Lemma 15.8 for Z: i.e. it scans consecutive copies of Z, but it ignores letters between a, c (including c, excluding a). Hence, it enforces that $k = |Z|$.

The second component works similarly for Y (ignoring letters between c and b), but once it finds c directly after a letter it stores in the register, it waits for a to appear, on which it goes to an accepting state and then restarts, i.e. it waits for b and then starts the computation again. Hence it is in an accepting state for each $(aXbYcZ)^{\ell|Y|}a$. We refer to the computation between accepting states as full pass, i.e. one full pass corresponds to $|Y|$ copies of Y read.

The third component performs the computation for X, but does one pass for X and then waits for the component responsible for Y to go to an accepting state, in which case it makes another pass, and so on. Hence, it makes one pass per full pass for Y. Once an accepting state is reached, it remains there until another full pas of Y is completed. Afterwards, it goes to a rejecting state. In this way, we ensure that $k = |X| \cdot |Y|$, hence $|Z| = |X| \cdot |Y|$. □

Lemmata 15.8, 15.9, 15.10 straightforwardly allow encoding the Hilbert's tenth problem, and so show the undecidability of sequence constraints and register constraints over infinite domains.

Exercises

Task 82 Show that the deterministic register automata are weaker than the non-deterministic ones. Take into the account the number of registers: ideally, you should show that there is a language recognised by 1-register nondeterministic automaton and not by k-register deterministic one.

It is enough to consider the logic using (boolean combinations of) equality tests between read letter and registers.

Hint: Language of words such that the last letter has occurred before seems a good candidate.

Task 83 Show that the symbolic automata are determinizable, i.e. given a non-deterministic symbolic automaton we can construct a deterministic one recognizing the same language.

Conclude that parametric automata are determinizable.

Task 84 Show that the hierarchy of languages recognized by k-parametric automata is strict, i.e. that for each k there is a language recognized by k-parametric automaton but not by any $k-1$-parametric automaton. A language

$$\bigcup_{a_1,\ldots,a_k} \{a_1, \ldots, a_k\}^*$$

is a good candidate.

Task 85 Show that the hierarchy of languages recognized by k-register automata is strict, i.e. that for each k there is a language recognized by deterministic k-register automaton but not by any $k-1$-register automaton. A language

$$\bigcup_{a_1,\ldots,a_k} (a_1 \cdots a_k)^*$$

is a good candidate.

Task 86 Show there are languages recognized by deterministic 1-register automaton and not by parametric automata (with any number of parameters).

Consider the logic using (boolean combinations of) equality tests (between read letter and registers/parameters).

Note that there is a subtlety that the parameters are guessed at the beginning but it could in principle hold that, say, for two accepted words w, ww' they are accepted for different parameters values.

Hint: A language of words $\bigcup_a (aa)^*$ is one of good candidates.

Chapter 16

Linear Monadic Second Order Unification

This Chapter is more or less based on [35], but hopefully much simplified. We present a simplified variant of linear monadic second order unification, in which 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 "$\perp$" 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_x is built solely of symbols and possibly x used once and at the end, it may be followed by the $\perp$, though. The difference is that X can ignore its argument and simply terminate the hole term.

Example 16.1.

$$Xa\perp = Yb\perp$$

There is a valid solution $X = \lambda x.a\perp$ and $Y = \lambda y.a\perp$. Note that there are also other solutions. Note that the equation $Xa = Yb$ is not satisfiable as a word equation.

Lemma 16.1. *If an equation $u = v$ is satisfiable as a word equation then the equation $u\perp = v\perp$ is satisfiable as linear monadic second order unification problem.*

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 16.2. *Satisfiability of 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 are also allowed to replace a variable by a "$\perp$".

Lemma 16.3. Pop, BlockCompNCr, PairCompNCr *are sound and complete for the linear monadic second order unification.*

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. Some properties of the algorithm are needed to show that this is sound: if we remove the variable from the left-hand side then either we substitute it with a word or with word ended with '$\perp$' and we know which case this is.

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

Lemma 16.4. *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.*

A simple reduction to the word equation case is left as an exercise.

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 begins 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 16.5. *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.

A simple proof is left as an exercise.

Our (nondeterministic) algorithm shall eliminate one variable (i.e. remove all occurrences of) 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.

