Combinatorics on Words

Suppression of Unfavourable Factors in Pattern Avoidance

Veikko Keränen

Rovaniemi University of Applied Sciences
Jokiväylä 11
96300 Rovaniemi
Finland
veikko.keranen@ramk.fi

We explain extensive computer aided searches that have been carried out over 15 years to find new ways of constructing abelian square−free words over 4 letters. Indeed, these structures have turned out to be very rare and hard to find. Basically, only two examples, and slight modifications based on them, have been found so far. We have also encountered highly nonlinear phenomena which considerably affect our everyday calculations and usually make their accomplishment hard. However, quite recently, we have gained new insight of why these structures are so very rare. Consequently, the present work has a potential to make the future explorations easier. The rareness of long words avoiding abelian squares can be explained, at least partly, by using the concept of an unfavourable factor. The purpose of this paper is to describe the utilisation of *Mathematica* in searching and suppressing these factors. In principle, the same code can be used with slight modifications for other kinds of patterns as well.

■ Introduction

In the year 1961, Paul Erdös [E] raised the question whether abelian squares can be avoided (as factors) in infinitely long words (also called strings). An abelian square is a non−empty word uv, where u and v are permutations (anagrams) of each other. In 1969, Pleasants [Pl] solved positively the question by Erdös in the case of a five letter alphabet, but the four letter case remained open till 1992 when we presented an abelian square−free (a−2−free) endomorphism g_{85} over the four letter alphabet $\Sigma_4 = \{a, b, c, d\}$. This endomorphism g_{85} was found after long computer experiments, and, until quite recently, all known methods for constructing arbitrarily long a−2−free words on Σ_4 have been based on the structure of this g_{85}.

In 2002, after over 11 years of exhaustive searches, we found a completely new endomorphism g_{98} of Σ_4^*, the iteration of which produces an infinite abelian square−free word. The endomorphism g_{98} is not an a−2−free endomorphism itself, since it does not preserve the a−2−freeness of all words of length 7. However, g_{98} can be used together with g_{85} to produce a−2−free DT0L−languages of unlimited size. Nowadays, abelian square−free words have also found their way to applications, for instance, in number theory, algorithmic music, and only recently in cryptography (Rivest [R] in 2005).

In spite of these findings and applications, our understanding of a−2−free words has not advanced very much at all contrasted to the situation of ordinary repetition−free words. Excluding g_{85}, Carpi's [C2] modification of it, and g_{98}, every systematic attempt for constructing long abelian square−free words over 4 letters has failed. Moreover, all the succesful constructions have been found only by exhaustive computerised searches

through extremely large number of candidates. There is also an important analogous avoidability problem for the 3 letter case in which one allows short abelian repetitions of the form xx or xxx for a letter $x \in \Sigma_3 = \{a, b, c\}$. This open problem was posed by Sami Mäkelä [Mä] in 2002.

Quite recently, however, we have gained new insight of why these structures are so very rare. We are now able to explain, at least partly, this rareness of long words avoiding abelian squares by using the concept of an unfavourable factor. The purpose of the paper is to describe the utilisation of *Mathematica* in searching and suppressing these factors. In principle, the same code can be used with slight modifications for other kinds of (possibly non−abelian) patterns as well.

We have carefully studied the options to make the code efficient with regards to running time and the required memory space. At the same time, the structures need to be flexible for further development. Indeed, it is expected that this code will be run interactively for a quite long time. The code is also likely to be transformed into various computational environments (such as GRID, C++, FPGA). This work has already been started. In the process of these computations, efficiency will still be increasing due to the natural reduction process. Indeed, the long lists of words that we now have to use, can most likely be considerably reduced as we move on to study longer words.

The code and some of the structures involved are quite complicated and would have been really hard to develope without using *Mathematica*. We have been using integer coding and cumulative integer lists for words incorporated into quite extensive precomputations. Moreover, certain symbolic representations for words and *Mathematica*'s pattern matching properties (for lists and strings) have been extremely useful for us.

At the same time, we feel that it would be really great for the programmability and computational efficiency, if in *Mathematica* there was an efficient way of saying, for example, that when matching the pattern **{___,u__,v__}**, we are actually interested only in those cases of **u** and **v** in which their lengths are equal. Indeed, our words can contain thousands of letters, and the absence of restricted pattern matching led us to write part of the code in a quite tedious C-like fashion. Fortunately, to our big relief, debuging turned out to be a comfortable process.

We define an *unfavourable* (or *forbidden*) word or factor to be an abelian square−free (a−2−free) word over a fixed alphabet Σ (in our case $\Sigma = \Sigma_4 = \{a,b,c,d\}$, or $\Sigma = \Sigma_3 = \{a,b,c\}$ in which case we allow xx or xxx for a letter x) to be an a−2−free word over Σ which cannot anyhow occur as a proper factor inside any infinite a−2−free word. That is to say, over Σ, an unfavourable a−2−free word cannot be continued infinitely long to the left and to the right without necessarily creating an abelian square at some point. However, it might well be possible to extend such a word boundlessly to one direction, say to the right, without producing any abelian squares. Experiments support this conjecture but the existence of such unfavourable factors remains an open question.

In what follows, we explain shortly our search for unfavourable factors. Let the alphabet Σ be fixed and consider words over it. We take a word (finally we actually need to consider all the a−2−free words of a given length) and try to extend it in a−2−free fashion to the right and to the left with all possible ways up to a given upper bound for the total length. At a time, the length of the word increases only by a given fixed length. We extend alternately to right and left, and backtrack if need be. If the upper bounds are reached then the original word is a *so−far−favourable* one (it may still turn out to be unfavourable on later experiments). If there is no way to reach the upper bounds, then the original word is classified, without any doupt, to be unfavourable. Thus for a given length we obtain three kinds of words: *unfavourable* (*bad*), *so−far−favourable* (*so−far−so−good*), and

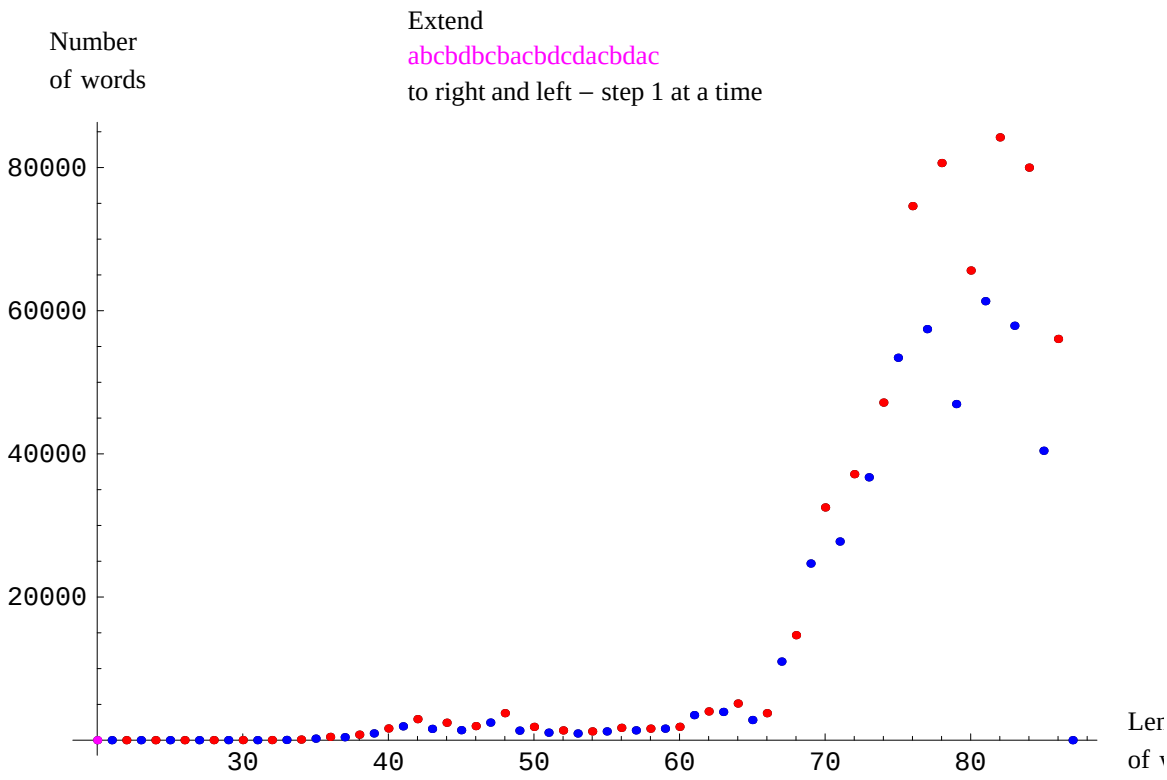
favourable (*good*). At present, the latter type consists of words occurring as factor in a−2−free words obtained by using g_{85}, g_{98}, and Carpi's modification of g_{85}.

