On Computability 1A

Darius Lorek

February 26, 2023

Abstract

We continue to show how constraint satisfaction can help solve various configuration and pattern matching problems. Here, we focus on the topic of decoding. Decoding a substitution cipher is a complex task, particularly when there is a lack of information about the plaintext language or the type of encryption key used. For older ciphers, mathematical methods can be ruled out, such as the RSA algorithm based on prime number encryption, which was introduced in the 1970s. Therefore, for historical ciphers, we can assume that some kind of substitution was involved.
We define a substitution cipher (not in a mathematical sense) as a code that uses glyphs, signs, or letters to substitute for letters or phonemes of the plaintext language during the encryption process. This replacement can be pre- or post-processed. As pre-processing, we understand the obscuration of the plaintext before substitution, such as the usage of unusual grammars, omitting words containing specific letters, interspersing abbreviations into the text, or similar techniques. After the substitution, as post-processing, the resulting code can be further obscured by replacing code letters at special positions with additional letters or changing the order of the code letters.
For the following example, we investigate the Voynich Manuscript (VMS). We will not provide an introduction to the history or an account of the decipherment attempts of this 15th-century encoded script and will refer to numerous books on these topics. Instead, we will jump directly into the decipherment process.

6 Pattern Matching (preliminary considerations)

We still use bash scripts for the implementation of the pattern matching routines. Bash scripts can be run on Unix/Linux, on Windows e. g. Cygwin (a collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows) can be installed[1] to run bash scripts.

We use the term “vord” for a word in Voynich language. We don’t know what exactly a vord is, but assume that it’s one or many words of a plaintext. The term “word” we use only in context of plaintext languages. As first step we examine the lexica of candidate plaintext languages and scan them for patterns, which occur in the VMS. In the VMS we can find odd vord patterns, among them patterns containing consecutive repetitions of the same glyph like in ABBBBA or generally patterns of the type …BBB… It seems obvious, that such patterns must be investigated with preference, as they make the difference to the word patterns we can normally find in a vast majority of natural languages. As working hypothesis, we assume that a glyph or letter in Voynichese stands mostly for the same plaintext letter.

[1] we do not guarantee and do not assume any liability, that the script is runnable on your operating system, we could run it on Unix and port it successfully to Cygwin on Windows

At first, we will search for such patterns in words or word combinations of plaintext languages and extract them into output files. The idea behind this approach is, that among all these extractions must be semantically correct word combinations, preconditioned that we come across the correct plaintext language. In this first decoding step, we however won't examine semantics and deal almost exclusive with the syntax. The pattern matching routines give us a database for further analysis of the potential plaintext.

The investigation shows quickly that as for the most of the languages there aren't reasonable pattern matches for the "abnormal" patterns like those mentioned above. The pattern matching for long consecutive chains of the same letter generates no results at all or implausible results, such as repetitions of the same, single, very short words like …ooo… -> or, or, or in Spanish or …aaa…, …iii… -> and, and, and in Czech resp. Polish. The general problem here is a lack of sufficient many short words and therefore a lack of short written expressions.

Hence, we are directed to languages with short words and writing systems, which makes the written expressions even more compact, often at the expense of unambiguousness. One of these writing systems is "Ketiv" or "Ketib" (which means *written*), a system used for Hebrew and Aramaic, in which vowels are written only exceptionally at the beginning or the end of the words (vocal reduction). Later, with the help of the Masoretes (medieval Jewish scribe-scholars) these old, mostly biblical Hebrew and Aramaic texts received a vocalisation punctuation, which together with marginal notes (Qere, which means *read*), allowed a standardisation for the vocalisation and pronunciation of the Ketiv-texts. We begin our investigation with these languages and try to find patterns which correspond with patterns existent in the VMS.

In the following chapters we describe the used routines and present the statistics for the generated pattern matching results.