Example 16.2. Suppose that we have only one equation $u = v$. If after applying Lemma 16.5 we end up with an equation $X\ldots = Y\ldots$ then we can substitute $s(X) = s(Y) = \bot$; similarly, if the equation is $X\ldots = wY\ldots$ where $X \neq Y$ and w is a word then we can take $s(X) = w\bot$ and $s(Y) = \bot$ So the only remaining case is $X\ldots = wX\ldots$. But in this case $s(X)$ is periodic with a period that is shorter than w. We can guess it, guess the exponent and make the substitution.

The situation becomes more complex when there are more equations involved, also we need to keep the instance small and this is the main result presented here.

Dependency Graph

Definition 16.6. For a system of linear second order unification define a *dependence graph*. Its vertices are labelled with variables that have at least one occurrence in the equations and there is an edge $X \xrightarrow{w} Y$ for each equation $XU = wYV$, where w is a (perhaps empty) word.

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

Lemma 16.7. *If there is an edge from* $X \xrightarrow{w} Y$ *then for each solution* s *of this system either*

- $s(X) = w'$ *where* w' *is a prefix of* w or
- $s(X)$ *has a prefix* w *(this includes* $s(X) = w\bot$*).*

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.

A simple proof is left as an exercise.

Corollary 16.8. *If there are two edges from* X *labelled with nonempty words then for each solution* s *they have the same first letter* or $s(X) = \bot$ or $s(X) = \epsilon$.

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. Note that $<$ is transitive and it is well defined also on equivalence classes of $\sim$, but it is not a partial order, as it may contain cycles.

Lemma 16.9. *If X is a minimal element of $<$, so are all its equivalent variables.*

Lemma 16.10. *Let $X_1, \ldots, X_m$ be the equivalence class of $\sim$. Then in each solution either one of them is ϵ or they all begin with the same letter (which may be $\perp$).*

The proof is obvious.

Lemma 16.11. *Let s be a solution, and let $X_1, \ldots, X_m$ be all variables equivalent to X_1, assume that $s(X_i) \notin \{\epsilon, \perp\}$ for each i and let a be the first letter of $s(X_1)$. Then after left-popping a from all $X_1, \ldots, X_m$ either one variable is removed or*

- *all $X_1, \ldots, X_m$ are still equivalent and*
- *each edge $X_i \xrightarrow{w} Y$ for $w \neq \epsilon$ is replaced with $X_i \xrightarrow{a^{-1}w} Y$.*
- *each edge $Y \xrightarrow{w} X_i$ for $w \neq \epsilon$ is replaced with $Y \xrightarrow{wa} X_i$.*

The proof follows by a simple case inspection of edges, depending on the occurrences of variables X_i on both sides of the equation.

Lemma 16.12. *If relation $<$ on a component of a graph (i.e. one equivalence of $\sim$) is empty, then there is a solution in which $s(X) = \perp$ for each variable in this component.*

Lemma 16.13. *If relation $<$ on the graph is acyclic, X is one of minimal nodes and there is an edge $X \xrightarrow{aw} Y$, then either there is a solution $s(X) \in \{\epsilon, \perp\}$ or left popping (appropriate) a from all variables in the equivalence class of X yields an instance with smaller sum of label lengths over all edges.*

Proof. If there is no outgoing edge from the nodes in the equivalance class, then this equivalance class form a component in the dependency graph and we apply Lemma 16.12.

Otherwise, there is an outging edge, say its first letter is a, let us left-pop this a. By Lemma 16.11:

- the labels on edges $X_i \xrightarrow{\epsilon} X_j$ remain the same
- the labels on edges $X_i \xrightarrow{w} Y$ shorten by one letter (there is at least one such edge)
- by assumption there are no edges $Y \xrightarrow{w} X_i$ for $w \neq \epsilon$.
- all other edges remain as they were.

Hence indeed the total length of labels decreases. □

Note that we change the $<$ order and the $\sim$ 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.