It is a remarkable phenomenon that already relatively short so−far−favourable words turn out to be unfavourable factors after being 'safely' extendable (to right and left) for quite a long distance (and with a huge number of branches). One might have expected the quite long buffers to be sufficient for further grow. Due to Carpi [C2], we know that the number of a−2−free words over four letters grows exponentially ($\geq 1.000021^n$) with respect to word length n. But how do these words grow then? We conjecture that in spite of the exponential growth, the ratio between the number of properly extendable, i.e., favourable, words of length n, and between the number of unfavourable factors of the same length, actually tends to zero when the length n tends to infinity! Thus we suspect that the vast majority of a−2−free words over four (and, arguably, three) letters cannot occur as proper factors in the middle of very long (infinite) words. In a way, this would also explain why it is so extremely difficult to find a−2−free endomorphisms over four letters. At present we know, for example, that just a little bit over half (50.5737 %) of the a−2−free words over Σ_4 of length 20 are indeed unfavourable. In the future, this and other similar observations could lead to discovery of new (now so hard to find) abelian square−free endomorphisms over Σ_4. This, in turn, could lead to a better understanding of the structures involved. For example, the new a−2−f endomorphisms could allow to establish sharper bounds for the growth of the number of a−2−f words over Σ_4.

As an example, in the case of 4 letters, the following words together with their permutations and mirror images are unfavourable (forbidden) factors:

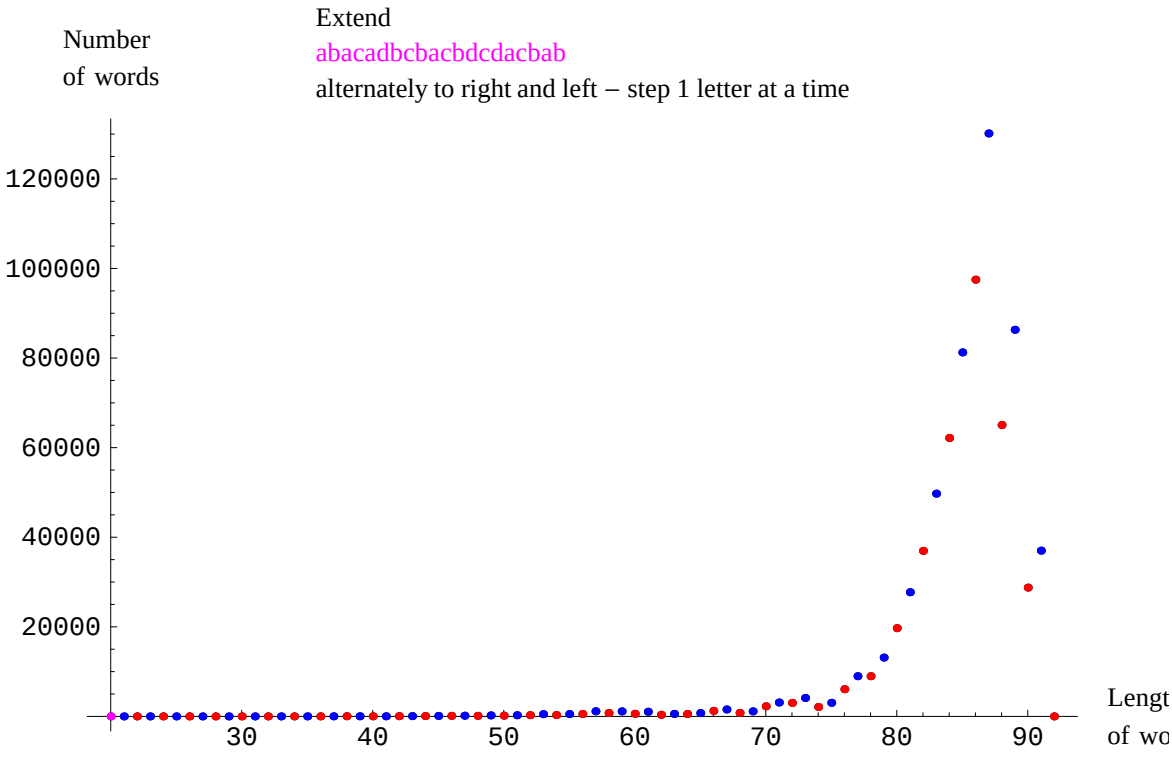
```
unfavourableFactorsOfLength8
{"abacdaba","abacdbab","abcabdab","abcabdcb","abcadbad"};

someUnfavourableFactorsOfLength20
{"abacabadbacbcdbacdbd","abacabdcadcdbcbacbdc","abacadcabcbdcabdcdad","ab
acdabcbdbcadabdacd",
"abcabadbcadbdcbcabcd","abcabadcbadbdcbcabcd","abcacdabcdadbabcabda","ab
cacdabdcadbabcabda",
"abcacdbacdadbabcabda","abcacdcbabdcdacdcbad","abcbabcdcacdbdabacbc","abc
babdabcacdabcdadb",
"abcbadbcacdcabdbcdba","abcbadcbdcdacabcadcb","abcbdbcbacbdcdacbdac","abc
dbdadcbcadabdadcb"};
```

