The Enhanced Versions of the Program "Ka Minimalnim Parovima" (Towards Minimal Pairs)

UDC 81'322.2
DOI 10.18485/infotheca.2022.22.1.1

Danilo Aleksić
danilo.aleksic@fil.bg.ac.rs
University of Belgrade
Faculty of Philology
Belgrade, Serbia

Lazar Mrkela
lazar.mrkela@metropolitan.ac.rs
Belgrade Metropolitan
University, Faculty of
Information Technologies
Belgrade, Serbia

ABSTRACT: The program KaMP finds word pairs whose members are segmentally (in terms of speech) different only by two selected factors (Deza and Deza 2016, 215), each factor with length 1 or more, e.g. *pêć* ∼ *pêt*, *fȉlma* ∼ *fȋrma*, *istòrizovati* ∼ *majòrizovati*, *pȅsničkī* ∼ *polìtičkī*. The paper introduces the faster variants of KaMP with improved sorting and with a supplementary mode.

KEYWORDS: phonetics, phonology, natural language processing, corpus linguistics, Python.

PAPER SUBMITTED: 11 January 2022
PAPER ACCEPTED: 11 May 2022

1 Introduction

According to (Bugarski 2003, 128), minimal pairs are pairs in which two semantically distinct words formally differ in one phoneme only, e.g *bȁs* ∼ *čȁs*.[1] Ignoring prosody and letter case, in a Serbian corpus, the program Ka minimalnim parovima (Towards Minimal Pairs; Алексић and Шандрих 2021) finds word pairs whose members formally differ from each other by selected substrings[2] only. The corpus needs to be UTF-8 encoded. Apart

1. Ivić (1961–1962, 75) mentions prosodic systems with thousands of minimal pairs of words ie. pairs of forms differentiated by prosodic contrasts exclusively. Such pairs are *vȉla* ∼ *víla*, *lôza* (the genitive form of *loz* 'lottery ticket') ∼ *lòza* etc.

2. Globally it holds true that every string is a substring of itself (Partee, Meulen, and Wall 1993, 433; Singh 2009, 33) and that a string can be of length 1 (Partee, Meulen, and Wall 1993, 432; Python 2021b). Therefore, it would not be a misnomer to refer as a substring to (i) the string `"ima"` from the regular expression `\b\S*ima\S*\b` observed in relation to a match `"ima"`, nor to (ii) the character `"a"`.

from the selected substrings, the "words" can contain (i) characters from `"A"` to `"Z"`, from `"a"` to `"z"` and from `"Ć"` to `"ž"` in the corresponding Unicode charts and (ii) hyphens in medial position.

Content of the input file	**Selected substrings**	**String for the output**
`"Klima-uređaji pre klima-uređaja"`	`"a", "i"`	`"klima-uređaja ∼ Klima-uređaji"` (or `"Klima-uređaji ∼ klima-uređaja"`)
`"α-čestica, α-čestice, α-čestici"`	`"a", "e"`	`"čestica ∼ čestice"`
`"α-čestica, β-čestica"`	`"α", "β"`	`"α-čestica ∼ β-čestica"`

Table 1. KaMP: Examples of input and output

The program can be of use to teachers of Serbian as a foreign language and to linguists (Алексић and Шандрих 2021, 574–75).

Field	**Selected substrings**	**Utilization**
Teaching Serbian as a foreign language	`"c", "č"`	Basis for a task in an exercise: "*C* or *č*? 1) Šta bi ti uradio, dragi čitao_e? Naše novine će poštovati svoje čitao_e. 2) [...]"
Derivatology	`"auto", "samo"`	Data on competition between the segments *auto-* and *samo-*.

Table 2. KaMP: Examples of utilization

In the present paper the authors are publishing and commenting on the improved versions of KaMP which they built, KaMP 2 and KaMP 2.1.[3]

3. V. Appendix 1.

KaMP 2 and KaMP 2.1 are natural language processing tools if natural language processing is taken “in a wide sense to cover any kind of computer manipulation of natural language” (Bird, Klein, and Loper 2009, ix), because the two programs do not accent even a small number of all Serbian words, but process Serbian language superficially. Being somewhat adapted to searching large corpora, the new KaMPs are modest contributions to corpus linguistics as well if it is defined e.g. as “the computer-aided analysis of very extensive collections of transcribed utterances or written texts” (McEnery and Hardie 2012, i).