Lemma 16.14. *If the $<$ has a cycle and the concatenation of labels on a path from X to X is w, then using $\mathcal{O}(|w|)$ compression steps we obtain an instance in which we either remove a variable or obtain an instance in which the concatenation of the labels from X to X is a^k for some k.*

Proof. Suppose that w is made as concatenation of $w_1, \ldots, w_k$. Compressing pairs within each w_i can be done as before. If a is the last letter of some w_i and b is the first letter of some w_j (and all ws in between are ϵ) left left-popping b from the equivalence class of appropriate variable moves b to w_i, as requested: there is a sequence of variables $X_1, X_2, \ldots, X_k$ and variables Z, Y such that there are equations

$$\begin{aligned} Z \ldots &= w_i a X_1 \ldots \\ X_1 \ldots &= X_2 \ldots \\ &\vdots \\ X_{k-1} \ldots &= X_k \ldots \\ X_k \ldots &= b w_{i+1} Y \ldots \end{aligned}$$

Now left-popping b from all X_is yields

$$\begin{aligned} Z \ldots &= w_i a b X_1 \ldots \\ b X_1 \ldots &= b X_2 \ldots \\ &\vdots \\ b X_{k-1} \ldots &= b X_k \ldots \\ b X_k \ldots &= b w_{i+1} Y \ldots \end{aligned}$$

which is simplified to

$$\begin{aligned} Z \ldots &= w_i a b X_1 \ldots \\ X_1 \ldots &= X_2 \ldots \\ &\vdots \\ X_{k-1} \ldots &= X_k \ldots \\ X_k \ldots &= w_{i+1} Y \ldots \end{aligned}$$

In particular, ab can be compressed.

When we make the block compression, popping a single a does not affect the edges labelled with a^ℓ (including ϵ). However, the whole sequence has an end, so we will pop at most the length of the a-block times, moving the block to a single label. □

Lemma 16.15. *If there is a cycle in the dependency graph from X to X such that each of its edges is labelled with a power of a (perhaps ϵ) then there is a variable $Y \sim X'$, where X' is on this cycle, such that $s(Y) \in a^*\{\epsilon, \bot\}$.*

Proof. Some of the consecutive variables on the cycle may be equivalent, by taking the first of each equivalence class we can identify variables $X_1, \ldots, X_k$ such that there are $X'_1, \ldots X'_k$ where

- $X_i \sim X'_i$
- $X'_i \xrightarrow{a^{\ell_i}} X_i$

Then

- each of the variables in the equivalence classes of $X_1, \ldots, X_k$ begins with a or
- there is a variable Y in n the equivalence classes of $X_1, \ldots, X_k$ such that $s(Y) \in \{\epsilon, \bot\}$.

In the first case we can left-pop a from each of the variables of the equivalance class and by Lemma 16.11 the invariant still holds. Hence at some point this will terminate and we obtain Y with $s(Y) \in \{\epsilon, \bot\}$, so initially $s(Y) \in \{a^\ell, a^\ell \bot\}$, as claimed. □

The algorithm

We can now move to the algorithm. A phase ends when one variable is removed. At the beginning of the phase the equation is reduced to size $\mathcal{O}(n^2)$ using Pop, BlockCompNCr, PairCompNCr appropriate amount of times (each application reduces the size by at least 1, till appropriate size is reached).

If there is no edge labelled with non-empty word, then By Lemma 16.12 we can set $s(X) = \bot$ for each variable.

If the graph is acyclic then by Lemma 16.13 we can pop letters from minimal nodes and this shortens the total length of labels. So either we remove a variable or the graph stops being acyclic.

If there is a cycle, then by Lemma 16.14 we can, using $\mathcal{O}(|w|)$ compressions, where w is the label on the cycle, remove a variable or make a cycle with label a^k.

Then by Lemma 16.15 there is a variable which has a substitution in $a^k\{\epsilon, \bot\}$ and by Lemma 16.4 the k is at most exponential.

Exercises

Task 87 Show that the variant of monadic second order unification considered here is NP-hard.

Task 88 Prove Lemma 16.4.

Task 89 Prove Lemma 16.5.

Task 90 Prove Lemma 16.7.

Chapter 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 17.1. 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 17.2. *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* s assigns 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.

Exercises

Task 91 Show that the term unification (so the case in which the variables can represent only well-formed terms) can be solved in linear time, in the sense that we can say, whether it has a solution or not. Some assumptions on the model may be needed.

Chapter 18

General second order unification

In general, the second order unification is undecidable

Theorem 18.1. *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

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

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

Chapter 19

Context Unification

19.1 Introduction

19.1.1 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 [6, 7] (who also coined the name) and independently by Schmidt-Schauß [56]. It found usage in analysis of rewrite systems with membership constraints [6, 7], analysis of natural language [46, 45], distributive unification [57], bi-rewriting systems [31].