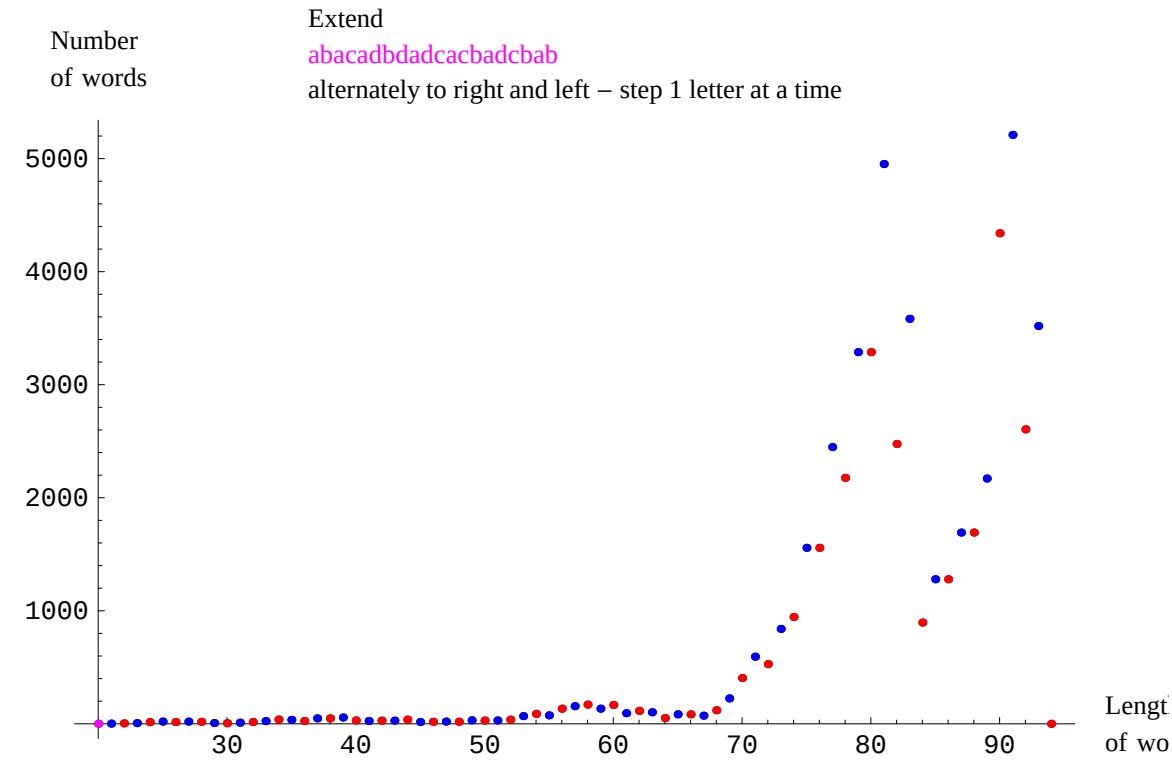
The latter words form a very interesting case, leading to a million−fold fluctuational phenomenon (with respect to needed running time, and memory space, if we wish to see the tree of all possibilities). This, actually quite frequently appearing, case is illustrated (in a bit different setting than the one we use elsewhere in this paper) online at [K5]. For example, there is the following illustration for "**abcbdbcbacbdcdacbdac**". Note the top at (82, 84218) and the death of all branches at word length of 87:

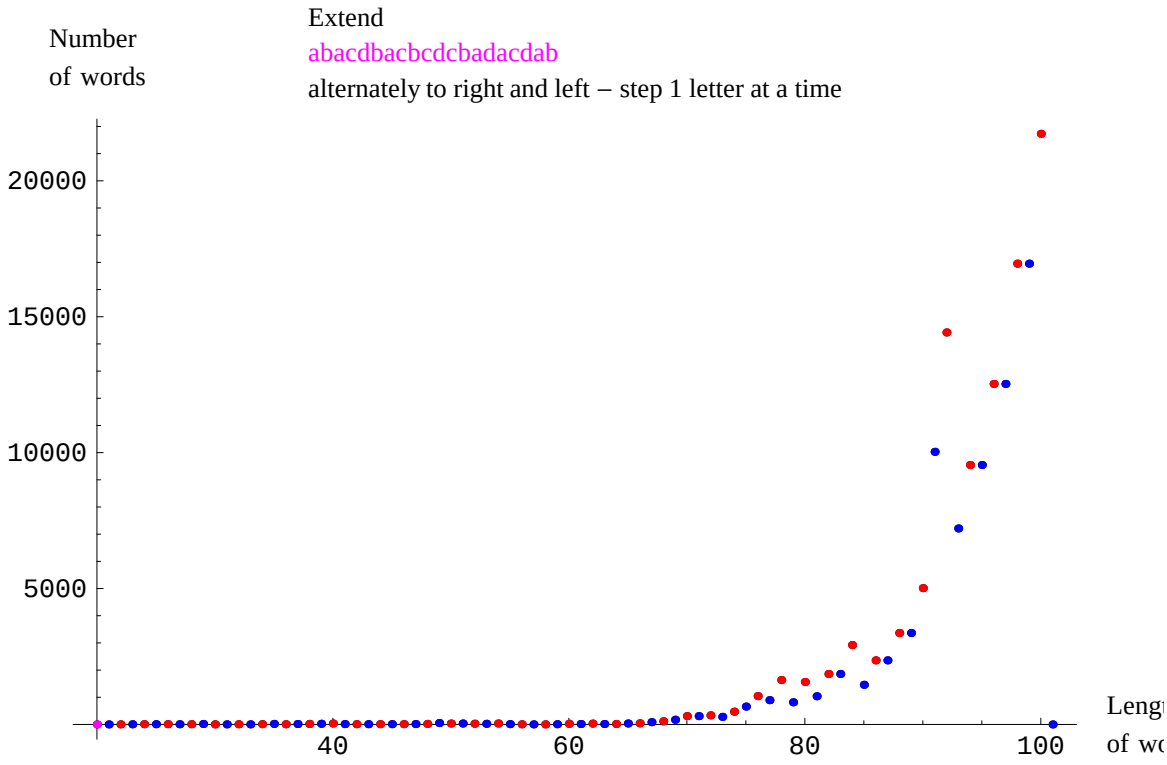


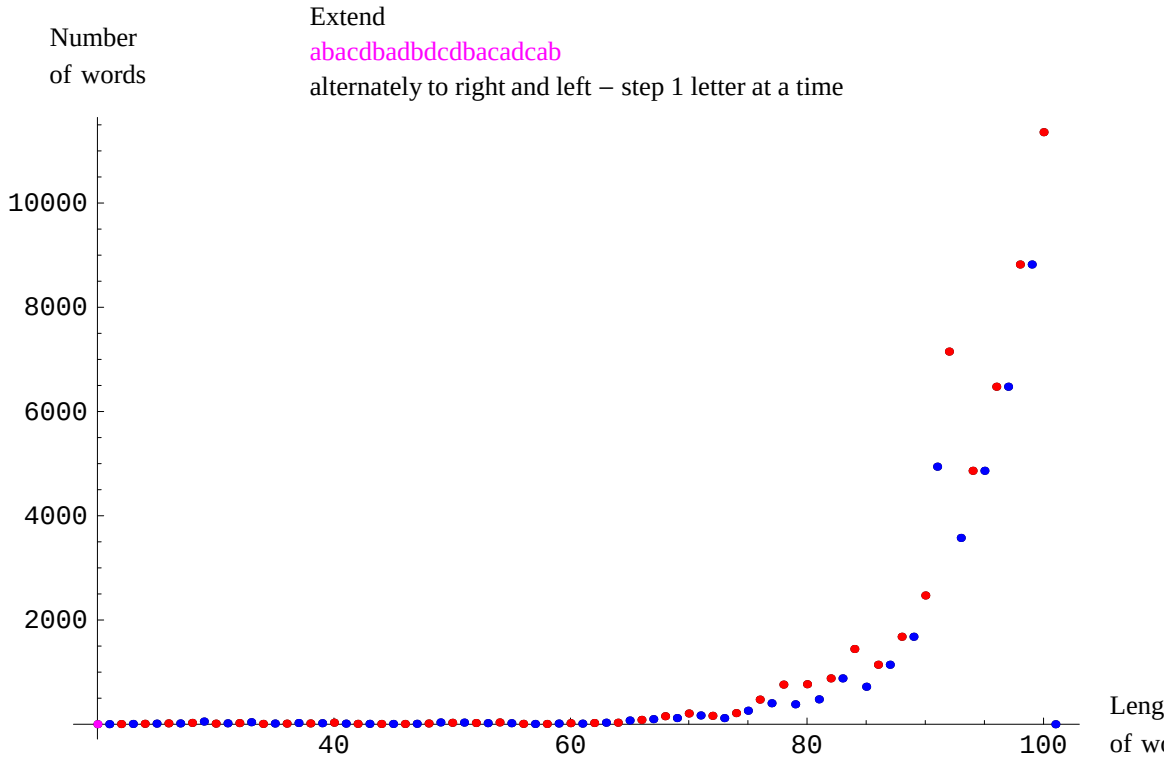
We provide a little more explanation for the case of the figure above. Here we have tried to extend the word **"abcbdbcbacbdcdacbdac"** to the right (blue) and to the left (red) with all possible letters over Σ_4. The length of the words increases only by one letter at a time (alternately to right and left). All possible words are gathered to a list and the number of words are plotted according to the length of the words. For each word there are four possibilities: either it can be continued with 3, 2, 1, or 0 letters. The case of 0 letters means the word cannot be continued (in an a–2–f way) at all. If no words can be continued, then we get an empty list and the computation ends. The resulting list consists merely of unfavourable (or bad, or forbidden) factors that start from the originally given word, and this starting word is 'in the middle' of every non–empty word in the list. Indeed, all the computations for the words in the list of **someUnfavourableFactorsOfLength20**, together with a great many other similar words, do end with an empty list (i.e. with a dead end) even though the width of the bidirectional–tree, and the computational time, can be quite huge. All this can make finding of unfavourable factors really challenging.

