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).

The study of satisfiability word equations has a long tradition. It is fairly easy to see that WordEquation reduces to Hilbert's 10th Problem. Hence in the mid 1960s the Russian school of mathematics outlined the roadmap 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 a seminal paper [43] 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 [44, 45].

1.2 Definitions

Definition 1.1 (Alphabet, variables)**.** In context of word equations we always 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 $s : \mathcal{X} \mapsto \Gamma^+ \supseteq \Sigma^+$. 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 (Constraints)**.** *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 SLPs

Definition 1.6 (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.

1.3.1 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 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 1.7. *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. □

1.5 Satisfiability via SLPs

This section is based on [55].

Definition 1.8 (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 or at its beginning or end.

Fact 1.9. *There are $|u| + |v| + 2$ cuts in a word equation $u = v$.*

Definition 1.10. 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 1.11. 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 1.12. *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 1.13. *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 $s(v)$).

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 1.13, 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 1.14. *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 v_i 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 overlaps with some 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

Task 6 Show that the problem of word equations with context-free constraints is undecidable. (Meaning, that we allow constraints with languages from the class of context-free languages).

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 allowe NFAs.
Deduce from this that word equations with regular constraints are PSPACE-hard.

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?

Task 10 Give a PSPACE algorithm for the equivalence problem of two composition systems.

Problem: Equivalance of Compositions Sytems
Input: Two composition systems $\mathcal{C}$ and $\mathcal{C}'$.
Task: Decide, whether $\mathrm{val}(\mathcal{C}) = \mathrm{val}(\mathcal{C}')$.

Give also a coNP algorithm for this problem.

Chapter 2

Satisfiability of word equations in PSPACE

Idea

In Section 1.5 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.

"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

```
while |w| > 1 do
    L ← list of letters in w
    for a ∈ L do
        compress blocks of a
    P ← list pairs in w
    for ab ∈ P do
        replace all occurrences of ab in w by a fresh letter c
```

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. *replacing occurrences of a variable X with a nonempty word $w \neq \epsilon$.*

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 $s(X) = w$ 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

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

```
1: for X ∈ 𝒳 do
2:     let b be the first letter of s(X)                          ▷ Guess
3:     if the first letter of s(X) is b then
4:         replace each X in u and v by bX                        ▷ Implicitly change s(X) = bw to s(X) = w
5:         if s(X) = ϵ then                                       ▷ Guess
6:             remove X from u and v
7:     ...                                  ▷ 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:     while there is a ∈ L without crossing blocks do                ▷ Guess
4:         BlockCompNCr(a)
5:         remove a from L
6:     for a ∈ L do                                                   ▷ |L| ≤ 2n
7:         Pop (a)
8:         BlockCompNCr(a)
9:     P ← list pairs in the equation
10:    while there is non-crossing ab ∈ P do
11:        take some non-crossing ab ∈ P
12:        PairCompNCr(a, b)
13:        remove ab from P
14:    for ab ∈ P do                                                  ▷ |P| ≤ 2n
15:        Pop (a, b)
16:        PairCompNCr(a, b)
17: Solve the problem naively          ▷ With sides of length 1, the problem is trivial
```

One iteration of the main loop is a called a *phase*.

Lemma 2.9. *Consider two consecutive letters a, b at the beginning of the phase in u or v. At least one of those letters is compressed in this phase.*

Note that this does not depend on the nondeterministic choices.

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

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

Lemma 2.10. *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 the next Chapter 3 and it follows from a more general bound on the exponent of periodicity.

Lemma 2.11. *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.

By Lemma 2.5 there are at most $2n$ crossing blocks and pairs. Each uncrossing brings at most $2n$ new letters, so in total at most $8n^2$ new letters are introduced during a phase.

On the other hand, from Lemma 2.9 the equation gets shorter: if the words between variables are $w_1, w_2, \dots w_{n+2}$. Each w_i is reduced by at least $\frac{|w_i|-1}{3}$ letters due to compression. Thus we remove at least

$$\sum_{i=1} \frac{|w_i|-1}{3} \geq \frac{|uv|-n}{3} - \frac{n+2}{3} = \frac{|uv|}{3} - \frac{2n+2}{3}$$

So the equation stabilises at at most $24n^2+2n+2$ letters: if $|uv| \leq 24n^2+2n+2$ then $|u'v'| \leq 24n^2+2n+2$:

$$\begin{aligned} |u'v'| &\leq |uv| - \frac{|uv|}{3} + \frac{2n+2}{3} + 8n^2 \\ &= \frac{2|uv|}{3} + \frac{2n+2}{3} + 8n^2 \\ &\leq \frac{2}{3}(24n^2+2n+2) + \frac{2n+2}{3} + 8n^2 \\ &= 24n^2+2n+2 \end{aligned}$$

As the input equation has length n, so smaller than $24n^2+2n+2$ a simple inductive argument shows the bound. □

Theorem 2.12. *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 11 Show that for a word equation with n occurrences of variables there are at most $2n$ different crossing pairs and at most $2n$ different letters with crossing blocks.

Task 12 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 13 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 14 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 15 Consider a word $w \in \Sigma^*$ such that none of its two consecutive letters are the same. 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 [48] as well as in Vijay Vazirani *Approximation algorithms* book [71].

Task 16 Using Tasks 13–15 give a linear-time algorithm for compressing a word based on the algorithm presented during the lecture. (Model assumption: all letters are integers, we can employ RadixSort on them.)

Task 17 Using Tasks 13–15 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 18 Underspecified, but should be doable with the hints inside.

Show that for a suitable constant c, if a word equation with at most n occurrences of variables and size at least cn^2 has a length-minimal solution s, then one of the following holds:

- there is a non-crossing pair ab in $s(u)$;
- for some non-crossing a there is a block a^ℓ in $s(u)$, for $\ell > 1$;
- there is a crossing ab such that uncrossing and compressing it does not increase the size of the equation;
- there is a with crossing blocks such that uncrossing and compressing a-blocks does not increase the size of the equation.

Employ this observation in an algorithm for word equations.

Task 19 (Long and tedious, but not that difficult) The goal of this task is to create a variant of algorithm that performs only compression of pairs, perhaps pairs of the same letter.

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 20 Suppose that the above algorithm was implemented, 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 size-minimal solution is at most doubly exponential.

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

Task 21 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 12 should be helpful.

Chapter 3

Exponent of periodicity: simple case

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

3.1 Exponent of periodicity

Definition 3.1. 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 $u \in \Sigma^+$; Σ-*exponent of periodicity* $\mathrm{per}_\Sigma(w)$ restricts the choice of u to Σ.

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

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

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

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

Theorem 3.2 (Kościelski and Pacholski [31])**.** *Given an equation $u = v$ its exponent of periodicity* $\mathrm{per}(u = v)$ *is at most exponential in $|uv|$:*

$$\mathrm{per}(u = v) = 2^{\mathcal{O}(|uv|)} \ .$$

.

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

In this chapter we shall show an exponential bound on the Σ-exponent of periodicity (so in other words: exponential bound on the lengths of maximal blocks in length-minimal solutions). The general case needs some further knowledge in word combinatorics, Chapter 4, and is given in Chapter 5.

3.2 Touching blocks and their lengths

Definition 3.3. A maximal a-block in $s(u)$ or $s(v)$ is *touching* for a solution s, if it contains an explicit letter or non-empty a-prefix or a-suffix of some occurrence of $s(X)$. In other words, it touches a cut.

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

Lemma 3.4. *Given a word equation of length n and n_v occurrences of variables, there are at most $n + 2n_v$ different touching blocks.*

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

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

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

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

Example 3.1. Consider an equation

$$XabXXa = aXbYYY \ . \tag{3.1}$$

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

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

3.3 Arithmetic expressions and their equations

The Example 3.1 suggests that the solution can have a-blocks whose lengths are parametrised and those parameters are to satisfy a certain system of Diophantine equations. We first formalise, what type of lengths we are dealing with.

Definition 3.5. Arithmetic expressions may use positive constants and variables $\{L_X, R_X\}_{X\in\mathcal{X}}$, which represent positive natural numbers. They are usually denoted by E_1, E_2, ..., E_k.