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 [20] (even in very restricted cases [16, 30, 32]), 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 [50]) and up to recently there were essentially only three different algorithms for this problem [40, 52, 50];

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 [53], 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 19.1.2. 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 [7];
- 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 [58] (this problem is even known to be NP-complete [36]);
- a fragment in which every variable and context variable occurs at most twice (such equations are usually called *quadratic*) is decidable [30];
- a fragment in which there are only two context variables is decidable [61];
- 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 [60];
- context unification reduces to its fragment in which the signature contains only one binary symbol and constants [34];
- context unification with one context variable is known to be in NP [17] and some of its fragments are in P [18]. 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.

19.1.2 Extensions and connections to other problems

The context unification was shown to be equivalent to 'equality up to constraint' problem [46] (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 [45]. It is known that the first-order theory of one-step rewriting constraints is undecidable [64, 43, 67]. The case of general context unification was improved by Vorobyov, who showed that its $\forall\,\exists^8$-equational theory is Π^0_1-hard [68].

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 [35]. This fragment on one hand easily reduces to context unification and on the other hand it is known to be decidable [59] (in fact its generalisation to higher-order unification is decidable as well [62] and it is known that bounded second order unification is NP-complete [36]). 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 [33], defined by Levy [30]: 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 [63, 11, 12].

19.1.3 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 (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.

19.2 Compression of trees

19.2.1 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.

[1] Note that by work of Levy [34] 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.

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.

19.2.2 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 19.1. *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). □

19.3 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. [61], 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 $\mathcal{X}$. 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 19.2. The *terms* over Σ, $\mathcal{X}$, $\mathcal{V}$ are ground terms with alphabet $\Sigma \cup \mathcal{X} \cup \mathcal{V}$ in which we extend ar to $\mathcal{X} \cup \mathcal{V}$ by $\mathrm{ar}(X) = 1$ and $\mathrm{ar}(x) = 0$ for each $X \in \mathcal{V}$ and $x \in \mathcal{X}$.

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 19.3. 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 $x \in \mathcal{X}$. 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 $X \in \mathcal{X}$.

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 $X \in \mathcal{X}$ 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 19.3 and that it comes from X (or x) if it was obtained from $s(X)$ (or $s(x)$, respectively) in Definition 19.3.

Example 19.1. 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 [51, 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.

19.4 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 19.9.5.

A very general class of operations is sound:

Lemma 19.4. *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 19.5. 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 19.6. 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 PattCompNCr$(ab, \text{`}u = v\text{'})$ *or* PattCompNCr$(a, \text{`}u = v\text{'})$ *or* PattCompNCr$((f, i, c), \text{`}u$ *is complete. To be more precise, the returned equation $u' = v'$ has a solution s' such that $s'(u') =$* TreePattComp$(ab, s(u))$ *(or $s'(u') =$* TreePattComp$(a, s(u))$ *or $s'(u') =$* 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 19.4 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 $\mathsf{PattCompNCr}(ab, \text{`}u = v\text{'})$ 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. □

19.5 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.

19.6 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 19.7. *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 14 Uncross$((f,i,c),\text{`}u=v\text{'})$

1: **for** $x \in \mathcal{X}$ **do**
2: **if** $s(x) = c$ **then** ▷ Guess
3: replace each x in $u = v$ by c ▷ s is no longer defined on x
4: let $m \leftarrow \mathrm{ar}(f)$
5: **for** $X \in \mathcal{V}$ **do**
6: **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
7: 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 $\mathcal{X}$ and extend s by setting $s(x_j) = t_j$
8: **if** $s(X)$ is empty **then** ▷ Guess
9: remove X from the equation: replace each $X(t)$ in the equation by t
10: **for** new variables $x \in \mathcal{X}$ **do**
11: **if** $s(x) = c$ **then** ▷ Guess
12: 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 19.15.

Lemma 19.8. *Let* $\mathrm{ar}(f) \geq i \geq 1$ *and* $\mathrm{ar}(c) = 0$, *then* Uncross$((f,i,c),\text{`}u=v\text{'})$ *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 19.4 we obtain that Uncross$((f,i,c),\text{`}u=v\text{'})$ is sound.