7 Pattern Matching (full evaluations)

The data basis for the pattern matching is a lexicon with 8675 Hebrew and Aramaic words[2], which on its part is a collection of different lexicons and Bible dictionaries. As for this analysis we deviate from the conventional transliteration and base upon the abjad derived from the standard transliteration with an additional replacement of all digrams by a single letter or sign (ph → f, ch → x, sh → š, ts → ß and th → & respectively). The last replacement (see table below) makes it more convenient to formulate the constraints and to analyse the matches because now related characters in different words are at the same position inside these words. In addition, the examination of billions word combinations is a time-consuming task and every representation simplification reduces significantly the runtimes.

[2] Hebrew Dictionary of the Old Testament Online Bible with Strong's Exhaustive Concordance, Brown Driver Briggs Lexicon, Etymology, Translations Definitions Meanings & Key Word Studies - Lexiconcordance.com

1	Hebrew/Aramaic	Transliteration	Pronunciation	Generate	Abjad	Abjad
8633	תֹּקֶף	*t@qoph*	{tek-ofe'}		*tqph*	*tqf*
8634	תֹּקֶף	*toqeph*	{to'-kef}		*tqph*	*tqf*
8635	תַּרְאֲלָה	*Tar'alah*	{tar-al-aw'}		*Trlh*	*Trlh*
8636	תַּרְבוּת	*tarbuwth*	{tar-booth'}		*trbwth*	*trbw&*
8637	תַּרְבִּית	*tarbiyth*	{tar-beeth'}		*trbyth*	*trby&*
8638	תִּרְגַּל	*tirgal*	{teer-gal'}		*trgl*	*trgl*
8639	תִּרְגַּם	*tirgam*	{teer-gam'}		*trgm*	*trgm*
8640	תַּרְדֵּמָה	*tardemah*	{tar-day-maw'}		*trdmh*	*trdmh*
8641	תִּרְהָקָה	*Tirhaqah*	{teer-haw'-kaw}		*Trhqh*	*Trhqh*
8642	תְּרוּמָה	*t@ruwmah*	{ter-oo-maw'}		*trwmh*	*trwmh*
8643	תְּרוּמִיָּה	*t@ruwmiyah*	{ter-oo-mee-yaw'}		*trwmyh*	*trwmyh*
8644	תְּרוּעָה	*t@ruw'ah*	{ter-oo-aw'}		*trwh*	*trwh*
8645	תְּרוּפָה	*t@ruwphah*	{ter-oo-faw'}		*trwphh*	*trwfh*

For performance reasons we split the Aramaic abjad database into 10 different files (aramaic<n>.txt), each containing words of length n.

```
$ wc -l aramaic?.txt

    1 aramaic0.txt
  108 aramaic1.txt
 1341 aramaic2.txt
 3293 aramaic3.txt
 2115 aramaic4.txt
 1145 aramaic5.txt
  369 aramaic6.txt
   70 aramaic7.txt
   12 aramaic8.txt
    3 aramaic9.txt
 8457 total
```