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

- it uses variables $\{L_X, R_X\}_{X\in\mathcal{X}}$, where $\mathcal{X}$ is a set of variables used in the word eqution
- the sum of constants in those expressions is at most n
- variables L_X (R_X) is multiplied only by positive constants and the sum of those constants is at most the number of occurrences of the variable X in the word equation.

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

Consider an equation $u = v$ and its solution s. We shall construct a system of small Diophantine expressions associated with $u = v$ and s, in the following way: list the touching lengths of a-blocks in $s(u)$ and $s(v)$, $e_1, \ldots, e_k$, we create small Diophantine expressions $E_1, \ldots, E_k$. If the e_i includes a prefix/suffix ℓ_X/r_X then we include L_X/R_X in E_i. When e_i spans (in total) over m explicit letters a then we add to E_i the constant m.

Lemma 3.6. *For a given word equation $u = v$ with variables $\mathcal{X}$ and lengths of touching blocks e_1, e_2, ..., e_k the constructed set of arithmetic expressions E_1, E_2, ..., E_k in variables $\{L_X, R_X\}_{X\in\mathcal{X}}$ is small, moreover $e_i = E_i[\{\ell_X, r_X\}_{X\in\mathcal{X}}]$, where ℓ_X and r_X are the lengths of the a-prefix and a-suffix of $s(X)$.*

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

Example (Example 3.1 continued). For the equation (3.1) the maximal blocks (from left to right) are e_1, e_2, e_3, e_4 and we have

$$E_1 = L_X + 1,\ E_2 = 2L_X + 1,\ E_3 = L_X + 1,\ E_4 = 3L_Y \ .$$

Let $e_1, \ldots, e_k$ be touching lengths of maximal blocks of a and $E_1, \ldots, E_k$ the corresponding arithmetic expressions. Define a system of equations: partition the lengths $e_1, e_2, \ldots, e_k$ into groups of the same value. For each such group $\{e_{i_1}, e_{i_2}, \ldots, e_{i_j}\}$ add equations

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

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

Lemma 3.7. *$\{\ell_X, r_X\}_{X\in\mathcal{X}}$ is a solution of $\mathcal{D}$.*

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

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

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

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

3.4 Parametrised solutions

Consider a length-minimal solution $s(X)$. Each maximal block of a in $s(X)$ that is not touching is of touching length. Assign to each such block one of e_i (of the same value) in an arbitrary way. To a touching block assign its length.

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

Lemma 3.8. *$S(u)$ and $S(v)$ are obtained by replacing each maximal a-block of length e_i (or assigned to length e_i) with a^{E_i}.*

Proof. Easy induction. □

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

Lemma 3.9. *Each $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is a well-defined substitution.*

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

Proof. First: obvious.

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

Theorem 3.10. *If $\{\ell'_X, r'_X\}_{X\in\mathcal{X}}$ is a solution of $\mathcal{D}$ then $S[\{\ell'_X, r'_X\}_{X\in\mathcal{X}}]$ is a solution of the word equation.*

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

Proof. Both follow from the second observation in Lemma 3.7. □

3.5 Solutions of system of linear Diophantine equations

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

$$\sum_{j=1}^{r} n_{i,j} x_j = n_i \qquad \text{for } i = 1, \ldots, m \tag{3.2a}$$

together with inequalities guaranteeing that each x_i is positive

$$x_j \geq 1 \qquad \text{for } j = 1, \ldots, r \ . \tag{3.2b}$$

In the following, we are interested only in *natural* solutions, i.e. the ones in which each component is a natural number; observe that inequality (3.2b) guarantees that each of the component is greater than zero. We introduce a partial ordering on such solutions:

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

A solution $(q_1, \ldots, q_r)$ is *minimal* if it satisfies (3.2) 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 3.11 (cf. [31, Corollary 4.4])**.** *For a system of linear Diophantine equations* (3.2) *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 extension of the original proof of Kościelski and Pacholski, which takes in to the account also the inequalities. For completeness, we recall its proof, as given in [31].

proof, cf. [31]. The proof follows by estimation based on work of [72] and independently by [33]

Claim 3.11.1 (cf. [31])**.** *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 3.11.1, we recall the argument of [31].
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

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

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

3.6 Bound on Σ-exponent of periodicity

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

As a first step, let us estimate the values w, r, c from Lemma 3.11 in case of system of equations $\mathcal{D}$

Lemma 3.12. *For a system of equations* $\mathcal{D}$ *Lemma 3.11 yields a bound of*

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

on coordinates of its minimal solutions.

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

Observe that as the matrix for the inequalities, so the matrix C from Claim 3.11.1, in our case is an identity, it is enough to consider the bound on the values of determinants of square submatrices of A from this Claim, i.e. the matrix of equalities. Those are coefficients by the variables, so by definition of small set of arithmetic expressions, this sum is at most $4n_v$.

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

Lemma 3.13 (cf. [31]). *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 lengths of longest a-block in $s'(u)$ is $\mathcal{O}(\mathrm{poly}(n)e^{4n_v/e})$, while for any variable it 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 lengths of the a-prefix and suffix of $s'(X)$. 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

$$\ell''_X \leq \ell'_X \quad \text{and} \quad r''_X \leq r'_X \quad \text{for each } X \in \mathcal{X} \tag{3.3}$$

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

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

Now we can use Lemma 3.12 to get our bounds (note that we need to sum some of such coordinates to get a length of one a-block). □

As a short corollary we obtain:

Theorem 3.14 (cf. [31]). *The Σ-exponent of periodicity of a word equation $u = v$ with n_v occurrences of variables is $\mathcal{O}(\mathrm{poly}(n)e^{4n_v/e})$.*

Proof. We can estimate the lengths of maximal a-blocks for each a separately by Lemma 3.13. □

Exercises

Task 22 Show Hadamard inequality for a square matrix $N = (n_{i,j})_{i,j=1}^{k}$

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

Task 23 Show that the exponential bound on the length of a-blocks for length-minimal solutions is tight (the exact constant at the exponent is not tight, though).

Task 24 Show that we can use the a-presentation approach for the compression algorithm: 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 25 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. Do you see some connection with block compression, both ways?

Task 26 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 our original algorithm

Task 27 Show that the exponential bound on the exponent of periodicity (but not with a $2^{\mathcal{O}(n)}$ bound, though) can be inferred already from our original algorithm for word equations plus the bound on the length of a-blocks in the length-minimal solutions.

Hint: How does the exponent of periodicity changes after one compression step? What is the difference between pair compression and block compression?

Chapter 4

Basic string combinatorics (stringology)

4.1 Periodicity

Definition 4.1 (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 denote 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 4.2 (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 4.3. *A word w has a period of length p if and only if it has a border of length $|w| - p$.*

Fact 4.4. *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 4.1. 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 4.5 (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 4.6 (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 a 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 4.7 (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 4.8 (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.*

4.2 Failure function

Definition 4.9 (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 4.10. *Given a word w its failure function π_w can be computed in $\mathcal{O}(|w|)$ time.*

Example 4.2. Consider the word *aabaaba*. Then

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

4.3 Primitive words

Definition 4.11. A word $u \neq \epsilon$ is *primitive* if $u = w^k$ implies $w = u$ and $k = 1$

Example 4.3. Word $p = aabaa$ is primitive, so is a word $p' = aabaaabaa$. Note that p is border of p'.

Definition 4.12. We say that word u, v are *conjugate* if there are words p, q such that

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

Lemma 4.13. *Word u is primitive if an only if it is conjugates are pairwise different.*

Lemma 4.14. *Let u, u' be nonempty, conjugate words. Then u is primitive if and only if u' is primitive.*

Lemma 4.15. *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 4.16. *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 4.17. *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 ax exercise. It follows from Theorem 4.16 and simple application of the MP array. □

4.4 Suffix trees

Definition 4.18 (Trie)**.** A trie for a set of word $w_1, \ldots, w_k$ over an alphabet Σ is a tree whose edges are labelled with elements of Σ such that:

- each node has at most one edge to its child labelled with a given letter a;
- for every word w_i in the set there is a leaf such that the sequence of labels to this leaf forms w_i;
- all leaves have the property described above.