Concerning the second part of the lemma, we proceed as in Lemma **??**: let Uncross$((f,i,c),\text{`}u=v\text{'})$ 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 Uncross.
- For a context variable X from which we did not pop a letter we set $s'(X) = s(X)$.
- For X from which Uncross popped down $f(x_1, \ldots, x_{i-1}, \bullet, x_i, \ldots, x_{m-1})$ let $s(X) = sf(t_1, \ldots, t_{i-1}, \bullet, t_i, \ldots, t_n$ (such a representation is possible as 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 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 $\mathsf{Pop}((f, i, c), \text{`}u = v\text{'})$ should have popped the last letter of f from X, as in line 6 we were supposed to guess according to s, contradiction. □

19.7 Uncrossing patterns

We can state a general lemma about uncrossing.

Lemma 19.9. 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'.*

19.8 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 15 PreProc(‘$u = v$’, Σ) 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)

Algorithm 16 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:             Pop(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
```

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

Theorem 19.10. 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 19.11. *Context unification is in* PSPACE.

The proofs of both theorems are postponed.

19.9 Analysis of the algorithm

The analysis focus on several points. Firstly, we show that we can trim the input signature, see Section 19.9.1, 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 19.9.2. Then, in Section 19.9.3 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 19.9.4. 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 19.9.5.

19.9.1 Input signature

Trimming of signature does not affect the satisfiability and the needed space.

Lemma 19.12. *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 19.13. *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. □

19.9.2 Exponent of periodicity

The following lemma shows that the size of the a-chains can be limited in case of size-minimal solutions.

Lemma 19.14 (Exponent of periodicity bound [60])**.** *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 19.14 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 [27]).

Note that bound applies to *every* signature: given an equation, we can change the signature and the bound remains the same.

19.9.3 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 19.15. *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 19.16. *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 19.15 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 19.17. *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 19.15, 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. □

19.9.4 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 19.18. *Suppose that the equation $u = v$ has a solution s (over a signature Σ) 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 Σ 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 19.6 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 19.19. *Suppose that the equation $u = v$ has a solution s (over a signature Σ) 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 Σ 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 19.9 the obtained equation has a solution of the same size, for which the subpattern is non-crossing. Compress this subpattern. By Lemma 19.18 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 19.18.

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 19.16, 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 19.17. 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 19.15). 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 19.16, 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 19.20. *Suppose that the equation $u = v$ has a solution s (over a signature Σ) 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 Σ plus the letters replacing the compressed subpatterns).*

Proof. The bound on the number of introduced letters follow from Lemma 19.17.

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 19.16 there are at most $n(k+1)$ different crossing subpatterns, see Lemma 19.16, 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 19.19.

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 19.10) for the case of ContextEqSatSimp.

proof of Theorem 19.10 for ContextEqSatSimp . Suppose that we are given a satisfiable equation $u = v$. By Lemma 19.12 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 19.18, 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 19.18.

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 19.19. After uncrossing it and the compression the size of the equation and size-minimal solution decrease, see Lemma 19.19.

If the equation has at most $48n^2k^2$ letters then choose according to Lemma 19.20. 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 19.20 or Lemma 19.19, 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 19.19 guarantees that the size of the equation does not increase.

Concerning the number of phases: each compression according to Lemma 19.20 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 19.20? 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. □