We deal thus with 108 words of length 1, 1341 words of length 2, 3293 words of length 3, etc. We ignore words longer then 9 (mostly composite proper names). This eventually reduces the total count of examined words to 8457. See below the bash script extracting the pattern ABBBB out of all possible word combinations in biblical Aramaic. The array *arr* contains all word length combinations within a vord of length 5. Such a vord can be a word of length 5 or a combination of two or more words of length < 5. We limit here the max. count of words within a vord to 4, otherwise the words of length 1 would be highly overrepresented. Furthermore, we take consecutive word repetitions of the same one-letter words out of consideration. We regard such combinations as very unlikely. For the above pattern the valid string length combinations are: 5 41 32 23 14 311 221 131 212 122 113 2111 1211 1121 1112. The string length combinations 311, 2111, 1211 and 1112 would violate the second rule, so finally we examine here the combinations: 5 41 32 23 14 221 131 212 122 113 1121 (we right-pad the items of *arr* with 0's to simplify the nesting of the while loops). The simple set of constraints Ψ for ABBBB is then defined as Ψ = (0#1 1=2 2=3 3=4 $) specifying the glyph positions within a vord and '#' as a diff-constraint, '=' as an equal-constraint and with $ as a terminal sign.

```
#!/bin/bash
#matching for pattern ABBBB (4 words max)
typeset -a map=(0#1 1=2 2=3 3=4 $)
typeset -a arr=(5000 4100 3200 2300 1400 2210 1310 2120 1220 1130 1121 0000)
typeset -i m=0
prefix=aramaic
pattern="ABBBB-"
output=matchesABBBB.txt

while (( arr[m] != arr[-1] ))
do
s=${arr[m++]}
echo $s

file1="${prefix}${s:0:1}.txt"
echo $file1
file2="${prefix}${s:1:1}.txt"
echo $file2
file3="${prefix}${s:2:1}.txt"
echo $file3
file4="${prefix}${s:3:1}.txt"
echo $file4
echo "---"

    # main read loop
    while IFS= read -r word1
    do
        while IFS= read -r word2
        do
            while IFS= read -r word3
            do
                while IFS= read -r word4
                do
                    vord="${word1}${word2}${word3}${word4}"
                    #echo $vord
                    for constr in "${map[@]}"; do
                        if [[ ${constr:1:1} == "=" && ${vord:${constr:0:1}:1} != ${vord:${constr:2:1}:1} ]]; then
                        break
                        fi
                        if [[ ${constr:1:1} == "#" && ${vord:${constr:0:1}:1} == ${vord:${constr:2:1}:1} ]]; then
                        break
                        fi
                    done
                    if [[ ${constr} == "$" ]]; then
                    #echo $constr >> $output
                    echo "$pattern$vord" >> $output
                    fi
                done < "$file4"
            done < "$file3"
        done < "$file2"
    done < "$file1"
done
```

We are interested in all matches and are collecting them in an output file called "matchesABBBB.txt". The result is a database of all word combinations in Biblical Hebrew/Aramaic that match this pattern (see picture below). We use it for statistical analysis.

See below some concrete evaluations (please note that for Hebrew/Aramaic, the reading direction is right to left, whereas for convenience, the transliteration is left to right).

1. **_ccc9** (max 4 words; arbitrary last glyph in a 5-glyphs vord; **all** words match)

There were 5,233,545 matches out of 4,085,709,095 checks (~0.128%) in all word length-combinations: 6000, 5100, 4200, 3300, 2400, 1500, 4110, 3210, 2310, 1410, 3120, 2220, 1320, 2130, 1230, 1140, 2211, 1311, 2121, 1221, 1131, 1212, and 1122. We are interested in the most frequent chain BBB and the most frequent first letter within the vord (in the yellow background, we show the top 5 most frequent letters and chains of letters, and their percentage of occurrence among all matches).

Examples ABBBC	Freq. _BBB_	%	Freq. A____	%
lhhhy	**1217065 lll**		**482476 l**	
lhhhy	**1043278 rrr**		**481622 r**	
lhhhy	**629764 nnn**		**438809 d**	
lhhhk	**437352 ddd**		**405844 y**	
lhhhm	**410725 ššš**	**~71%**	**383563 b**	**~42%**
lhhhm	371976 zzz		369230 m	
lhhhn	350627 mmm		305108 š	
lhhhn	217852 fff		287751 z	
lhhhc	212093 xxx		266531 t	
lhhhf	103097 ggg		260354 x	
lhhhf	54396 ccc		234745 &	

2. **889** (max 3 words; **all** words match)

There were 99,231 matches out of 1,552,661 checks (~ 6.4%) in all word length-combinations: 3000 2100 1200 1110.

Examples ABB	Freq. _BB	%	Freq. A__	%
wnn	**26024 ll**		**10847 l**	
ldd	**18956 rr**		**10486 r**	
lhh	**10794 dd**		**8632 d**	
mll	**6770 yy**		**7434 y**	
fll	**5764 mm**	**~69%**	**6926 b**	**~45%**
rnn	5753 bb		6824 m	
rnn	4255 šš		5932 š	
&nn	4093 zz		5655 z	
&nn	3082 ff		4909 t	
bdd	2881 xx		4822 x	
bdd	2762 &&		4685 f	
Bdd	2753 nn		4594 &	

3. **cccco** (e. g. folio 66r; max 4 words; **all** words match)

There were 292,689 matches out of 2,358,025,751 checks (~ 0.012%) in all word length-combinations: 5000 4100 3200 2300 1400 2210 1310 2120 1220 1130 1121.