Definition 4.19 (Suffix tree)**.** Given a word $w[1\,..\,n]$ the suffix tree of w is a trie for the set of suffixes of $w\$$, where $\$$ is a special symbol outside the alphabet. Each path on which there are no branching nodes is compressed, i.e. represented as a single edge (labelled with a word obtained as a concatenation of labels on this path).

Lemma 4.20. *The suffix tree of $w\$$ has $|w|+1$ leaves and so also at most $|w|$ branching nodes.*

Theorem 4.21. *Suffix tree for w can be constructed on-line in time $\mathcal{O}(|w| \log |\Sigma|)$.*

It can be also constructed offline in time $\mathcal{O}(|w|)$ assuming that $\Sigma \subseteq \{0, 1, 2, \ldots, |w|^c\}$ for some constant c.

Definition 4.22 (LCP data structure for suffix tree)**.** The LCP da structure for a tree answers queries of the form: given two leaves in the tree, what is their lowest common ancestor.

Theorem 4.23. *Given a suffix tree, a structure for answering the LCP queries in $\mathcal{O}(1)$ and using $\mathcal{O}(|w|)$ memory can be constructed in $\mathcal{O}(|w|)$ time, in the standard RAM model.*

Fact 4.24. *The suffix tree for a word w together with the LCP supports queries for the longest common prefix of substring of w in $\mathcal{O}(1)$ time.*

Exercises

Task 28 Show equivalence of Lemma 4.5 and Corollary 4.6 and the equivalences of Lemma 4.8 and Corollary 4.8.

Task 29 Show that Lemma 4.7 follows from its variant in which $\gcd(|u|, |v|) = 1$.

Task 30 Prove Lemma 4.7, it may be easiest to prove Corollary 4.8 by adapting the proof of Corollary 4.6.

Task 31 (Alternative proof of Periodicity Lemma 4.5) 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 32 Show that u is primitive if and only if all its conjugates are pairwise distinct.

Let u, u' be conjugate. Show that u is primitive if and only if u' is primitive.

Task 33 Prove Theorem 4.16.

Task 34 Prove Theorem 4.17.

Task 35 Recall the linear-time construction of the MP array.

Chapter 5

Exponent of periodicity: general case

5.1 P-presentations

Definition 5.1. 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 $u = 0$.

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 does not depend on the sequences $\ell_1, \ldots, \ell_u$.

Theorem 5.2. *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 strings 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 5.2 is not overaly difficult, but it follows in a couple of steps.

Lemma 5.3. *For a given primitive word P the P-presentation of a word W exists and it is unique. It can be found greedily.*

Proof left as an exercise.

Lemma 5.4. *Given a primitive P and two words W, W' with presentations $(W), (W')$ the presentation of WW' is of one of the following forms:*

- (WW')
- (U, V) *and* $WW' = UV$
- (U, V) *and* $WW' = UPV$
- (U, Z, V) *and* $WW' = UZV$.

Proof left as an exercise.

Example 5.1. $P = aabaa$, $W = aabaaaaba = Paaba$, $W' = abaaaabaa = abaaP$. Then WW' has a P-presentation $(P, aabaabaa, P)$. And this is the last case in the Lemma 5.4

The Theorem follows from case inspection and Lemma 5.4 an is left as an exercise.

5.2 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.

The approach is similar as in the restricted case of exponent of periodicity. 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 $\{L_X, R_X\}_{X \in \mathcal{X}}$ 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 5.2 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 $S(u'X)$ uses fresh variables and adds equations that formalise the equalities between old and new variables, according to Theorem 5.2. 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 $S(R'a)$.

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 5.5. *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 5.6. *Each solution of $\mathcal{D}$ gives a solution of the word equation, obtained by replacing variables with their values in the P-presentations.*

Recall that in $\mathcal{D}$ there are at most $4|uv|$ equations whose at least one side that is other than the variables from $\{N_i\}_{i\in\mathbb{N}}$, take sides of those equations. Those terms can be grouped into parts representing those that are equalised by $\mathcal{D}$. This can be formalised as an equation with $8|uv|$ equations. Each side contains at most 2 variables and a constant at most 3.

Lastly, we argue as in Section 3.6: if there is a length-minimal solution then it has to correspond to a minimal solution of $\mathcal{D}$. So every component in it, so the variable L_X, R_X is at most exponential. For other variables are equal to one of those terms, they are at most exponential. If there are any other variables: we set them to 0.

Exercises

Task 36 Prove Lemma 5.4.

Task 37 Prove Lemma 5.3.

Task 38 Prove Theorem 5.2.