Some additional illustrations of this phenomenon can be found from below:


Extend
abacadbcbacbdcdacbab
alternately to right and left – step 1 letter at a time
Number
of words
120000
100000
80000
60000
40000
20000
30
40
50
60
70
80
90
Lengt
of wo


Extend
abacadbdadcacbadcbab
alternately to right and left – step 1 letter at a time
Number
of words
5000
4000
3000
2000
1000
30
40
50
60
70
80
90
Lengt
of wo





In passing we mention that some unfavourable factors can even be cut shorther from the right or left ends to (shortened) factors which, after a transitional phase, have the same final trajectory.

■ Preliminaries

In this section we present some mathematical notations and terminology. Our terminology is more or less standard in the field of combinatorics on words. Consequently, the reader might consult this section later, if need arises.

An *alphabet* Σ is a finite non−empty set of abstract symbols called *letters*. A *word* (*string*) over Σ is a finite (unless otherwise indicated) string, or sequence, of letters belonging to Σ. The set of all words [non−empty words] over Σ is denoted by Σ^* [Σ^+]. On the Σ^*, the associative binary operation of *catenation* is defined. For words u and v, it is the juxtaposition uv. The *empty word*, which is the neutral element of catenation, is denoted by λ. The algebraic structures Σ^* and Σ^+ are called, respectively, the free monoid and the free semigroup generated by Σ.

Let $w = x_1 \cdots x_m$, $x_i \in \Sigma$. The *length* of the word w, denoted by $|w|$, is the number of occurrences of letters in w, i.e., $|w| = m$. Let $\Sigma = \{ a_1, \ldots, a_n \}$. The number of occurrences of one letter $x \in \Sigma$ in w is denoted by $|w|_x$, or simply by $|w|_i$ if $x = a_i$. The notation $\psi_\Sigma(w)$ stands for the *Parikh vector* of w, i.e., $\psi_\Sigma(w) = (|w|_1, \ldots, |w|_n)$. Usually we will omit the subscript Σ and write simply ψ instead of ψ_Σ. Quite interchangeably with the Parikh vector notation also formal sums $\psi_s(w) = \sum_{x \in \{a_1, \ldots, a_n\}} k_x\, x$, with $k_x \in \mathbb{N}$, are used. For example $\psi_s(abacaba) = 4a + 2b + c$. Thus $\psi_s(w)$, with $w \in \Sigma$, is an element of the abelian free monoid $\mathbb{N}^\Sigma$ generated by Σ. We will also consider differences of Parikh vectors and differences of formal sums. Consequently, these vectors and sums are extended into elements of the abelian free group $\mathbb{Z}^\Sigma$ generated by Σ. The neutral element of $\mathbb{Z}^\Sigma$ is denoted by $\mathbf{0}$.

A word u is called a *factor* (some authors call it *subword*) of a word w, if $w = p\,u\,s$ for some words p and s. If $p = \lambda$ [$s = \lambda$], then u is called a *prefix* [a *suffix*] of w.

Let $k \geq 2$ be a given integer. A *k−repetition* [an *abelian k−repetition*] is a non−empty word of the form R^k [$P_1 \cdots P_k$, where $\psi(P_\mu) = \psi(P_\nu)$ for all $1 \leq \mu < \nu \leq k$, i.e., P_i:s are permutations of each other]. Instead of [abelian] 2− and 3−repetitions, terms [*abelian*] *squares* and [*abelian*] *cubes* are often used. A word or an ω−word (explained below) is called *k−repetition free* [*abelian k−repetition free*, or *k−free in the abelian sense*], or in short *k−free* [*a−k−free*], if it does not contain any *k*−repetition [abelian *k*−repetition] as a factor. A word sequence or a word set is *k−free* [*a−k−free*], if all words in it are *k*−free [a−*k*−free]. If, for a fixed k, it is possible to construct arbitrarily long (infinite) a−*k*−free (or other pattern−free) words over a given alphabet Σ, then we say that abelian *k*−repetitions (or those patterns) are *avoidable* over Σ.

A *morphism* h is a mapping between free monoids Σ^* and Δ^* with $h(uv) = h(u)h(v)$ for every u and v in Σ^*. Especially, $h(\lambda) = \lambda$. A morphism $h: \Sigma^* \to \Delta^*$, being compatible with the catenation of words, is uniquely defined, if the word $h(x) \in \Delta^*$ is (effectively) given for each $x \in \Sigma$. If $\Delta = \Sigma$, we call h an *endomorphism* (and usually write g instead of h). For a morphism h and a language L we define $h(L) = \{h(w) \mid w \in L\}$. A morphism h is termed *uniformly growing*, if $|h(x)| = |h(y)| \geq 2$ for every x and $y \in \Sigma$.

For a given integer $k \geq 2$, a morphism $h: \Sigma^* \to \Delta^*$ is called *k−free* [*a−k−free*], if $h(w)$ is *k*−free [a−*k*−free] for every *k*−free [a−*k*−free] word $w \in \Sigma^*$.

With regards to *L−systems* (Aristid Lindenmayer 1925-1989), we specify the following concepts. A *D0L−system* is a triple $G = (\Sigma, g, \alpha_0)$, where Σ is an alphabet, $g: \Sigma^* \to \Sigma^*$ is an endomorphism, and α_0, called the *axiom*, is a word over Σ. The (*word*) *sequence* $S(G)$ generated by G consists of the words

$$\alpha_0 = g^0(\alpha_0),\ g^1(\alpha_0),\ g^2(\alpha_0),\ g^3(\alpha_0),\ \ldots,$$

where $g^i(\alpha_0) = g(g^{i-1}(\alpha_0))$ for $i \geq 1$. The *language* of G is defined by $L(G) = \{g^i(\alpha_0) \mid i \geq 0\}$. Languages [sequences] defined by a D0L−system are referred to as *D0L−languages* [*D0L−sequences*]. D0L−systems provide a very convenient way for defining languages and infinite words. Furthermore, if g and α_0 are k−free [a−k−free], then the iteration of g will yield a k−free [a−k−free] D0L−sequence. An *HD0L−system* is a 5−tuple $G_1 = (\Sigma, \Delta, g, h, \alpha_0)$, where (Σ, g, α_0) is a D0L−system, called the underlying D0L−system of G_1, Δ is an alphabet, and $h: \Sigma^* \to \Delta^*$ is a morphism. The *HD0L−sequence* $S(G_1)$ generated by G_1 consists of the words

$$h(\alpha_0) = h(g^0(\alpha_0)),\ h(g^1(\alpha_0)),\ h(g^2(\alpha_0)),\ h(g^3(\alpha_0)),\ \ldots,$$

and the *HD0L−language* of G_1 is the set $L(G_1) = \{h(g^i(\alpha_0)) \mid i \geq 0\}$. A *DT0L−system* is a triple $G_2 = (\Sigma, H, \alpha_0)$, where H is a finite non−empty set of morphisms (called tables) and (Σ, h, α_0) is a D0L−system for every $h \in H$. The *DT0L−language* of G_2 is the set $L(G_2) = \{w \mid w = \alpha_0$ or $w = h_k \cdots h_1(\alpha_0)$, where the compositions $h_k \cdots h_1$ of morphims are constructed from $h_1, ..., h_k \in H\}$. Obviously, a DT0L−system can be regarded as a D0L−system, when H contains only one (endo)morphism. For a thorough discussion of various L−systems we refer the reader to Rozenberg and Salomaa [RS].