KaMP 2 and KaMP 2.1 were coded in Python 3.8.2 (Python 2021a). Python is “a high-level, interpreted, general-purpose programming language” (Pajankar 2020, 52). This language is “both elegant and pragmatic, both simple and powerful”; “it’s suitable for programming novices as well as great for experts, too” (Martelli, Ravenscroft, and Holden 2017, ix). Python “is becoming more and more popular, and in 2017 it became the most popular language in the world according to IEEE Spectrum” (Shovic and Simpson 2021, 1). Python is “the most widely used language for natural language processing” (Antić 2021, vii). It “may be expected” that Python be “*slow* as compared to compiled languages”, but it is faster “[i]f you start the clock to account for developer time, not just code runtime” (Unpingco 2021, 2). Python was created by the Dutch programmer Guido van Rossum in the late 1980s (Cicolani 2021, 41; Rajagopalan 2021, 1).

2 Similar resources

Four tools for finding minimal pairs or “phonological neighbours” (Mairano and Calabrò 2016, 258) which were written before KaMP are listed in (Алексић and Шандрих 2021, 569). The Python 3 package Minpair (PyPI 2021) and the short program in Python 2.7 from the page (Stack Overflow 2021a) can be added to that list.

Minpair looks for “minimal pairs (and minimal sets) for [only monosyllabic – D. A.] US English words”. The user selects two or more “vowel phonological element[s]” by which the members of the minimal pairs or minimal sets will differ. Minpair (A) uses `defaultdict` to group the words by the accompanying transcriptions in which (B) the package replaced the chosen vowels with a dot by means of a regular expression and the `enumerate()` function.

The approach A has a general parallel, but somewhat more efficient (v. Appendix 2), in KaMP 2.1. KaMP 2.1 pairs words by means of a standard dictionary (v. Appendix 1).

The approach B has a general parallel, but much more efficient (v. Appendix 3), in KaMP 2 and KaMP 2.1. KaMP 2 and KaMP 2.1 replace the selected elements with the special string by means of the methods `str.replace()` and `str.format()` (v. Appendix 1).

`cmudict` **entry**	**Tuple for grouping**

`("cat", ["K", "AE1", "T"])`
`("coat", ["K", "OW1", "T"])`
	`("K", ".", "T")`

Table 3. Minpair: An example of input and of the tuple for grouping (if the selected vowels are `"AE"` and `"OW"`)

```
# Minpair: Examples of use 1 and 2
import minpair
print(minpair.vowel_minpair(["AO", "ER"])[12:13])
# Output: [{'AO': 'saw', 'ER': 'sir'}]

print(minpair.vowel_minpair(["AA", "AO", "EH"])[6:7])
# Output: [{'AO': 'dawn', 'EH': 'den', 'AA': 'don'}]
```

Part(s) of speech can also be chosen by the user.

```
# Minpair: Examples of use 3, 4 and 5
import minpair
print(minpair.generator(pos=["ADV"]).vowel_minpair(
        ["AH", "EH"]))
# Output: [{'AH': 'once', 'EH': 'whence'}]

print(minpair.generator(
    pos=["ADJ", "VERB"]).vowel_minpair(
    ["AE", "IH"])[:1])
# Output: [{'AE': 'bad', 'IH': 'bid'}]

print(minpair.generator(
    pos=["ADJ", "VERB"]).vowel_minpair(
    ["AE", "IH"])[22:23])
# Output: [{'AE': 'sang', 'IH': 'sing'}]
```

The word source(s) cannot be chosen by the user. Minpair "depends on a few NLTK's corpora, namely: *brown*, *cmudict*, *universal_ tagset*, and *words* corpus".

The code from the page (Stack Overflow 2021a) pairs strings which differ by one character and have the same length. The strings must be inside e.g. a list, but they do not have to meet any natural language conditions (they do not have to come from a specific language or be written in a specific script, or even consist of alphabetic characters). During the execution of the code, every input string is compared to every subsequent input string, character by character. If all characters but one are the same, the pair of strings is printed.

```
# (Stack Overflow 2021a)
for n1,word1 in enumerate(wordlist):
    for word2 in wordlist[n1+1:]:
        if len(word1)==len(word2):
            ndiff=0
            for n,letter in enumerate(word1):
                if word2[n]!=letter:
                    ndiff+=1
            if ndiff==1:
                print word1, word2

"""The program from the page (Stack Overflow 2021a): Example of use 1
(added by D. A.)
Input: ["kula", "kule", "kuli", "kulom"]
Output:
kula kule
kula kuli
kule kuli
"""
```

This code is case sensitive in handling all characters except the differential one.

```
"""The program from the page (Stack Overflow 2021a):
Examples of use 2 and 3
Input: ["kula", "kulE", "kuli", "kulom"]
Output:
kula kulE
kula kuli
kulE kuli