Chapter 6

Quadratic word equations

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

Algorithm 7 Lentin/Plotkin/Siekman 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
```

6.1 Analysis

It is easy to see that Algorithm 7 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 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 [30], but the argument is heavy in terms of geometry, so it will not be presented here.

Exercises

Task 39 Show that the algorithm for quadratic equations in fact yields a description of all solutions of such na equation.

Task 40 Consider a restricted class of word equations satisfying the following two conditions:

regular Each variable occurs at most once at each side;

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 of such equations is in NP.

Task 41 Give an algorithm that computes π_w for a given word w.

Chapter 7

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 [58]).

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 [14] and it runs in $\mathcal{O}(n^5)$ and returns a description of all solutions.

7.1 One variable equations

Clearly, the case of equations with only one variable is in P. Constructing a cubic algorithm is almost trivial, small improvements are needed to guarantee a quadratic running time. First non-trivial bound was given by Obono, Goralcik and Maksimenko, who devised an $\mathcal{O}(n \log n)$ algorithm [51]. This was improved by Dąbrowksi and Plandowski [15] to $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of occurrences of the variable in the equation. Furthermore they showed that there are at most $\mathcal{O}(\log n)$ distinct solutions and at most one infinite family of solutions. Intuitively, the $\mathcal{O}(\#_X \log n)$ summand in the running time comes from the time needed to find and test these $\mathcal{O}(\log n)$ solutions.

This work was not completely model-independent, as it assumed that the alphabet Σ is finite or that it can be identified with numbers. A more general solution was presented by Laine and Plandowski [32], who improved the bound on the number of solutions to $\mathcal{O}(\log \#_X)$ (plus the infinite family) and gave an $\mathcal{O}(n \log \#_X)$ algorithm that runs in a pointer machine model (i.e. letters can be only compared and no arithmetical operations on them are allowed); roughly one candidate for the solution is found and tested in linear time. Note that there is a conjecture that one variable word equations have $\mathcal{O}(1)$ solutions (plus the infinite family), in fact, an equation with 3 solutions outside the infinite family is not known.

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

7.2 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

$$A_0 X A_1 \dots A_{n_\mathcal{A}-1} X A_{n_\mathcal{A}} = X B_1 \dots B_{n_\mathcal{B}-1} X B_{n_\mathcal{B}} \ , \tag{7.1}$$

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 than 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

$$A_0 X A_1 \dots A_{n_X - 1} X = X B_1 \dots B_{n_X - 1} X B_{n_X} \ . \tag{7.2}$$

7.3 Via word combinatorics

Lemma 7.1. *Given a word equation $xp = qx$, if it is satisfiable then:*

- *p, q are conjugate and consequently also their 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 of the equation is of the form $(uv)^*u$.*

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

Proof is left as an exercise

7.3.1 $|A_0| \leq |B_1|$

Let B_0 be a prefix of B_1 of length A_0. Then $A_0X = XB_0$. Hence by Lemma 7.1 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 $s(X) = (uv)^j u$ and uv is primitive.

Lemma 7.2. *Given two words u, v such that uv is primitive solutions of (7.2) 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 $s_1, \dots, s_n$, ... either none, one or all are solutions.

Proof. We treat $s_0(X) = u$ separately. Using the MP table for u 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

A_0 is not a power of uv

> **Claim 7.2.1.** *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 $k+1$ uv and the beginning u' of the $k+2$ 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 7.2.2.** *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 7.2.1. 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 7.2.2 holds.

Since the uv-prefixes of both sides must be equal, we obtain an equation for i, i', 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 $s(X) = (uv)^j u$ and it can be easily tested in linear time using MP table. 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 7.3. *Given an equation* (7.2) *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.*

7.3.2 $|s(X)| \geq |A_0| - |B_1| > 0$

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 $A_0 \sqsubseteq s(X)$ *or* $s(X) \sqsubseteq A_0$ and $A' \sqsubseteq A_0$ and $|s(X)| \geq |A'|$, in any case $A' \sqsubseteq s(X)$. Thus $A_0 s(X) = s(X) B_1 A'$. The rest of the argument is as in the case above.

Lemma 7.4. *Given an equation* (7.2) *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.*

7.3.3 $|A_0| - |B_1| > |s(X)| > 0$

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 $(B_1 X)^2$ and the left with $B_1 A_0 X$. As $|B_1 s(X)| \leq |A_0|$ it follows that $(B_1 X)^2 \sqsubseteq B_1 A_0 A_0$.

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

$$B_1 = (vu)^j v \quad \text{and} \quad s(X) = (uv)^i u \tag{7.3}$$

Moreover, $P^2 \sqsubseteq B_1 A_0 A_0$. It is known that that there are at most $\mathcal{O}(\log |B_1 A_0 A_0|)$ such P and all of them can be found in linear $\mathcal{O}(|B_1 A_0 A_0|)$ time, see Theorem 4.17.

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 7.2 to test others in linear time. This in total yields $\mathcal{O}(n \log n)$ running time for the algorithm.

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 7.3.4.

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 $B_1 A_0 A_0$. We first show that at most one of them spans through the whole $B_1 A_0 A_0$.

Lemma 7.5. *Suppose that the* (vu)*-prefix of* $B_1 A_0 A_0$ *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 7.2.1 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 separately by Lemma 7.2.

So in the following we can assume that the vu-prefix of $B_1 A_0 A_0$ ends not later than after $|vu|$ letters of the second A_0; note that this is the same as the vu prefix of $B_1 A_0 s(X)$ as $|s(x)| \geq |uv|$ and $s(X) \sqsubseteq A_0$.

Lemma 7.6. *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 will also calculate the length of the P-prefixes of the right-hand side of the equation.

Lemma 7.7. *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, 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 7.6: 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 P we can establish the vu prefix of $s(\mathcal{B})$ in linear time: using an argument as in Claim 7.2.2 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 depend of $k|s(X)|$, 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 7.7 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 7.7.

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 7.2.2 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 7.6.

7.3.4 Verification of candidate solutions

Lemma 7.8. *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.*

7.4 Via recompression

If (7.2) 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 (7.2) 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 7.9. *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} (whichever is non-empty).*

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

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

Two comments are in place:

- Symmetric version of Lemma 7.9 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 7.16.

A simple proof is left as an exercise.

By TestSimpleSolution(a) we denote a procedure, described in Lemma 7.9, 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.

7.4.1 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 7.10. *Each solution s of equation of the form (7.2) 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. □

7.4.2 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.

7.4.3 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 $X b^r$), some solutions may be lost in the process and so they should be reported. We formalise these notions.

Definition 7.11 (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 $s(X) = u s'(X) v$ and $s(\mathcal{A}) = u s'(\mathcal{A}') v$;
- for each solution s' of $\mathcal{A}' = \mathcal{B}'$ the $s(X) = u s'(X) v$ is an unreported solution of $\mathcal{A} = \mathcal{B}$ and additionally $s(\mathcal{A}) = u s'(\mathcal{A}') v$.

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 7.11, 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.

7.4.4 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 7.12. $\mathsf{Pop}(a, b)$ *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 7.9.

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 $\mathsf{PreProc}$ Ensures that there are at most 2 crossing pairs

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

Lemma 7.13. $\mathsf{PreProc}$ *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 $\mathsf{PreProc}$. 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 7.9, 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

let a be the first letter of $s(X)$
report solution found by TestSimpleSolution(a) ▷ Excludes $s(X) \in a^+$ from further considerations.
let $\ell > 0$ be the length of the a-prefix of A_0
▷ By Lemma 7.9 $s(X)$ has the same a-prefix
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 $s(X) = w$
let b be the last letter of $s(X)$
report solution found by TestSimpleSolution(b) ▷ Exclude $s(X) \in b^+$ from further considerations.
let $r > 0$ be the length of the b-suffix of the B_{n_X}
▷ By Lemma 7.9 $s(X)$ has the same b-suffix
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 7.14. *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.

7.4.5 The algorithm

The following algorithm OneVarWordEq is basically a specialisation of the general algorithm for testing the satisfiability of word equations [26] and is built up from procedures presented in the previous section.

Algorithm 11 OneVarWordEq Reports solutions of a given one-variable word equation

while the first block and the last block are not blocks of a letter **do**
  Pairs ← pairs occurring in $s(\mathcal{A}) = s(\mathcal{B})$
  BlockComp ▷ Compress blocks, in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time.
  PreProc ▷ There are only two crossing pairs, see Lemma 7.13
  Crossing ← list of crossing pairs from *Pairs* ▷ There are two such pairs
  Non-Crossing ← list of non-crossing pairs from *Pairs*
  for each $ab \in$ *Non-Crossing* **do** ▷ Compress non-crossing pairs, in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$
    PairCompNCr(a, b)
  for $ab \in$ *Crossing* **do** ▷ Compress the 2 crossing pairs, in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$
    PairComp(a, b)
TestSolution ▷ Test solutions from a^*, see Lemma 7.16

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 by Lemma 2.9 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 7.15. *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 2.9