Examples ABBBB	Freq. _BBBB	%	Freq. A____	%
hllll	**122608 llll**		**26772 r**	
hllll	**67372 rrrr**		**23921 d**	
hllll	**24958 dddd**		**22764 y**	
hmmmm	**19522 nnnn**		**22067 l**	
hmmmm	**13827 šššš**	**~85%**	**22050 m**	**~40%**
hrrrr	13271 mmmm		21666 b	
hrrrr	12855 zzzz		18095 š	
hrrrr	6904 ffff		15972 z	
zxxxx	6087 xxxx		15004 t	
zxxxx	1564 gggg		14224 x	
zllll	1112 hhhh		13077 f	
zllll	767 cccc		12375 &	

When analysing longer vords, these simple routines take too long (on >>1 billion checks) to evaluate all matches. We can reduce the number of checks by limiting the pattern recognition to unique word strings only. In this case, we reduce the number of one-letter words from 108 to 20. This may lead to an overrepresentation of word combinations with words having only one meaning associated with an individual string but still provides a good estimation of the frequencies of letter distribution within particular patterns.

4. **9cccc9** (e. g. folio 102v part1; max 4 words; **unique** words match)

There were 1,081 matches out of 302,662,492 checks (~ 3.57 e-4 %) in all word length-combinations: 6000 5100 4200 3300 2400 1500 4110 3210 2310 1410 3120 2220 1320 2130 1230 1140 2211 1311 2121 1221 1131 1212 1122.

Examples ABBBBA	Freq. _BBBB_	%	Freq. A____A	%
bddddb	**112 nnnn**		**91 r**	
blllllb	**110 llll**		**74 y**	
bccccb	**107 rrrr**		**70 n**	
bßßßßßb	**102 dddd**		**70 m**	
bqqqqb	**88 mmmm**	**~48%**	**63 l**	**~34%**
brrrrb	79 šššš		62 d	
glllllg	78 qqqq		61 b	
grrrrg	62 ffff		59 š	
dmmmmd	61 xxxx		54 x	
dnnnnd	61 ßßßß		52 q	
drrrrd	55 gggg		52 h	

5. **898989** (e. g. folio 14v; max 4 words; **unique** words match)

There were 372 matches out of 450,188,289 checks (~ 8.26 e-5 %) in all word length-combinations: 6000 5100 4200 3300 2400 1500 4110 3210 2310 1410 3120 2220 1320 2130 1230 1140 3111 2211 1311 2121 1221 1131 2112 1212 1122 1113.

Examples ABABAB	Freq. A_A_A_	%	Freq. _B_B_B	%
dmdmdm	**59 y**		**54 y**	
ymymym	**33 r**		**37 r**	
glglgl	**30 m**		**36 l**	
grgrgr	**29 n**		**35 m**	
drdrdr	**27 h**	**~48%**	**31 n**	**~52%**
hbhbhb	25 š		26 w	
hrhrhr	23 l		25 h	
zlzlzl	20 w		23 š	
x&x&x&	18 d		22 d	
ydydyd	16 g		17 b	
yšyšyš	13 ß		12 &	

8 Interim Results

As for the chains of repeated letters often starting from the second position within a vord there exists a group of letters (l, r, d, m, n, **š**) in written biblical Aramaic/Hebrew which covers up to 85% of all matches in the investigated patterns. This is a strong indication for potential substitution of the glyphs **c** and **8** by candidates from this group.

Here the low character entropy of Voynichese purchased by uniform letter chains, which often puzzled the Voynichese analysts, turns out to be an obfuscation weakness, because it clearly favours specific letters for repetitions.

Furthermore, the rare combination **898989** shows the preference for the letters y and r for both positions. The regular occurrence of the letter y in the top scores for the first position makes it

a good candidate for **9**. Interestingly, y isn't a top choice for the longer chains of repeated letters other than the letters from the group (l, r, d, m, n, **š**), which occur with a high frequency in both, the first position and the repeated letters chains.