An *ω−word* is an infinite sequence, from left to right, of letters of an alphabet Σ. Thus an ω−word can be identified with a mapping of $\mathbb{N}_+$ into Σ. One can construct an ω−word, for example, by iterating an endomorphism $g: \Sigma^* \to \Sigma^*$ such that $\lambda \notin g(\Sigma)$ and $g(x) = xw$ for some $x \in \Sigma$, $w \in \Sigma^+$. Such a morphism g is called *prefix preserving* for the reason that $g^i(x)$ is a proper prefix of $g^{i+1}(x)$ whenever $i \geq 0$. An ω−word is obtained as the "limit" of the sequence $g^i(a)$; $i = 0, 1, 2, \ldots$.

■ Suppression of Unfavourable Factors with *Mathematica*

In this section we give examples of the developed *Mathematica* program. The correctness of the code and the related packages has been tested but exhaustive computer runs are still likely to take a long time in the future. The full code and further versions of it can be downloaded from [K6], or from [K7], the latter of which is a general link page for the topic. In addition, the code (in a partly preliminary form) is attached at the end of this notebook, and the reader is invited to make experiments with it. For this purpose, one may copy and edit the Input Cells of the examples presented below. The code consists of Initialization Cells and will be activated automatically. Nevertheless, it should be noted that, intially, only the 4 letter case can be tried out. The 3 letter case requires the user to activate the very last Input Cell of this notebook which lies inside the last section.

In the following examples, many of the function definitions are omitted (they can be found at the end). It should be noted that all the structures, variables and constants starting with **€** are global. For example, the global variable **€state** represents the state of the construction. Its values are strings, including, as an example, the following ones: **"extendOrChangeRight"**, **"extendOrChangeLeft"**, **"testRight"**, **"testLeft"**, **"failBoth"**, and **"succeeded"**.

As mentioned in the Introduction, we firstly fix the alphabet Σ and consider words over it. We take a word (in the final investigation, we will actually take all the a−2−free words of a given length) and try to extend it in a−2−free fashion to the right and to the left with all possible ways up to a given upper bound for the total length. At a time, the length of the word increases only by a given fixed length. We extend alternately to right and left, and

backtrack when necessary. If the upper bounds are reached then the original word is a *so−far−favourable* one. If there is no way to reach the upper bounds, then the original word is classified, without any doupt, to be *unfavourable*. At present, the *favourable* words (4 letter case) consist of only those occurring as factors in a−2−free words obtained by using the endomorphims g_{85}, g_{98}, and Carpi's modification of g_{85}.

In the final program we use integer coding for letters of the alphabet Σ and cumulative integer lists for words. This makes the detection of abelian squares fast. We use two different cumulative integer lists, **€cumulIntListRight** for the right hand, and **€cumulIntListLeft** for the left hand extensions. This makes addition of integers (addition of cumulative integer lists, in fact) fast but forces us to write the testing function **testA2RLpairCumulIntList** in a more complicated fashion (nevertheless, still maintaining the necessary high speed). In building all the structures, quite extensive precomputations are needed.

Fistly, let us define the alphabets by using letters (strings of length one) and integers:

```
€alphTwoLet = {"a", "b"};
€alphTwoInt = {0, 1};
€alphThreeLet = {"a", "b", "c"};
€alphThreeInt = {0, 1, 2^16};
(* Words of length ≤ (2^16 -1)*4/3 = 87380
    are safe to use - provided that they do not
    contain xxxx for a letter x in €alphThreeLet *)
€alphFourLet = {"a", "b", "c", "d"};
€alphFourInt = {0, 1, 2^10, 2^20};
(* Words of length ≤ (2^10 -1)*2 = 2046
    are safe to use - provided that they do
    not contain xx for a letter x in €alphFourLet *)
€alphThreeLetInt = {€alphTwoLet, €alphTwoInt};
€alphThreeLetInt = {€alphThreeLet, €alphThreeInt};
€alphFourLetInt = {€alphFourLet, €alphFourInt};
```

The first example of the use of cumulative Parikh vectors is a symbolic one (Parikh vector is explained in the Preliminaries section):

```
convertStrToCumulIntList["bacabcacab",
 {{"a", "b", "c"}, {"a", "b", "c"}}]
```

```
{b, a + b, a + b + c, 2 a + b + c, 2 a + 2 b + c, 2 a + 2 b + 2 c,
 3 a + 2 b + 2 c, 3 a + 2 b + 3 c, 4 a + 2 b + 3 c, 4 a + 3 b + 3 c}
```

In actual computations, we use integer representation for these cumulative Parikh vectors:

```
convertStrToCumulIntList["bacabcacab", €alphThreeLetInt]
```

```
{1, 1, 65537, 65537, 65538, 131074, 131074, 196610, 196610, 196611}
```

Transforming back to the string representation over three letters:

```
convertCumulIntListToStr[%, €alphThreeLetInt]
```

```
bacabcacab
```

The case of four letters can be handled in a similar way:

```
convertStrToCumulIntList["bacabdacad", €alphFourLetInt]
```

```
{1, 1, 1025, 1025, 1026, 1049602,
 1049602, 1050626, 1050626, 2099202}
```

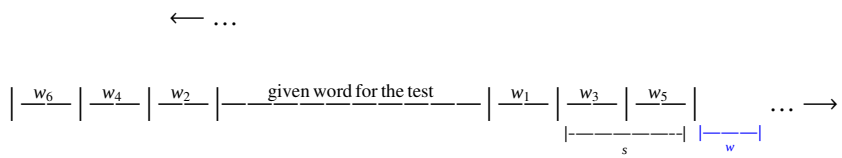
```
convertCumulIntListToStr[%, €alphFourLetInt]
```

```
bacabdacad
```

When extending the words (to right or left) the program looks at the suffix s (of fixed length) of the constructed word and then catenates a new word w of a fixed length to it.

This new word w is selected, in an orderly fashion, from a precomputed list for s and it is always good, firstly, in the sense that no abelian squares are produced in the new suffix sw, and, secondly, in the sense that sw itself is never an unfavourable (i.e. bad) factor. However, new longer abelian squares can appear in the whole structure in which case the catenation is not successful and the next candidate w, if it exists, will be tried out. The process is depicted in Figure 1. We have also taken care that the selection of the next proper candidate is done in an efficient way. If all the possible candidates for w have already been tried out, then the program backtracks to change the other end's suffix. Indeed, the process is the same for for the right and left hand extensions (both of their cumulative list representations grow from left to right), so we can really speak of suffixes only. The lengths for the suffix s and the extension w need to be fixed at the beginning of the computation, and changing (re−fixing) the lengths usually requires new quite extensive precomputations. In the code at the end of this notebook, the lengths have been selected to be $|s| = 4$ and $|w| = 4$, but they can be, and usually are, selected differentely as well. Up to now, we have been using e.g. the lengths $|s| = 8, |w| = 4$, and $|s| = 12, |w| = 4$. Especially, longer suffixes s would increase the computaional time efficiency considerably but, on the other hand, the structures might need too much memory in our present environment for distributed computing. Moreover, selecting $|w| = 1$ would allow the maximal avoidance of unfavourable factors in the extensions. However, the setting $|w| > 1$ probably allows to detect the abelian squares more quickly, albeit the final decision of this still requires quite extensive experiments.