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 2.9, 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 (7.2) 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 7.10. 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 7.16. *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.

7.4.6 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 7.17. *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 7.9 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 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 7.13 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.

[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.

Lemma 7.18. *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 7.17 (this is at most $\mathcal{O}(n)$ by Lemma 7.15) and as Lemma 7.15 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 7.16 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 7.19. *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 7.17 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 7.15 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 $\mathcal{O}(n + n_X \log n)$. □

Exercises

Task 42 Given a word equation (7.2)

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

show that if the number of occurrences of variables on both sides are different (so $n_\mathcal{A} \neq n_\mathcal{B}$) then this equation has at most one solution and it can be easily given in linear time.

Task 43 Show that if in (7.2) the $A_{n_\mathcal{A}} \neq \epsilon$ and $B_{n_\mathcal{A}} = \epsilon$ then it has an equivalent equation in which $A'_{n_\mathcal{A}} = \epsilon$ and $B'_{n_\mathcal{A}} \neq \epsilon$.

Hint: First show that there is a system of equivalent equations and then concatenate them.

Task 44 Prove Lemma 7.1.

Task 45 Show Lemma 7.6.

Task 46 Suppose that given a word w we can construct in $\mathcal{O}(n)$ time a structure 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 (7.2); recall that n_X is the number of occurrences of the variable X in the equation.

Task 47 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 $A_{n_\mathcal{A}}$ or $B_{n_\mathcal{B}}$ (whichever is non-empty).

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

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

Task 48 Let $A_0 \in a^+$. Show how to compute all solutions of the equation in linear time.

Hint: Look for the first non-a symbol at both sides of the equation and recurse on the rest.

Task 49 Show that a word equation with 1 variable of length n has $\mathcal{O}(\log n)$ solutions and at most one infinite family of solutions of the form $\{w^k w' : k \geq 0\}$ and w' is a prefix of w.

Chapter 8

Word equations with two variables

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

8.1 Parametrised words

This chapter is based on [14].

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 and $|pq| \in \mathcal{O}(n)$; 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 $|pqp'| \in \mathcal{O}(n)$ and $a \in \mathbb{N}$; the size of this parametrised word is $|pqp'|$.

8.2 Canonisation

We say that a word equation with two variables is in a *canonised* form if its sides begin with different variables and end with different variables.

Lemma 8.1. *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. □

8.3 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 $|ABCD| \in \mathcal{O}(n)$.

8.3.1 $\mathcal{S}_1$

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

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

8.3.2 $\mathcal{S}_2$

Let additionally $|A| = |B| \leq |C| = |D|$. 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 $s(Y) < s(X)$.

8.3.3 $\mathcal{S}_3$

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

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

8.3.4 $\mathcal{S}_4$

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

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

8.4 Solving system $\mathcal{S}_1$

Lemma 8.2. *Given a system of equations $\mathcal{S}_1$, in time $\mathcal{O}(n^2)$ we can find a superset of subsitutuions 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.

Proof. First of all, from Lemma 7.1, we know that the set of solutions (for Y) for the second equation is $(CD)^*C$. After substituting this to the first equation we are left with the task of solving the equation

$$(CD)^i CAX = XB(CD)^i C \tag{$\mathcal{S}_1'$}$$

for each value of i. Clearly, if $|A| \neq |B|$ then there is no solution at all, so we deal only with the case that $|A| = |B|$.

If CD is the primitive root of CA then it is the primitive root of the whole $(CD)^i CA$, thus, again from Lemma 7.1 we conclude that $B(CD)^i C$ is conjugate to $(CD)^i CA$, so they have primitive roots of the same lengths and so DC is the primitive root of $B(CD)^i C$, as it is its suffix of appropriate length.

In this case the set of solutions (for $s(X)$) is exactly the $(CD)^*C$, which is a 1-parametric word (for $p = CD$ and $q = C$).

So in the following we deal with the case when CD is not the primitive root of $(CD)^iCA$, and so also DC is not the primitive root of $B(CD)^iC$. Let $v = (CD)^iCA$ and $w = B(CD)^iC$. Let also $m = 3 + \left\lceil \frac{|CA|}{|CD|} \right\rceil$. For each $i \leq m$ we consider the v and w separately. Note that their lengths (which are equal) are in $\mathcal{O}(n)$: $|(CD)^iCA| \leq m|CD||CA| \leq 2|CD|+3|CA|$. Since this is a conjugate equation $vX = Xw$, it has the set of solutions of the form $(pq)^*p$, where pq is the primitive root of v and qp of w. This is a 1-parametric word.

So consider now the case in which $i > m$. We first show that v is primitive (and so also w, as in order for the equation $vX = Xw$ to be satisfiable, the words v and w need to be conjugate). Suppose it is not and let q be the primitive root of v, in particular, it is a period of v. Consider the CD-prefix of v, it is at least $(CD)^i$, we want to show that it is longer than $q + |CD|$ and apply the periodicity Lemma:

$$\begin{aligned} i \cdot |CD| - q - |CD| &\geq (|v| - |CA|) - \frac{|v|}{2} - |CD| \\ &= \frac{|v|}{2} - |CA| - |CD| \\ &\geq \frac{(m+1)|CD| + |CA|}{2} - |CA| - |CD| \\ &\geq \frac{m|CD| - |CD| - |CA|}{2} \\ &\geq \frac{3|CD| + |CA| - |CD| - |CA|}{2} \\ &> 0 \end{aligned}$$

Thus q and CD are both periods of the CD-prefix, thus they are both powers of the same word, contradiction (as CD is not the primitive root of v).

Thus by Lemma 7.1 v and w are conjugates.

Let $v = gh$ and $w = hg$, then each solution is of the form

$$s(X) = v^k g \tag{8.1}$$

NOTE, the original paper claims to show that $|g| \geq |CD|$ and g has a period CD. Unfortunately, it not necessarily holds that $|g| \geq |CD|$, but if it does then g has a period CD. The case $|g| < |CD|$ has to be shown somehow differently.

We show that $|g| \geq |CD|$ and g has a period CD. Suppose first that $|g| < |CD|$. We inspect the prefix and suffix of v which has period $|CD|$. Details are left as an exercise. Similarly, knowing that g has length at least $|CD|$ we inspect the longest prefix and suffix of g that have period $|CD|$. Details are again left as an exercise.

Thus $|g| \geq |CD|$ and g has period CD. We intend to show that $g = (CD)^jC$ for some $j \geq 0$. Let us look at CDg. Since CD is a period of g, we know that $CDg[1\,..\,|g|] = g$. On the other hand $w = hg$ and the last $|CD|$ letters of w are DC. So $CDg[|g| + 1\,..\,g + |CD|] = DC$ and consequently $CDg = gDC$. Since CD is primitive, from Lemma 7.1 we obtain that $g = (CD)^jC$ for some $j \geq 0$. This brings us closer to the form from (8.1), but we also need to express h in a similar way. Consider the largest $a \geq 0$ such that g has a prefix $(CD)^{j+a}C$, and note that this value is independent of i, j, as it is the length of the DC-prefix of A; denote $v = (CD)^{j+a}CV$. Let now $h = (DC)^bW$ with $b \geq 0$ maximal possible. Then $v = gh = (CD)^jC(DC)^bW$ and so $i+a = j+b \Rightarrow j = i+a-b$. Thus in (8.1) we obtain

$$s(X) = (\underbrace{(CD)^{i+a}CV}_{v=gh})^k \underbrace{(CD)^{i+a-b}C}_{g}$$

thus all substitutions for X in the solution are a subset of

$$\begin{aligned} &\{((CD)^{i+a}CV)^k(CD)^{i+a-b}C \,:\, i,k \geq 0\} \\ &= \{((CD)^{i+b}CV)^k(CD)^iC \,:\, k \geq 0, i \geq a-b\} \end{aligned}$$

□

Exercises

Chapter 9

Equations without constants and related topics

9.1 General results

9.1.1 Equivalent subsystems

Definition 9.1. 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).

Theorem 9.2. *Every infinite system of word equations $\mathcal{E}$ in a finite number of variables has a finite subsystem $\mathcal{E}' \subseteq \mathcal{E}$ that is equivalent to $\mathcal{E}$.*

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.

9.1.2 Defect Theorem

Definition 9.3. A finite set $A \subseteq \Sigma^*$ of words is a *code*, if any word $w \in \Sigma^*$ has at most one representation

$$w = v_1 v_2 \cdots v_k$$

with $k \in \mathbb{N}$ and $v_1, \ldots, v_k \in A$.

Theorem 9.4 (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 9.1. 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 $uuvv = vvuu$.

9.2 An interesting new result/proof

This Section is based on [59].

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

$$X_0^k = X_1^k X_2^k \cdots X_n^k \text{ for } k = k_1, k_2, k_3 \tag{9.1}$$

with $0 < k_1 < k_2 < k_3$ being natural numbers.

Theorem 9.5. *A system* (9.1) *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 (9.1) has a nonperiodic solution, it has one over the binary alphabet.

Lemma 9.6. *If the system* (9.1) *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 $a_1 + a_2 + \dots + a_\ell$. Also, by $\mathrm{psw}_q(w)$ (partial sum word) we denote a sequence $q + a_1, q + a_1 + a_2, \dots, q + \sum(w)$ and think of if as a piecewise linear function that goes through the points $(0, q), (1, q + a_1), (2, q + a_1 + a_2), \dots$. 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 $\mathrm{psw}(ww') = \mathrm{psw}(w)\,\mathrm{psw}_{\Sigma(w)}(w')$, 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 9.7. *Given a solution s to a system of word equations* (9.1) *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 9.8. *Let s be a length-minimal among the nonperiodic solutionso of the system of equations* (9.1)*. 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 $V = \{v_1, \dots, v_\ell\}$ 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 9.5

proof of Theorem 9.5. Take the shortest non-periodic solution of the system of word equations (9.1). By Lemma 9.7 we can assume that $\sum(s(X_0)) = 0$ and by Lemma 9.8 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 $\sum(s(X_i)) = 0 = \sum(s(X_1 X_2 \cdots X_{i-1}))$ 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 $\mathrm{psw}(s(X_1)^k s(X_2)^k \cdots s(X_n)^k)$ 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 $s + \sum(s(X_i))$). □

9.3 Lyndon-Schützenberge Theorem

The presentation is based on [16].

The main goal is to prove the following Theorem, originally stated for the free groups [42], so in a slightly stronger form.

Theorem 9.9 (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 9.10. A word w is *bordered*, if there exists $\epsilon \neq v \neq w$ such that $w = vw'' = w'v$ for some w', w''.