19.9.5 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 19.21. *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 19.12: take any signature Σ' and let Σ be the simple signature. If $s(u)$ uses a letter $g \in \Sigma' \setminus \Sigma$ then it must be used inside a substitution $s(X)$. We can replace all occurrences of g with a letter $f_i \in \Sigma'$ of the same arity; this is possible as there is a letter of each arity up to k in Σ and the arity of Σ' 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 19.10 in case of ContextEqSat.

proof of Theorem 19.10 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 19.18–19.20 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 19.1 it is over a signature of maximal arity at most k and so from Lemma 19.21 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 19.11.

proof of Theorem 19.11. By Theorem 19.10 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 19.12. 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 19.14. 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 19.18, 19.19 and 19.20. 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. □

Bibliography

[1] Kenneth I. Appel. One-variable equations in free groups. *Proceedings of the American Mathematical Society*, 19:912–918, 1968.

[2] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993. doi:10.1137/0222017.

[3] Dimitri Bormotov, Robert Gilman, and Alexei Myasnikov. Solving one-variable equations in free groups. *Journal of Group Theory*, 12:317–330, 2009. doi:10.1515/JGT.2008.080. URL `https://doi.org/10.1515/JGT.2008.080`.

[4] Witold Charatonik and Leszek Pacholski. Word equations with two variables. In *IWWERT*, pages 43–56, 1991. doi:10.1007/3-540-56730-5_30.

[5] Ian M. Chiswell and Vladimir N. Remeslennikov. Equations in free groups with one variable. I. *Journal of Group Theory*, 3(4), 2000. doi:10.1515/jgth.2000.035. URL `https://doi.org/10.1515/jgth.2000.035`.

[6] 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. URL `http://dx.doi.org/10.1006/jsco.1997.0185`.

[7] 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. URL `http://dx.doi.org/10.1006/jsco.1997.0186`.

[8] Elena Czeizler. The non-parametrizability of the word equation $xyz{=}zvx$: A short proof. *Theor. Comput. Sci.*, 345(2-3):296–303, 2005. doi:10.1016/J.TCS.2005.07.012. URL `https://doi.org/10.1016/j.tcs.2005.07.012`.

[9] Joel D. Day and Florin Manea. On the structure of solution-sets to regular word equations. *Theory Comput. Syst.*, 68(4):662–739, 2024. doi:10.1007/S00224-021-10058-5. URL `https://doi.org/10.1007/s00224-021-10058-5`.

[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. URL `http://dx.doi.org/10.1016/j.ic.2005.04.002`.

[12] Volker Diekert, Artur Jeż, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. *Inf. Comput.*, 251:263–286, 2016. doi:10.1016/j.ic.2016.09.009. URL `http://dx.doi.org/10.1016/j.ic.2016.09.009`.

[13] Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. In *ICALP*, pages 408–419, 2004. 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] Pál Dömösi and Géza Horváth. Alternative proof of the lyndon-schützenberger theorem. *Theoretical Computer Science*, 366(3):194–198, 2006. doi:10.1016/j.tcs.2006.08.023. URL `https://doi.org/10.1016/j.tcs.2006.08.023`.

[16] 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. URL `http://dx.doi.org/10.1016/S0304-3975(06)80003-4`.

[17] 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. URL `http://dx.doi.org/10.1016/j.jsc.2008.10.005`.

[18] 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`.

[19] Robert H. Gilman and Alexei G. Myasnikov. One variable equations in free groups via context free languages. *Contemporary Mathematics*, 349:83–88, 2004.

[20] 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. URL `http://dx.doi.org/10.1016/0304-3975(81)90040-2`.

[21] Tero Harju, Juhani Karhumäki, and Wojciech Plandowski. Independent systems of equations. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 13, pages 392–416. Cambridge University Press, 2002.

[22] Artur Jeż. One-variable word equations in linear time. *Algorithmica*, 74:1–48, 2016. doi:10.1007/s00453-014-9931-3. URL `http://dx.doi.org/10.1007/s00453-014-9931-3`.

[23] Artur Jeż. Recompression: a simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, Mar 2016. 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. doi:10.1145/1217856.1217858.

[25] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM*, pages 181–192, 2001. doi:10.1007/3-540-48194-X_17.

[26] Olga Kharlampovich, Igor G. Lysënok, Alexei G. Myasnikov, and Nicholas W. M. Touikan. The solvability problem for quadratic equations over free groups is NP-complete. *Theory of Computing Systems*, 47(1):250–258, 2010. doi:10.1007/s00224-008-9153-7. URL `https://doi.org/10.1007/s00224-008-9153-7`.

[27] Antoni Kościelski and Leszek Pacholski. Complexity of Makanin's algorithm. *J. ACM*, 43(4):670–684, 1996. doi:10.1145/234533.234543. URL `http://doi.acm.org/10.1145/234533.234543`.

[28] Markku Laine and Wojciech Plandowski. Word equations with one unknown. *Int. J. Found. Comput. Sci.*, 22(2):345–375, 2011. doi:10.1142/S0129054111008088.