Figure 1:



In the example below, we add all possible words w of length 4, represented by lists that follow all possible *reduced* suffixes s of length 4. Here the term *reduced* means that we pay attention only to the structure of the word - the other possibilites can straightforwardly be obtained by permutations, i.e., by renaming letters. The example originates from the 3 letter case in which we allow short abelian repetitions of the form xx or xxx for a letter $x \in \Sigma_3 = \{a, b, c\}$. For simplicity, we show the words only in the string form:

```
€reducedWordListSuff4Add4 =
 selectWordsWithProperPrefixes[#, addWordList8StartWitha] & /@
  reducedWordList4
```

```
{{aaab, {aaac, aaca, aacb, aacc, acaa, acab, acba, acbb, acbc,
   accb, accc, bbaa, bbac, bbca, bbcb, bbcc, bcaa, bcab,
   bcac, bcba, bcbb, bcca, bccc, caaa, caab, caba, cabb,
   cacc, cbaa, cbab, cbba, cbbb, ccaa, ccac, ccca, cccb}},
 {aaba, {aaca, aacb, aacc, acaa, acab, acba, acbb,
   acbc, accb, accc, caaa, caab, caba, cabb, cbaa,
   cbab, cbba, cbbb, cbcc, ccbb, ccbc, ccca, cccb}},
 {aabb, {baaa, baac, baca, bacb, bacc, bcaa, bcab, bcac, bcba,
   bcbb, bcca, bccc, caaa, caab, caba, cabb, cabc, cacb, cacc,
   cbaa, cbab, cbac, cbba, cbbb, ccaa, ccab, ccac, ccca, cccb}},
 {aabc, {aaab, aaac, aaba, aabb, abaa, abbb, abbc, accb,
   accc, baaa, babb, babc, bbaa, bbab, bbac, bbba,
   caaa, cacb, cacc, ccaa, ccab, ccac, ccba, ccbb}},
 {abaa, {acaa, acab, acba, acbb, acbc, accb, accc, caaa,
   caab, caba, cabb, cabc, cbaa, cbab, cbac, cbba,
   cbbb, cbca, cbcc, ccba, ccbb, ccbc, ccca, cccb}},
 {abac, {aaab, aaba, aabb, abaa, abbb, abbc, baaa,
   babb, bbaa, bbab, bbba, bbbc, bcca, bccc, cbba,
   cbbb, cbca, cbcc, ccaa, ccab, ccba, ccbb, ccbc}},
 {abbb, {aaab, aaac, aaca, aacb, aacc, acaa, acab, acba, acbb,
   acbc, accb, accc, caaa, caab, caba, cabb, cabc, cacb, cacc,
   cbaa, cbab, cbac, cbba, cbbb, ccaa, ccab, ccac, ccca, cccb}},
 {abbc, {aaab, aaac, aaba, aabb, abaa, abbb, abcc, acbc, accb,
   accc, baaa, baac, babb, baca, bacc, bbaa, bbab, bbba, caaa,
   caab, caba, cabc, cacb, cacc, ccaa, ccab, ccac, ccba, ccbb}},
 {abca, {aaba, aabb, aacc, abaa, abbb, abbc, baaa,
   bbba, bbbc, bbcb, bbcc, ccba, ccbb, ccbc, cccb}},
 {abcb, {aaab, aaac, aaca, aacc, abbb, abcc, baaa,
   baac, babb, baca, bacc, bbaa, bbab}},
 {abcc, {aaab, aaac, aaba, aabb, aabc, acba, acbb, accc,
   caaa, caab, caba, cabb, cacc, cbaa, cbab, cbba, cbbb}}}
```

To save memory and to make the computations as fast as possible, the final calculations in fact use symbols instead of strings. The cumulative integer lists are associated to symbols by using rules. Moreover, to save memory, the permutations are added only when needed. In the following example we show a rule from symbols to cumulative integer lists (in an abbreviated form):

```
€ruleSymbolsToCumulIntegerLists =
  {aaab → {0, 0, 0, 1}, aaba → {0, 0, 1, 1}, aabb → {0, 0, 1, 2},
   aabc → {0, 0, 1, 65537}, abaa → {0, 1, 1, 1},
   abac → {0, 1, 1, 65537}, abbb → {0, 1, 2, 3},
   abbc → {0, 1, 2, 65538}, abca → {0, 1, 65537, 65537},
   abcb → {0, 1, 65537, 65538}, abcc → {0, 1, 65537, 131073}, ...,
   acbb → {0, 65536, 65537, 65538},
   bbba → {1, 2, 3, 3}, ..., bcaa → {1, 65537, 65537, 65537},
   ccca → {65536, 131072, 196608, 196608}, ...,
   cbaa → {65536, 65537, 65537, 65537}};
```

In our present setting, we cut the suffix s from the cumulative integer list(s) for the so–far constructed word and then select the new extension w from the precomputed symbolic list for s. After this, the corresponding cumulative integer list for w will be added to (the cumulative integer list of) the previously constructed word. This is done in order to detect the possible new abelian squares in a quick manner. One function we use for selecting the new extensions is **tryProperExtensionVisList**. Below we present, as an example, one rule (of a great many) connected to it. This time the example is from the 4 letter, and $|s| = 12$, $|w| = 4$, case:

```
tryProperExtensionVisList[{0, 1, 1, 1025, 1025, 1026,
   1026, 1049602, 1049603, 1050627, 1050627, 1051651}] =
 {bcda, bcdb, bcdc, daca, dacb, dadb, dcab, dcba, dcbc, dcbd}
```

These kinds of precomputed rules are quite fast to use, but, of course, a considerable amount of memory is needed to store all the structures.

The following function **generateRLthree** (or, equivalently, **generateRL**) constructs the extensions for a given word. Its arguments are states (strings that are also values of the global variable **€state**). As explained also earlier, some examples of these states include **"extendOrChangeRight"**, **"extendOrChangeLeft"**, **"testRight"**, **"testLeft"**, **"failBoth"**, and **"succeeded"**. By opening the cell brackets, the code for **generateRLthree** can be viewed from below. Note that even the code was originally developed for the 3 letter case (allowing short abelian repetitions of the form *xx* or *xxx* for a letter $x \in \Sigma_3 = \{a, b, c\}$), it works perfectly for the 4 letter case as well in all our settings. Therefore, the more generic name **generateRL** can be, and will be, also used for it.

```
? generateRLthree
```

Before starting the construction, we need to intialise the global (at this stage) variables and download the proper package (**alphThreeLetIntCase.m** (18 KB), or **alphFourLetIntCase.m** (35 KB)) depending on whether we use 3 or 4 letters. This is accomplished by using the **intialise** function that has the following main structure:

```
initialise[howManyStepsLeftAndRight_,
 givenWordForTest_, alphLetInt_:€alphThreeLetInt]
```