9 Pattern Matching (restricted evaluations)

With the above results we can assert restrictions to glyph mappings. For that we need to amend the bash routine. We introduce a new constraint (element of a set of letters) for specified positions within the vords. E. g., we define an array for the pattern ABCCDE (like **o$\mathcal{H}$cc89**) and restrict the second, third and fourth position exclusive to the letters l, r, d. In the definition of the *set* array, we use the character '$' to indicate an unrestricted position (the last '$' is a terminal sign):

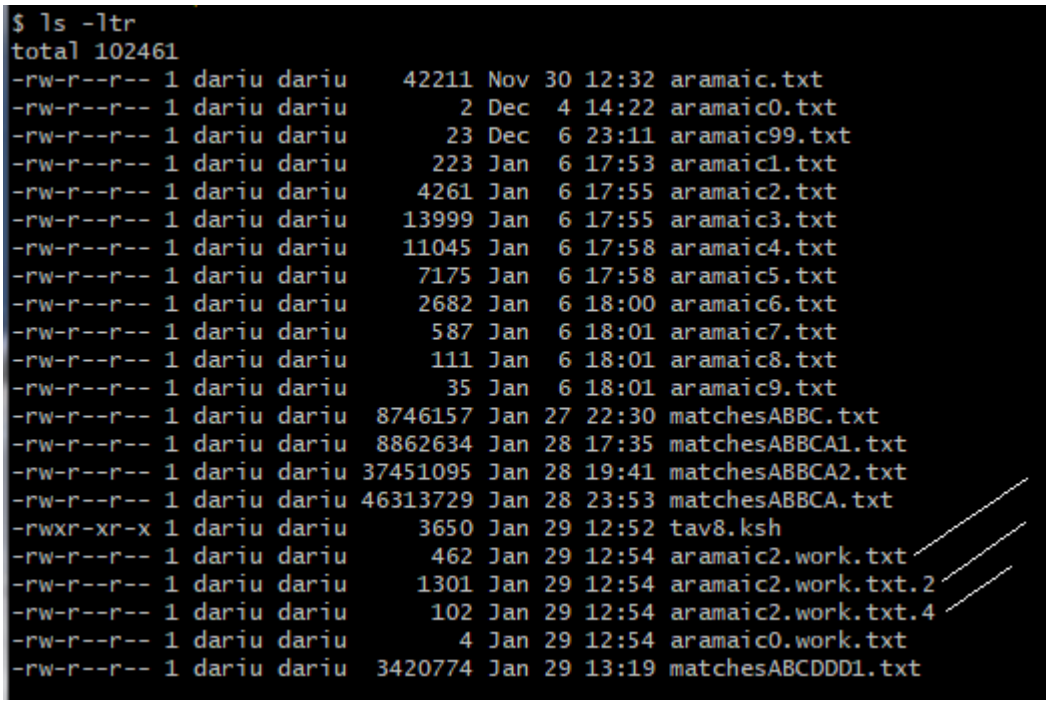
typeset -a set=($ lrd lrd lrd $ $ $)

```
#!/bin/bash
#pattern match for ABCCDE (4 words max), restricted by set constraints
typeset -a map=(0#1 0#2 0#4 0#5 1#2 1#4 1#5 2=3 2#4 2#5 4#5 $)
typeset -a arr=(6000 5100 4200 3300 2400 1500 4110 3210 2310 1410 3120 2220 1320 2130 1230
1140 3111 2211 1311 2121 1221 1131 1212 1122 1113 0000)
typeset -a set=($ lrd lrd lrd $ $ $)

typeset -i j=0 k=0 m=0 n=0
prefix=aramaic
pattern="ABCCDE-"
output=matchesABCCDE.txt

#main while loop

while (( arr[m] != arr[-1] ))
do
   s=${arr[m++]}
   t=${arr[m-1]}
   echo "word length combination: "$s

   # prepare working files

   for wfile in ${prefix}*.work.txt*; do
      [ -f "$wfile" ] && rm $wfile
   done
   cp aramaic0.txt aramaic0.work.txt
   j=0
   n=0

   for k in 0 1 2 3; do
      j=$((j+n))
      n=${t:$k:1}
      condition="[[ 1 == 1 "
      for (( i=0; i<n; i++ )); do
         if [[ "${set[j+i]}" != "$" ]]; then
         condition=$condition"&& ${set[$((j+i))]} =~ \${word:$i:1} "
         fi
      done
      condition=$condition"]]"

      file="${prefix}${s:$k:1}.txt"
```

```
        wfile="${prefix}${s:$k:1}.work.txt"
        if [ -f "$wfile" ] && [ "$wfile" != "aramaic0.work.txt" ]; then wfile=$wfile".$j"; fi

        while IFS= read -r word
        do
           if eval $condition; then
           echo "$word" >> $wfile
           fi
        done < "$file"
        [ ! -e "$wfile" ] && touch $wfile

        case "$k" in
           0) wfile1=$wfile;;
           1) wfile2=$wfile;;
           2) wfile3=$wfile;;
           3) wfile4=$wfile;;
           *) ;;
        esac
        echo "$file -> $condition -> $wfile"

    done

    # write output

    while IFS= read -r word1
    do
       while IFS= read -r word2
       do
          while IFS= read -r word3
          do
             while IFS= read -r word4
             do
             vord="${word1}${word2}${word3}${word4}"
             #echo $vord
                for constr in "${map[@]}"; do
                if [[ ${constr:1:1} == "=" && ${vord:${constr:0:1}:1} != ${vord:${constr:2:1}:1} ]]; then
                break
                fi
                if [[ ${constr:1:1} == "#" && ${vord:${constr:0:1}:1} == ${vord:${constr:2:1}:1} ]]; then
                break
                fi
                done
                if [[ ${constr} == "$" ]]; then echo "$pattern$vord" >> $output; fi
             done < "$wfile4"
          done < "$wfile3"
       done < "$wfile2"
    done < "$wfile1"
done
```