[29] 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.

[30] 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. URL http://dx.doi.org/10.1007/3-540-61464-8_63.

[31] Jordi Levy and Jaume Agustí-Cullell. Bi-rewrite systems. *J. Symb. Comput.*, 22(3):279–314, 1996. doi:10.1006/jsco.1996.0053. URL http://dx.doi.org/10.1006/jsco.1996.0053.

[32] 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. URL http://dx.doi.org/10.1006/inco.2000.2877.

[33] 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. URL http://dx.doi.org/10.1007/10721975_11.

[34] 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. URL http://dx.doi.org/10.1007/3-540-45610-4_23.

[35] 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.

[36] 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. URL http://dx.doi.org/10.1093/jigpal/jzq010.

[37] A. A. Lorents. Representations of sets of solutions of systems of equations with one unknown in a free group. *Dokl. Akad. Nauk. SSSR*, 178:290–292, 1968.

[38] Roger C. Lyndon and Marcel-Paul Schützenberger. The equation $a^M = b^N c^P$ in a free group. *Michigan Mathematical Journal*, 9(4):289–298, 1962.

[39] Roger Conant Lyndon. Equations in free groups. *Transansaction of American Mathematical Society*, 96:445–457, 1960. doi:10.1090/S0002-9947-1960-0151503-8. URL https//doi.org/10.1090/S0002-9947-1960-0151503-8.

[40] Gennadiĭ Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).

[41] Gennadiĭ Makanin. Equations in a free group. *Izv. Akad. Nauk SSR*, Ser. Math. 46:1199–1273, 1983. English transl. in Math. USSR Izv. 21 (1983).

[42] Gennadiĭ 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.

[43] 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. URL http://dx.doi.org/10.1007/3-540-62950-5_75.

[44] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. ISBN 9780521835404.

[45] 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. doi:10.3115/979617.979670. URL http://dx.doi.org/10.3115/979617.979670.

[46] 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. URL http://dx.doi.org/10.1007/3-540-63104-6_4.

[47] Jakob Nielsen. über die Isomorphismen unendlicher Gruppen ohne Relation. *Mathematische Annalen*, 79:269–272, 1918.

[48] Dirk Nowotka and Aleksi Saarela. An optimal bound on the solution sets of one-variable word equations and its consequences. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 136:1–136:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. ISBN 978-3-95977-076-7. doi:10.4230/LIPIcs.ICALP.2018.136. URL https://doi.org/10.4230/LIPIcs.ICALP.2018.136.

[49] Seraphin D. Eyono Obono, Pavel Goralčik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In *MFCS*, pages 336–341, 1994. doi:10.1007/3-540-58338-6_80.

[50] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725. ACM, 1999. doi:10.1145/301250.301443. URL http://doi.acm.org/10.1145/301250.301443.

[51] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. doi:10.1145/990308.990312. URL http://doi.acm.org/10.1145/990308.990312.

[52] 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. URL http://dx.doi.org/10.1007/BFb0055097.

[53] RTA problem list. Problem 90. http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html, 1990.

[54] 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.

[55] Aleksi Saarela. Word equations where a power equals a product of powers. In Heribert Vollmer and Brigitte Vallée, editors, *STACS*, volume 66 of *LIPIcs*, pages 55:1–55:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. ISBN 978-3-95977-028-6. doi:10.4230/LIPIcs.STACS.2017.55. URL https://doi.org/10.4230/LIPIcs.STACS.2017.55.

[56] Manfred Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universität, 1994.

[57] 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. URL http://dx.doi.org/10.1016/S0304-3975(98)00081-4.

[58] 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. URL http://dx.doi.org/10.1093/logcom/12.6.929.

[59] Manfred Schmidt-Schauß. Decidability of bounded second order unification. *Inf. Comput.*, 188 (2):143–178, 2004. doi:10.1016/j.ic.2003.08.002. URL http://dx.doi.org/10.1016/j.ic.2003.08.002.

[60] 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. URL `http://dx.doi.org/10.1007/BFb0052361`.

[61] 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. URL `http://dx.doi.org/10.1006/jsco.2001.0438`.

[62] 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. URL `http://dx.doi.org/10.1016/j.jsc.2005.01.005`.

[63] 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. URL `http://dx.doi.org/10.1007/3-540-55124-7_4`.

[64] 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. URL `http://dx.doi.org/10.1016/S0304-3975(98)00083-8`.

[65] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-65367-8.

[66] 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.

[67] 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. URL `http://dx.doi.org/10.1007/3-540-62950-5_76`.

[68] Sergei G. Vorobyov. $\forall\exists^*$-equational theory of context unification is Π_1^0-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. URL `http://dx.doi.org/10.1007/BFb0055810`.