Lemma 9.11. *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 9.12. *If w is a conjugate to a^m then there are words p, q such that $a = pq$ and $w = (qp)^m$.*

Definition 9.13. 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 9.14. *Lyndon word is not bordered.*

A simple proof is left as an exercise.

Lemma 9.15. *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 $w' \sqsubset w$. 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 7.1 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 9.16. *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 9.15.
- 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 9.15 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 9.9. 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 9.16 it follows that $b = c$; the proof is the same for $b = \epsilon$.

If $a = b$ then again we use Lemma 9.16 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 9.16 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 9.16 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 9.11, 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 9.14.

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 9.14 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 9.12 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. □

Chapter 10

Free groups

10.1 Free groups

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

$$aa^{-1} \to \epsilon, \quad a^{-1}a \to \epsilon \ . \tag{10.1}$$

Lemma 10.1. *Every word in* $(\Sigma \cup \Sigma^{-1})^*$ *has a unique normal form under the rewriting rules* (10.1).

A simple proof is left as an exercise.

Words as in Lemma 10.1 are called *reduced* or *irreducible*, for a word w this normal form is denoted by $\mathrm{IRR}(w)$.

Definition 10.2. 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 [44, 45]. 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 $Y = b^{-1}$.

10.2 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.

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

Theorem 10.3. *Given a system of equations over a free group it can transformed one can construct 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.

Chapter 11

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

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

where each Q_i is a quantifier and φ is a formula that uses only variables $x_1, x_2, \dots, x_k$, 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 an easy exercise to show that positive theory of a free semigroup is undecidable (exercise). On the other hand, the positive theory of a free semigroup is decidable, as shown by Makanin [45]. Below we show this result, in a variant given by Diekert and Lohrey [11], 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 (it provably has only a couple of letters that reduce) So in some sense it "is" a constant. Still, we need to allow the following variables to "use" this new constant, thus we allow Y_i to use $\{k_1, k_2, \dots, k_i\}$, 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 $Y = k^{-1}$.

11.1 Notation

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

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 $k_1, \dots, k_m$. 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.

11.2 Main result

The main result of this section is the following theorem.

Theorem 11.1. *Let $\mathbb{G}$ be a free group. Then for all $\vec{z} \in \mathbb{G}$ a positive formula with no free variables*

$$\varphi(\vec{Z}) = \forall X_1 \exists Y_1 \cdots \forall X_m \exists Y_m \varphi(X_1, \dots, X_m, Y_1, \dots, Y_m, \vec{z}). \tag{11.1}$$

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

$$\exists Y_1 \in \mathbb{G} * \langle k_1 \rangle \exists Y_2 \in \mathbb{G} * \langle k_1, k_2 \rangle \cdots \exists Y_m \in \mathbb{G} * \langle k_1, k_2, \ldots, k_m \rangle \varphi(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}). \quad (11.2)$$

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

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

11.3 Main technical Lemma

Lemma 11.2. *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:*

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

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 the $\vec{z}$.

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

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

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 $|r_i|$ 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 11.3. *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 og enough randomness notion is that

Lemma 11.4. *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 $P_1, \ldots, P_\lambda$, defined as

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

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 11.5. *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 11.6. *Let $\{x_j y_j = z_j\}_{j=1}^d$ be all nontrivial equations. The there is i such that for each j*

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

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

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

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

holds always.

proof of Lemma 11.2. Take s as the string substituted for X and all the witnesses $y_1, \ldots, y_m, \vec{y}$. We then take the rewriting system guaranteed to exist by Lemma 11.6 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 are too short to be rewritten;
- all equations that used to hold still hold by Lemma 11.6).
- if after the rewriting the equation holds then it held also originally (we can replace k back with ℓ to get the original equation). □

11.4 Main proof: quantifier elimination

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 11.7. *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). □

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 11.1 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

$$\forall X_1 \in \mathbb{G} \exists Y_1 \in \mathbb{G} \exists Y_2 \in \mathbb{G}_{[2..2]} \ldots \exists Y_m \in \mathbb{G}_{[2..m]} \varphi(X_1, k_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}), \tag{11.3}$$

In the following we equivalence of (11.3) and (11.2).

Lemma 11.8. *For φ positive if $\vec{z}$ satisfies* (11.2) *then it satisfies* (11.3).

Proof. We use Lemma 11.7.

Take any $x_1 \in \mathbb{G}$. Take a homomorphism $h : \mathbb{G}_{[1..m]} \to \mathbb{G}$ 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 $y_i \in G[1\,..\,i]$ such that $\varphi(k_1, \ldots, k_m, y_1, \ldots, y_m, \vec{z})$ holds. Then Lemma 11.7 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 (11.3) holds, as claimed. □

Lemma 11.9. *For φ positive if $\vec{z}$ satisfies* (11.3) *then it satisfies* (11.2).

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 (11.3) is equivalent to:

$$\forall X_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_1 \in \mathrm{IRR}(\mathbb{M}) \exists Y_2 \in \mathrm{IRR}(\mathbb{M}_{[2..2]}) \ldots \exists Y_{[2..m]} \in \mathrm{IRR}(\mathbb{M}_{[2..m]}) \exists \vec{Y} \in \mathrm{IRR}(\mathbb{M}_{[2..m]})$$
$$\varphi'(k_1, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}, \vec{Y}) \ .$$

By the Lemma 11.2 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

$$\exists Y_1 \in \mathbb{G}_{[1..1]} \exists Y_2 \in \mathbb{G}_{[1..2]} \ldots \exists Y_m \in \mathbb{G}_{[1..m]} \varphi(s_1 k_1 s_2, \ldots, k_m, Y_1, \ldots, Y_m, \vec{z}) \ . \tag{11.4}$$

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 11.11). Since it is an isomorphism, we can apply it on (11.4), see Lemma 11.10 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 (11.2). □

Lemma 11.10. *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 $i(\mathbb{G}_j) = \mathbb{G}_j$. Show that*

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

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 11.11. *Let $\mathbb{G} = \langle c_1, \ldots, c_m \rangle$ be a free group and consider $h : \mathbb{G} \to \mathbb{G}$ defined as $h(c_1) = g c_1 g'$ where $g, g' \in \langle c_2, \ldots, c_m \rangle$. Show that 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 11.8 and 11.9 give the proof of Theorem 11.1.

Exercises

Task 50 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 form 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 51 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).

Hint: First make the claim about the whole theory and then eliminate the negation as we did before.

Task 52 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 53 (Newman's lemma) A rewriting system $S = \{(\ell_i, r_i)\}_{i \in I}$ is called length-reducing if $(\ell, r) \in S$ implies $|\ell| > |r|$. S is called 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 $t, u \to_S^* v$.

Show that if S is length-reducing, then S is confluent if and only if it is local confluent.

Task 54 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 55 Prove Lemma 11.10.

Task 56 Prove Lemma 11.11.

Chapter 12

Solving equations in free groups

By Theorem 10.3 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 11 also ask for constraints of the form "w does not use letter a"). In general we will do this with the regular constraints.

12.1 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 $n \times n$.

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. The involution can be the inverse on the Boolean matrices, 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.

12.2 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, as we can initially non-deterministically guess the appropriate transition from a set of transitions.

Lemma 12.1. *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*

12.3 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).

12.4 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.

12.5 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.

12.6 Needed modifications

12.6.1 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 $\rho(X) = \rho(\epsilon)$.

12.6.2 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.

12.6.3 Pair compression

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

12.6.4 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.

12.6.5 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.

12.7 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 12.2. 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 12.1. 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 12.3. *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 12.4. 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 12.5. *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 12 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' ∪ Σ̄')
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

$$\mathrm{w}(s,\alpha) = |U| + |V| + 2\sum_{X\in\mathcal{X}} |UV|_X |\alpha(s(X))| \ , \tag{12.1}$$

Lemma 12.6. *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 12.7. *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 (12.1) 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 12.8. *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 57 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 $\bar{\ }$ 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.