The variables **€extensionBoundaryLengthRight** = **howManyStepsLeftAndRight** * **€addWordLength** and **€extensionBoundaryLengthLeft** = **howManyStepsLeftAndRight** * **€addWordLength** stand for extension lengths. Both of these lengths should be equal and of the form *k* * **€addWordLength** for a positive integer *k*. In our examples, and in our present code, the value for **€addWordLength** (length of *w* above) is equal to 4. The full definition of the **intialise** function is as follows:

```
In[10]:= Clear[initialise];
initialise[howManyStepsLeftAndRight_,
   givenWordForTest_, alphLetInt_:€alphThreeLetInt] :=
  (€alphLetInt = alphLetInt;  (* €alphLetInt needs to be
     set to €alphThreeLetInt or to €alphFourLetInt *)
   If[alphLetInt == €alphThreeLetInt, << "alphThreeLetIntCase.m",
    << "alphFourLetIntCase.m"];
   €addWordLength = 4;
   €preCheckedSuffLength = 8;
   €suffLengthForWhichTryProperExtension =
    €preCheckedSuffLength - €addWordLength;
   €reducedWordListSuffNAddNToExpressions =
    €reducedWordListSuff4Add4ToExpressions;
   ordinalIndexForCumulIntegerListN =
    ordinalIndexForCumulIntegerList4; (* function names *)
   €pointerExtendRight = 0;
€pointerExtendLeft = 0;
€extensionBoundaryLengthRight =
    howManyStepsLeftAndRight * €addWordLength;
   €extensionBoundaryLengthLeft =
    howManyStepsLeftAndRight * €addWordLength;
   (* Indeed, both of these should be equal and of the
     form  k * €addWordLength for a positive integer  k *)
   €howFarExtendedRight = 0;
   €howFarExtendedLeft = 0;
   €indexListRight = {0};
   €indexListLeft = {};
   €givenWordForTest = givenWordForTest;
   €cumulIntListOfGivenWordForTest =
    convertStrToCumulIntList[€givenWordForTest, €alphLetInt];
   €cumulIntListReverseOfGivenWordForTest =
    convertStrToCumulIntList[
     StringReverse[€givenWordForTest], €alphLetInt]);
```

For the purpose of this notebook, however, the user does not need to download any packages. The code will be activated automatically for the 4 letter case (the 3 letter case requires the evaluation of the very last Input Cell of this notebook).

An example of the three letter case follows (note that this Input Cell is not activated):

```
initialise[3, "aaabbbaaacccaaabbb"];

Print["€givenWordForTest = ", €givenWordForTest];
generateRLthree["startConstruction"];
While[ €state =!= "failBoth" ∧ €state =!= "succeeded",
  generateRLthree[€state] ] // Timing
generateRLthree[€state]
```

```
€givenWordForTest = aaabbbaaacccaaabbb
```

```
{0.01 Second, Null}
```

```
For  aaabbbaaacccaaabbb  the extension failed!
```

Thus, the given word is an *unfavourable* one.

The examples below deal with the four letter case. Fistly, we try to extend a word of length 20:

```
"abacabadbacbcdbacdbd" // StringLength
```

```
20
```

The extra condition in the **While** loop guarantees that computation will not last too long:

```
initialise[8, "abacabadbacbcdbacdbd", €alphFourLetInt];

generateRL["startConstruction"]; iii = 1;
Print["€givenWordForTest = ", €givenWordForTest];
While[ iii ≤ 50000 ∧ €state =!= "failBoth" ∧ €state =!= "succeeded",
  generateRL[€state]; iii++ ] // Timing
generateRL[€state]
```

```
€givenWordForTest = abacabadbacbcdbacdbd
```

```
{7.501 Second, Null}
```

```
For  abacabadbacbcdbacdbd  the extension was successful!
```

Well, after all, the computation was not too long. At this point, we know only that the given word is a *so−far−favourable* one.

We may look at some of the inner values and structures produced by the previous computation:

```
myPrint["4 letters"]
```

```
€cumulIntListLeft as reverse
   string | €cumulIntListRight as string =
 cbcacbcdcabdbabcdabcacdadbdadcadabacabadba|
 cbcdbacdbdadbcacdacabdabcbdadcdbcbadbcbdbc
```

```
€howFarExtendedLeft = 0
```

```
€indexListLeft = {23, 1, 2, 12, 4, 24, 12, 25}
```

```
€howFarExtendedRight = 0
```

```
€indexListRight = {24, 11, 3, 21, 20, 12, 14, 34, 0}
```

```
€state = succeeded
```

Above the lists **€indexListLeft** and **€indexListRight** contain the pointer values that indicate which words *w* from the precomputed lists should be next catenated for testing to corresponding suffixes *s*. Variables **€howFarExtendedLeft** and **€howFarExtendedRight** are used for efficient a−2−freeness testing and for finding the next proper word *w* for catenation as efficiently as possible.

Below we test that the reached extension is indeed an abelian square−free word:

```
testA2[
 "cbcacbcdcabdbabcdabcacdadbdadcadabacabadbacbcdbacdbdadbcacda
    cabdabcbdadcdbcbadbcbdbc"]
```

```
True
```

We will extend the same word **"abacabadbacbcdbacdbd"** of length 20 one more step (of length 4) to the right and to the left:

```
initialise[9, "abacabadbacbcdbacdbd", €alphFourLetInt];

generateRL["startConstruction"]; iii = 1;
Print["€givenWordForTest = ", €givenWordForTest];
While[ iii ≤ 100000 ∧ €state =!= "failBoth" ∧ €state =!= "succeeded",
  generateRL[€state]; iii++ ] // Timing
generateRL[€state]
```

```
€givenWordForTest = abacabadbacbcdbacdbd
```

```
{37.895 Second, Null}
```

```
testRight
```

Once again, we know only that the given word is a *so−far−favourable* one.

Let us consider the following word of length 84:

```
"cbcacbcdcabdbabcdabcacdadbdadcadabacabadbacbcdbacdbdadbcacdac
    abdabcbdadcdbcbadbcbdbc" // StringLength
```

```
84
```

```
initialise[3,
  "cbcacbcdcabdbabcdabcacdadbdadcadabacabadbacbcdbacdbdadbcacd
     acabdabcbdadcdbcbadbcbdbc", €alphFourLetInt];

generateRL["startConstruction"]; iii = 1;
Print["€givenWordForTest = ", €givenWordForTest];
While[ iii ≤ 100000 ∧ €state =!= "failBoth" ∧ €state =!= "succeeded",
  generateRL[€state]; iii++ ] // Timing
generateRL[€state]
```

```
€givenWordForTest =
 cbcacbcdcabdbabcdabcacdadbdadcadabacabadbacbcdbacdbdadbcacda‹
    cabdabcbdadcdbcbadbcbdbc
```

```
{0.03 Second, Null}
```

```
For
 cbcacbcdcabdbabcdabcacdadbdadcadabacabadbacbcdbacdbdadbcacda‹
    cabdabcbdadcdbcbadbcbdbc  the extension failed!
```

In this case we know definitely that the given word of length 84 is an *unfavourable* one. Actually, it turns out that the given word is already the longest possible extension of w = "**abacabadbacbcdbacdbd**" ! Note that above the **€givenWordForTest** has the form of **"cbcacbcdcabdbabcdabcacdadbdadcad"** <> w <> **"adbcacdacabdabcbdadcdbcbadbcbdbc"**. However, we will not elaborate on this here.

Let us consider another given word:

```
initialise[8, "abcbdbcbacbdcdacbdac", €alphFourLetInt];

generateRL["startConstruction"];
Print["€givenWordForTest = ", €givenWordForTest];
While[€state =!= "failBoth" ∧ €state =!= "succeeded",
  generateRL[€state]; ] // Timing
generateRL[€state]
```

```
€givenWordForTest = abcbdbcbacbdcdacbdac
```

```
{10.953 Second, Null}
```

```
For  abcbdbcbacbdcdacbdac  the extension was successful!
```

Thus, for the time being, our case is a *so−far−favourable* one. The computation did not take a long time even the extension was quite long (8·4 = 32 to both sides).

We may view some of the inner values and structures produced by the previous computation:

```
myPrint["4 letters"]
```

```
€cumulIntListLeft as reverse
   string | €cumulIntListRight as string =
 abdabacbdbadbdcdadbabcacdbcabadbabcbdbcbac|
 bdcdacbdacabacdbdabdbcabcdcabadacdcbacdcac
```

```
€howFarExtendedLeft = 0
```

```
€indexListLeft = {10, 12, 14, 18, 19, 10, 16, 13}
```

```
€howFarExtendedRight = 0
```

```
€indexListRight = {28, 40, 13, 32, 25, 23, 16, 23, 0}
```

```
€state = succeeded
```

Let us test that the reached extension really is an abelian square−free word:

```
testA2[
 "abdabacbdbadbdcdadbabcacdbcabadbabcbdbcbacbdcdacbdacabacdbda
    bdbcabcdcabadacdcbacdcac"]
```

```
True
```

This is correct!