In the first part of the main loop, which scans over all word length combinations for a 6-glyph vord, we prepare 4 working files in each loop, as we assume for such vords to contain no more than 4 plaintext words. These working files are generated considering the positional restrictions defined in the *set* constraint array. They are therefore smaller than the original source files and reduce the runtime[3] for matching. In the second part of the main loop, we write the pattern matches based on the working files into the output file. Again, we generate a database for further statistical analysis.

[3] runtimes for billions of checks can nonetheless be many hours or even many days



The above working files were created during the evaluation of the word-length combination 2220 for a 6-glyph vord.

6. **o$\mathcal{H}$cc89** (e. g. folio 84v; max 4 words; **restricted** words match: $ lrd lrd lrd $ $)

There were 135,744,082 matches in all word length-combinations: 6000 5100 4200 3300 2400 1500 4110 3210 2310 1410 3120 2220 1320 2130 1230 1140 3111 2211 1311 2121 1221 1131 1212 1122 1113.

ABCCDE _(dlr)(dlr)(dlr)__	**Freq. ____D_**	**%**	**Freq. _____E**	**%**
srllm&	**14203659 b**		**10062859 b**	
bdllwn	**12161367 m**		**10040867 m**	
bdllwn	**11724943 h**		**9865506 y**	
bdllwn	**9580210 y**		**9100310 n**	
bdllwš	**8573683 n**	**~42%**	**8390646 l**	**~38%**
trllxš	7845063 š		8190510 š	
trllxš	6624786 f		7380029 f	
trllyh	6578837 g		7284648 h	
trllyš	6470585 k		7233629 &	
trllkd	5782895 x		7094446 z	
trllkd	5265436 ß		6423824 t	
trllkh	5257019 t		6394293 x	
trllmd	5012901 z		5382310 d	

7. **o𝓗cc89** (e. g. folio 84v; max 4 words; **restricted** words match: y r ld ld $ $)

There were 3,380,406 matches in all word length-combinations: 6000 5100 4200 3300 2400 1500 4110 3210 2310 1410 3120 2220 1320 2130 1230 1140 3111 2211 1311 2121 1221 1131 1212 1122 1113.