Task 58 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 we define the involution 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 each set regular in $\mathbb{B}_{n\times n}$ is regular in $\mathbb{B}_{m\times m}$ (but not necessarily vice-versa).

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

Task 59 (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 13

Linear Monadic Second Order Unification

This Chapter is more or less based on [39], 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 13.1.

$$Xa\perp = Yb\perp$$

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 13.1. *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 13.2. *Satisfiability of a a linear monadic second order unification is in* NP.

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

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

Lemma 13.3. 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 13.4. *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 on ly the case of left sides, as we are only interested in that.

Lemma 13.5. *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 algorithm shall eliminate one variable using polynomially many steps, each of those steps increases the size of the instance by $\mathcal{O}(n)$. This guarantees that the whole algorithm runs in NP: after the removal the instance is of polynomial size. In polynomially many steps we reduce it to size $\mathcal{O}(n^2)$ and then iterate again, with less variables.

Example 13.2. Suppose that we have only one equation $u = v$. If after applying Lemma 13.5 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 13.6. 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 13.7. *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 13.8. *If there are two edges from* X *labelled with nonempty words then they have the same first letter* or $s(X) = \bot$ or $s(X) = \epsilon$ *for each solution* s.

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

Lemma 13.9. *If X is a minimal element of $<$, so are all its equivalent variables.*

Lemma 13.10. *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 $\bot$).*

The proof is obvious.

Lemma 13.11. *Let s be a solution, $X = X_1$ be a minimal element according to $<$ (in particular there is no self-loop from X to X with non-empty label on it) and let $X_1, \ldots, X_m$ be all variables equivalent to X_1, assume that $s(X_i) \notin \{\epsilon, \bot\}$ and let a be the first letter of $s(X)$. Then after left-popping a from all $X_1, \ldots, X_m$ either one variable is removed*

- *all $X_1, \ldots, X_m$ are still equivalent*
- *each edge $X_i \xrightarrow{w} Y$ for $w \neq \epsilon$ is replaced with $X_i \xrightarrow{a^{-1}w} Y$.*

Proof. For every edge $X_i \xrightarrow{w} Y$ we change the label to $X \xrightarrow{a^{-1}w} Y$ and for every edge $Y \xrightarrow{w} X_i$ we change the label to wa (in particular, for ϵ edges we are left with ϵ).

Note that for the second claim it cannot be that $X \sim Y$ as then there would be a loop from X to X with label w on it. □

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).

We then look at the dependence graph. If there is no edge labelled with non-empty word then we set $s(X) = \bot$ for each variable, this is a solution. If $<$ is acyclic and there is an edge then we find a minimal element (say X) which has an out-going edge labeled with a non-empty word (we go back by edges to find it). Say a as the first letter on this label, we left-pop a from each $Y \sim X$. By Lemma 13.11 the ϵ edges are preserved (as we either pop from both their ends or from none), and so the variables that were equivalent to X stay equivalent. 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.

Since X is minimal, there are no incoming edges and so by Lemma 13.11 the total sum of length of labels decreases. Initially it was $\mathcal{O}(n^2)$, as each label occurs somewhere in the equations, so we end up after at most quadratic number of steps. We could also end up with one variable removed, which simply ends the phase.

When we finish with popping, either there is no edge with nonempty label, so all variables are equivalent and the equation

If there is a cycle in the dependency graph from X to X that defines a nonempty label then we take the shortest (in terms of number of edges) such a cycle, let it be from X to X and let the word on it be w_X. Note that $|w_X| = \mathcal{O}(n^2)$: the cycle cannot have repeating nodes, so also there are no repeating edges, so each label is used at most once and their concatenation is not longer than the current equation

We apply $\mathcal{O}(n^2)$ pair compressions which reduce w_X to a sequence of the form a^k. This is always possible: unless w_X is of desired form, it has two different consecutive labels, say ab. Then we make that ab compression and we proceed; there are quadratically many such compressions. We need to ensure, though, that indeed making the compression affects the label on the path in the appropriate way.

We consider the effect of popping and pair compression on the dependency graph and our chosen cycle from X to X; if somewhere during the procedure some variable is removed then we are done.

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

Thus after the compression we have a cycle from X to X such that each of its edges is labelled with a power of a (perhaps ϵ). We claim that in each solution of this system of equations there is at least one variable Y on this cycle which has a substitution $s(Y) = a^\ell(\bullet)$ (in particular, it does not forget the argument): if some of those variables Y has $s(Y) = \epsilon$ then we are done. For any variable with non-trivial outgoing edge on this cycle it begins with a; for other variables this we have that one of them is linked via ϵ-edges to a variable whose first letter is a, so all of them begin with a. Now imagine that we left-pop a from all variable on the cycle. Then the cycle is preserved, in particular, all variables still begin with a. This can terminate only when for one of them we have that it is $s(Y) = \epsilon$, so originally $s(Y) = a^\ell(\bullet)$ for some ℓ.

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

All of that can make the equation of size at most $\mathcal{O}(n^3)$, as each of quadratically many uncrossings introduce up to $\mathcal{O}(n)$ letters.

Exercises

Task 60 Show that the variant of monadic second order unification considered here is NP-hard.

Task 61 Prove Lemma 13.4.

Task 62 Prove Lemma 13.5.

Task 63 Prove Lemma 13.7.

Chapter 14

Compressed pattern matching: Combinatorial approach

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

The presented algorithm computes occurrences of a pattern given by an SLP $\mathcal{P}$ within the text given by the SLP $\mathcal{P}$. This is based on [41].

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

14.1 AP table

Lemma 14.1 (Basic Lemma)**.** *For any pattern p and a position α in a string t all occurrences of p in t touching α form a single arithmetical progression.*

Proof. If there are at most 2 occurrences then we are done. Otherwise take any three occurrences, let the second be offsetted by n from the first and the third by n' from the second, then n, n' are periods of p. As they touch a single position in t, we have that $n + n' \leq |t|$ and so by Periodicity Lemma $\gcd(n, n')$ is also a period and so there are occurrences offsetted by a multiplicity of $\gcd(n, n')$ from the first occurrence; they form an arithmetic progression. If there is an occurrence outside of this arithmetic progression then we can take the first and second occurrence in this progression and this occurrence outside. By the reasoning as above they all are in a single arithmetic progression; it is easy to see that this progression contains the whole previous arithmetic progression, thus this procedure terminates. □

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

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

Filling $AP[1, j]$ and $AP[i, 1]$ is easy: the former tells whether one/two letter that touch the cut in $\mathrm{val}(T_i)$ are equal to $\mathrm{val}(P_1) \in \Sigma$ and the latter tells whether $\mathrm{val}(P_i) = \mathrm{val}(T_1) \in \Sigma$. Then we fill $AP[i, j]$ in lexicographic order on $[i, j]$.

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

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

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

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

14.1.1 Filling AP using LSP

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

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

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

For the continental endings note that the corresponding occurrences of P_s are all within shifted occurrences of P_r, so due to periodicity either all continental occurrences of P_r extend by P_s or none. Thus we check one, using one local search. Note that this can be done easier, but we do not care.

For the seaside endings, let δ be the last ending of the P_r in the sequence. Then we can use the $LSP(s, j, [\delta - |\,\mathrm{val}(P_s)|, \delta + |\,\mathrm{val}(P_s)|])$. Thus we obtain 2 arithmetic progressions representing the occurrences of P_s. We can intersect them with the arithmetic progression representing endings of P_r in constant time (Task 65), which is again an arithmetic progression.

As a last step we merge the obtained arithmetic progressions. Note that we know that they form an arithmetic progression by Lemma 14.1.

For the running time: note that filling $A[i, j]$ takes $\mathcal{O}(j) \leq \mathcal{O}(n)$ time, so the total time is $\mathcal{O}(mn^2)$.

14.2 Local search procedure

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

It is easy to check, that the total recursion time is $\mathcal{O}(j)$: for an interval $[\alpha\,..\,\beta]$ there are two recursive calls, correspondign to the intervals $[\alpha\,..\,\gamma]$ and $[\gamma\,..\,\beta]$, where γ is the cut. If one of them is empty then we are done, and otherwise the lengths of those intervals is the same as the original one; bue we assume that $|\beta - \alpha| \leq 3|\,\mathrm{val}(P_i)|$ and termiante immediately when the interval is shorter than $|\,\mathrm{val}(P_i)|$; thus there are at most three paths in the recursive calls.

Lastly, we merge the resulting arithmetic progressions. It is easy to check that two such arithmetic progressions either are disjoint or have at most the first/last element in common. Thus we can merge them in constant time per item.

Concerning the bound on two arithmetic progressions, fix two positions in $[\beta\,..\,\alpha]$: $\alpha + |\,\mathrm{val}(P_i)|$ and $\beta - |\,\mathrm{val}(P_i)|$. As $\beta - \alpha \leq 3|\,\mathrm{val}(P_i)|$, each occurrence touches at least one of those two positions. And Lemma 14.1 gives that those touching a fixed of those positions form an arithmetic progression. This proof also shows that if we merge the 5 arithmetic progressions obtained from the recursive calls then we can divide those into two groups (that touch a fixed of those positions) and in each groups we can merge the arithmetic progressions.

Exercises

Task 64 Prove the basic lemma: All occurrences of val($\mathcal{P}$) in val($\mathcal{T}$) overlapping any given position form a single arithmetical progression.