The final example shows that our word **"abcbdbcbacbdcdacbdac"** above turns out to be an *unfavourable* one. However, this time the computation takes nearly two hours (in a 1.6 GHz machine) instead of 11 seconds above:

```
initialise[9, "abcbdbcbacbdcdacbdac", €alphFourLetInt];

generateRL["startConstruction"];
Print["€givenWordForTest = ", €givenWordForTest];
While[€state =!= "failBoth" ∧ €state =!= "succeeded",
  generateRL[€state]; ] // Timing
generateRL[€state]
```

```
€givenWordForTest = abcbdbcbacbdcdacbdac
```

```
{6659.5 Second, Null}
```

```
For  abcbdbcbacbdcdacbdac  the extension failed!
```

One might have expected that the long buffers (found before this final trial) of length 8·4 = 32 to the both directions of **"abcbdbcbacbdcdacbdac"** would already quarantee the given word to be *favourable*. Surprisingly enough, this turns out not to be the case.

■ Conclusions

Mathematica has versatile structures that firmly support the development of concepts. This has been extremely useful for constructing prototypes and full stand−alone code. The use of *Mathematica* has enabled us to discover phenomena which previously were either unbelievable or really hard to experiment on. Even so, it would be useful, if in the future, in *Mathematica*, one could also use restricted pattern matching in a fast way that was explained in the Introduction section. Lastly, we expect that the presented code, and its future updates, will be used for a quite long time in our research.

■ Acknowledgements

We gratefully acknowledge the participation of the following individuals most of who have been students at the Rovaniemi Polytechnic / University of Applied Sciences over the course of the research from 1990 to 2006. These people have made a number of computer programs for searching strings with desirable properties, helped in a crucial way to set up the computing environments, or made interactive graphical and musical representations of the structures. Starting year is given in parenthesis: Kari Tuovinen (1990); Minna Iivonen, Anja Keskinarkaus, Marko Manninen (1993); Abdeljalil Chabani, Tomi Laakso (1994); Mika Moilanen, Juha Särestöniemi (1996); Juho Alfthan (1999); Olli−Pentti Saira (2000); Marja Kenttä, Ville Mattila (2001); Lauri Autio, and Marianna Mölläri (2002); Antti Eskola (2003); Antti Karhu, Veli−Matti Lahtela, Olli−Pekka Siivola (2004); Esa Nyrhinen, Sami Vuolli (2005); Esa Taskila, and Mikhail Kalkov (2006).

■ References

[AS1] J.–P. Allouche and J. Shallit. The ubiquitous Prouhet–Thue–Morse sequenece. In C. Ding, T. Helleseth, and H. Niederreiter, editors, Sequences and Their Applications, Proc. SETA '98, 1–16. Springer–Verlag, 1999.
[AS2] J.–P. Allouche and J. Shallit. Automatic Sequences – Theory, Applications Generalizations. Cambridge University Press, 2003.
[B1] J. Berstel. Some recent results on square–free words. In M. Fontet and K. Melhorn, editors, Proc. STACS '84, Lecture Notes in Comp. Sci., 166:14–25. Springer–Verlag, Berlin, 1984.
[B2] J. Berstel. Axel Thue's work on repetitions in words: a translation, Publications du LCIM 20, 16 pages. Université du Quebec à Montréal, 1994.
[C1] A. Carpi. On abelian power–free morphisms. Int. J. Algebra Comput., 3:151-167. World Sci. Publ. Company, 1993.
[C2] A. Carpi. On the number of abelian square–free words on four letters. Discrete Appl. Math., 81:155-167. Elsevier, 1998.
[C3] A. Carpi. On abelian squares and substitutions. Theor. Comp. Sci., 218:61-81. Elsevier, 1999.
[CC] J. Cassaigne and J.D. Currie. Words strongly avoiding fractional powers. Europ. J. Combinatorics, 20:725-737. Academic Press, 1999.
[E] P. Erdös. Some unsolved problems. Magyar Tud. Akad. Mat. Kutató Int. Közl., 6:221-254, 1961.
[K1] V. Keränen. Abelian squares are avoidable on 4 letters. In W. Kuich, editor, Proc. ICALP '92, Lecture Notes in Comp. Sci., 623:41-52. Springer–Verlag, Berlin, 1992.
[K2] V. Keränen. Mathematica in research of avoidable patterns in strings. In V. Keränen and P. Mitic, editors, Mathematics with Vision, Proc. First International Mathematica Symposium (IMS '95, Southampton, England), 259-266. Computational Mechanics Publications, 1995.
[K3] V. Keränen, New abelian square–free DT0L–languages over 4 letters. Proc. Fifth International Arctic Seminar (*IAS* 2002, May 15 – 17, 2002, Murmansk, Russia), Murmansk State Pedagogical Institute, 2002, 14 pages. Available online at http://south.rotol.ramk.fi/keranen/ias2002/ias2002papers.html.
[K4] V. Keränen. On abelian square–free DT0L-languages over 4 letters. In T. Harju and J. Karhumäki, editors, Proc. 4th International Conference on Combinatorics on Words 27:95-109. Turku Centre for Computer Science, Turku, 2003.
[K5] V. Keränen. Forbidden abelian square–free factors over 4 Letters, 2004. Available online at
http://south.rotol.ramk.fi/keranen/research/ForbiddenFactors.html.
[K6] V. Keränen. Programs & notebooks & packages. *Mathematica* code for suppression of unfavourable factors in pattern avoidance, 2006. Available online at
http://south.rotol.ramk.fi/keranen/research/UnfavourableFactorsInPatternAvoidance/Programs&Notebooks&Packages.html.
[K7] V. Keränen. 2006. Avoidable regularities in strings. A collection of links to web pages on structures, graphics, and music of abelian square–free strings, 1996-2006. Available online at http://south.rotol.ramk.fi/keranen/StructuresGraphicsMusic.html.
[Ma1] V. Mattila. Creating permutation repetition–free words and testing permutation repetition–freeness of morphisms (in Finnish). B.Sc. Thesis. Rovaniemi Polytechnic, 2002.
[Ma2] V. Mattila. Constructing abelian square–free words and testing morphisms with *Mathematica.* In V. Demidov and V. Keränen, editors, Proc. IAS 2002. Murmansk State Pedagogical Institute and Rovaniemi Polytechnic, Murmansk 2002. Available online at http://south.rotol.ramk.fi/keranen/ias2002/ias2002papers.html.

[Mä] S. Mäkelä. Patterns in Words (in Finnish). M.Sc. Thesis. Univ. Turku, 2002.
[Pl] P.A.B. Pleasants. Non−repetitive sequences, Proc. Cambridge Phil. Soc., 68:267-274, 1970.
[Pr] M. E. Prouhet. Mémoire sur quelques relations entre les puissances des nombres. C. R. Acad. Sci. Paris, 33:225, 1851.
[R] R.L. Rivest. Abelian square−free dithering for iterated hash functions. MIT. Draft; to appear, 2005. Available online at
http://theory.lcs.mit.edu/~rivest/publications.html.
[RS] G. Rozenberg and A. Salomaa. The Mathematical Theory of L−systems. Academic Press, New York, London, Toronto, Sydney, San Fransisco, 1980.
[S] A. Salomaa. Jewels of Formal Language Theory. Computer Science Press, Rockville, Maryland, 1981.
[T1] A. Thue. Über unendliche Zeichenreihe. Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiania, 7:1-22, 1906.
[T2] A. Thue. Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiania, 7:1-67, 1912.
[W] S. Wolfram. A New Kind of Science. Wolfram Media, 2002.

■ Automatically Activated Code for the Case of 4 Letters