ABCCDE yr(dl)(dl)__	Freq. ____D_	%	Freq. _____E	%
yrddwl	**380176 m**		**298740 b**	
yrddwm	**343944 h**		**285488 n**	
yrddwn	**269626 b**		**280216 m**	
yrddwc	**265554 n**		**235584 l**	
yrddwf	**209358 š**	**~45%**	**221800 h**	**~42%**
yrddwš	186448 t		215136 d	
yrddw&	180856 g		211696 š	
yrddwh	162154 f		190476 f	
yrddwß	150066 x		187400 &	
yrddwq	140288 z		164960 z	
yrllhb	132746 w		159960 t	

8. **98a89** (e. g. folio 7r; max 4 words; **restricted** words match: y dlr $ dlr y)

There were 983,195 matches in all word length-combinations: 6000 5100 4200 3300 2400 1500 4110 3210 2310 1410 3120 2220 1320 2130 1230 1140 3111 2211 1311 2121 1221 1131 1212 1122 1113.

ABCBA y(dlr)_(dlr)y	Freq. __C__	%
ylhly	**113669 b**	
ylmly	**91294 m**	
ylnly	**64390 š**	
yl&ly	**54742 d**	
ydhdy	**53184 l**	**~40%**
yldly	49318 z	
ylkly	48946 f	
yrhry	47481 r	
yrdry	44215 x	
yrxry	44139 h	
yrtry	43868 t	
yrkry	39843 g	
yrqry	39004 ß	

In the 3 examinations above we restricted the occurrences of **9, 8** and **c** to (y, d, l, r). We aim to find the candidates for the very common vord ending **o<gallows>...** (like in **o𝓗** ...). With a less restrictive reduction - (d, l, r) for **9, 8** - we get (b, n, m, y) for **o** and (b, n, m, y, h) for the gallows with a probability of around 40%. A more restrictive reduction on **o𝓗cc89** results in (b, n, m, h) for **o** and (b, n, m, h, **š**) for the gallows with a slightly increased probability of

around 43%. Eventually we asked for a glyph surrounded by a pair of already allocated glyphs **9**8/8**9** like in the rare vord **9**8a8**9**. Here we get (b, m, š, d, l) with a probability of ~40%. The glyph **a** occurs very often in combinations like **aɯ𝔘** or **aɯ𝔘**. Our strategy here is to consider the more abnormal strings, which are nonetheless valid Voynichese vords.

10 Results and Conclusion

As for the chains of repeated glyphs **c** and **8**, we recognized a group of letters (l, d, r, n, m, š) with a substitution probability of up to ~85% in the examined plaintext languages. This is an impressively high likelihood, so we can limit here our investigation to this group. Regarding the first glyph before such chains, mostly **9**, the group is (l, d, r, y, b) with a probability of up to ~45%. Additionally, the rare vord **8**9**8**9**8**9 suggests the most likelihood for (y, r) in both positions. We then restricted the glyphs **8**, **9**, and **c** to the top candidates from these groups to find the candidates for the frequent combination **o<gallows>…** like in **o𝓗cc8**9**.** The result is the group (b, h, n, m, š) for both positions with a probability of up to ~45% of all matches. A restrained glyph **a** surrounded by **8**9 generates the highest probabilities for the letters (b, m, š, d, l) up to ~40%. At the end of this basic syntactical analysis, we can generate a list of the prevalent glyphs with their most probable substitutions in the considered plaintext languages (partially mutually exclusive, with the favourite substitution in bold letter):

c	**l** d r
8	**r** l d
9	**y** b
o	**b** h n m
𝓗	**b** h n m
a	**m** š

In this first analysis, we left the possible semantics and narration completely out of scope. Simple combinations of words won't constitute meaningful sentences for the vast majority. The rationale here is that the probability of finding grammatically correct and meaningful sentences is higher among the more frequent combinations than among the less frequent ones. Our yield is a plan for which substitutions to consider with priority when it comes to semantical and narrative examinations. In this way, we hope to find out further substitutions and the function of specifics of Voynichese like the connection between glyphs.

After this exhaustive example of a reasonable usage of constraint satisfaction, we will now return to the theoretical disquisition on computability.