Task 65 Give a procedure for intersecting two arithmetic progressions represented as triples (*first, step, end*), i.e. the first element, the step of the arithmetic progression and the last element.

Task 66 Show that the recursion in the $LSP(i, j, [\alpha, \beta])$ has at most 3 active branches (so one that do not terminate immediately). Deduce from this that the running time of $LSP(i, j, [\alpha, \beta])$ indeed takes $\mathcal{O}(j)$ time.

Task 67 Show that the result of the Local Search Procedure is always a collection of at most two arithmetic progressions.

Hint: Basic Lemma

Chapter 15

Equality testing for dynamic strings

An approach that somehow paved the way to recompression was introduced by Mehlhorn, Sundar, and Uhrig [47] in their work on data structures for equality testing of dynamic strings, see also [20] for a different presentation and [20] for a simpler randomised variant.

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

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

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

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

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

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

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

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

Theorem 15.1. *There exists an implementation of the data structure that supports the above operations in times:*

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

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

Equal $\mathcal{O}(1)$

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

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

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

The signature is built using two alternating operations:

- block encoding: the first one replaces each block a^ℓ with (a, ℓ) (treated as one symbol)
- the second groups the letters into segments of length between 2 and 12 and then replaces the segments with new symbols

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

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

This is based on the following marking algorithm

Lemma 15.2. *For any string of numbers (whose two consecutive elements are different) with values in* $\{0, \ldots, N\}$ *there is a function that assigns to each element* 0 *or* 1 *such that*

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

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

15.1 How to calculate assignment

This is based on [7, 21].

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

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

Lemma 15.3. *This procedure produces a valid* 6*-coloring and has* $\mathcal{O}(\log^* N)$ *many iterations.*

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

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

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

Then we make the assignment of 0 and 1 by assigning 1 to the nodes whose colours are local maxima. It is easy to check that it has the desired properties.

15.2 Storing

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

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

15.3 Update

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

To split a signature, we go in the SLP to appropriate position, we store $\mathcal{O}(\log^* N)$ elements from each side of the path on each level, when this is the assignment, or the length of the a-prefix/suffix,

when the level calculates the blocks compression. Since the signatures are computed locally, this is enough to recalculate the signature. (Exercise)

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

15.4 Comments

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

Exercises

Task 68 Show how to implement the split operation in the data structure for equality testing of dynamic texts. In particular show that it is enough to modify only $\mathcal{O}(\log^* N)$ elements per level, where N is the bound on the maximal size of the numbers occurring in the signature.

Task 69 Show that $\log^* N = \mathcal{O}(\log^*(\max(m, |\Sigma|)))$, where Σ is the input alphabet and m is the number of performed operations.

Hint: You can use the bound calculated in the previous task, assuming that it also applies to concatenation.

Task 70 Can you apply the signature-building algorithm from the data structure for equality of dynamic texts to word equations? What assumptions do you need? What is the size of the equation?

Bibliography

[1] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, William F. Smyth, German Tischler, and Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.*, 45(1):5, 2012.

[2] Stephen Alstrup, Gerth S. Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In David B. Shmoys, editor, *SODA*, pages 819–828. ACM/SIAM, 2000. ISBN 0-89871-453-2. doi:doi.acm.org/10.1145/338219.338645. URL `http://dl.acm.org/citation.cfm?id=338219.338645`.

[3] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993. doi:10.1137/0222017.

[4] Witold Charatonik and Leszek Pacholski. Word equations with two variables. In Habib Abdulrab and Jean-Pierre Pécuchet, editors, *IWWERT*, volume 677 of *LNCS*, pages 43–56. Springer, 1991. ISBN 3-540-56730-5. doi:10.1007/3-540-56730-5_30.

[5] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

[6] Gang Chen, Simon J. Puglisi, and William F. Smyth. Fast and practical algorithms for computing all the runs in a string. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 307–315. Springer, 2007.

[7] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7. URL `http://dx.doi.org/10.1016/S0019-9958(86)80023-7`.

[8] 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`.

[9] 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`.

[10] Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *DCC*, pages 482–488. IEEE Computer Society, 2008.

[11] 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`.

[12] 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`.

[13] 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.

[14] Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *LNCS*, pages 408–419. Springer, 2004. ISBN 3-540-22849-7. doi:10.1007/978-3-540-27836-8_36.

[15] Robert Dąbrowski and Wojciech Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011. doi:10.1007/s00453-009-9375-3.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In Artur Czumaj, editor, *SODA*, pages 1509–1528. SIAM, 2018. ISBN 978-1-61197-503-1. doi:10.1137/1.9781611975031.99. URL https://doi.org/10.1137/1.9781611975031.99.

[21] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988. doi:10.1137/0401044. URL http://dx.doi.org/10.1137/0401044.

[22] 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.

[23] Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC*, pages 133–142. IEEE, 2013. ISBN 978-1-4673-6037-1.

[24] Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC 2014*, pages 163–172. IEEE, 2014. doi:10.1109/DCC.2014.62. URL http://dx.doi.org/10.1109/DCC.2014.62.

[25] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027. URL http://dx.doi.org/10.1016/j.tcs.2015.05.027.

[26] 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.

[27] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

[28] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013. ISBN 978-3-642-38904-7, 978-3-642-38905-4.

[29] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *CPM*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001. ISBN 3-540-42271-4. doi:10.1007/3-540-48194-X_17.

[30] Olga Kharlampovich, I. 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`.

[31] 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`.

[32] Markku Laine and Wojciech Plandowski. Word equations with one unknown. *Int. J. Found. Comput. Sci.*, 22(2):345–375, 2011. doi:10.1142/S0129054111008088.

[33] 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.

[34] 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`.

[35] 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`.

[36] 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`.

[37] 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`.

[38] 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`.

[39] 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`.

[40] 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`.

[41] Yury Lifshits. Processing compressed texts: A tractability border. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 228–240. Springer, 2007. ISBN 978-3-540-73436-9. doi:10.1007/978-3-540-73437-6_24. URL `http://dx.doi.org/10.1007/978-3-540-73437-6_24`.

[42] 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.

[43] Gennadiĭ Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).

[44] 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).

[45] Gennadiĭ Semyonovich Makanin. Decidability of the universal and positive theories of a free group. *Izv. Akad. Nauk SSSR*, Ser. Mat. 48:735–749, 1984. In Russian; English translation in: *Math. USSR Izvestija, 25*, 75–88, 1985.

[46] 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`.

[47] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

[48] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. ISBN 9780521835404.

[49] 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`.

[50] 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`.

[51] S. Eyono Obono, Pavel Goralcik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *MFCS*, volume 841 of *LNCS*, pages 336–341. Springer, 1994. ISBN 3-540-58338-6. doi:10.1007/3-540-58338-6_80.

[52] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *CPM*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011. ISBN 978-3-642-21457-8.

[53] 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`.

[54] 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`.

[55] 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`.

[56] RTA problem list. Problem 90. http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/90.html, 1990.

[57] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

[58] 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.

[59] 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`.

[60] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005. doi:10.1016/j.jda.2004.08.016.

[61] Manfred Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universität, 1994.

[62] 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`.

[63] 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`.

[64] 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`.

[65] 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`.

[66] 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`.

[67] 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`.

[68] 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`.

[69] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 30–39. ACM, 1978.

[70] 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`.

[71] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-65367-8.

[72] 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.

[73] 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`.

[74] 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`.