```

```
Input: ["kula", "Kule", "kuli", "kulom"]
Output:
kula kuli
"""
```

When the first string in the input list was followed by its duplicate, the same pair was printed twice.

```
"""The program from the page (Stack Overflow 2021a): Example of use 4
Input: ["kula", "kula", "kule", "kulom"]
Output:
kula kule
kula kule
"""
```

	Minpair	**(Stack Overflow 2021a)**	**KaMP 2 and KaMP 2.1**
The tool prints such string pairs in each of which the strings differ by **any** character in the given position.	×	✓	×
The differential elements are chosen by the user.	✓	×	✓
The number of selected differential elements does not have to be 2.	✓		×
The selected differential elements do not have to be vowels.	×		✓
The words do not have to differ by a sequence of only one phoneme or of only one character.	×	×	✓
The input is chosen by the user.	×	✓	✓
The input does not have to be tokenized.		×	✓

Table 4. KaMP 2 / KaMP 2.1 compared to similar tools

3 Notable new characteristics of KaMP 2 and KaMP 2.1

In **the preparatory part of the algorithm**, tuples for the comparison of words are formed, such as `("Avali", "avali", "◸v◸li")`, `("Požeškom", "požeškom", "pož◸škom")`, `("sekretarijata", "sekretarijata", "s◸kr◸t◸rij◸t◸")`. The first member of the comparison tuple is a word that contains one or both selected substrings. The second member is the casefolded first member. The third member is the second member in which at least one instance of the first selected substring or of the second selected substring has been replaced with the string `"◸"`[4]. One excerpted word can have one word with the replacement of the selected substrings (see Table 5) or, (i) when the selected substrings have an "overlap" (Lothaire 2005, 7) or (ii) when one selected substring is "a proper substring" (Böckenhauer and Bongartz 2007, 24) of the other selected substring, more than one word with the replacement of a selected substring (see Table 6).

	Comparison strings	
Excerpted word	**Casefolded excerpted word**	**Casefolded excerpted word in which the selected substrings were replaced**
`"Knjiga"`	`"knjiga"`	`"knjig◸"`
`"knjigu"`	`"knjigu"`	`"knjig◸"`
`"sveska"`	`"sveska"`	`"svesk◸"`
`"SVESKU"`	`"svesku"`	`"svesk◸"`
`"računaljku"`	`"računaljku"`	`"r◸č◸n◸ljk◸"`

Table 5. KaMP 2 / KaMP 2.1: Examples of comparison strings (if the selected substrings are `"a"` and `"u"`)

In **the main part of the algorithm**, the formed tuples are compared. KaMP 2 generates all possible two-member combinations of the tuples with the first selected substring and the tuples with the second selected substring

4. It was chosen because it is conspicuous and relatively rare. Of course, those traits are present in many other strings.

Excerpted word	**Comparison strings**	
	Casefolded excerpted word	**Casefolded excerpted word in which one of the selected substrings was replaced**
`"ONA"`	`"ona"`	`"on⸢"`
`"Onima"`	`"onima"`	`"on⸢"`
`"Onima"`	`"onima"`	`"onim⸢"`
`"onimima"`	`"onimima"`	`"onim⸢"`
`"onimima"`	`"onimima"`	`"onimim⸢"`

Table 6. KaMP 2 / KaMP 2.1: Examples of comparison strings (if the selected substrings are `"a"` and `"ima"`)

and then skips the unwanted combinations. This program obtains the possible two-member combinations by calculating the Cartesian product[5] of the two groups of tuples. KaMP 2.1 transforms the tuples with the second selected substring into a hash map (table) and checks whether it contains the words with replacement taken from the tuples with the first selected substring. The map's key is the word with replacement, and the value of the map is a map of the words from which that same key is obtained. – KaMP 2 and KaMP 2.1 print those pairs of excerpted words whose members (i) differ when casefolded and (ii) match by the words with replacement, i.e. pairs of those words which differ by the selected substrings only. For example, if the selected substrings are `"a"` and `"u"`, and the input file only contains the string `"Knjiga, knjigu, sveska, SVESKU"`, KaMP 2 and KaMP 2.1 will not print the strings `"Knjiga ∼ SVESKU"` and `"knjigu ∼ sveska"`, since the string `"knjig⸢"` is not equal to the string `"svesk⸢"`. The programs will print the strings `"Knjiga ∼ knjigu"` and `"sveska ∼ SVESKU"`.

KaMP 2 and KaMP 2.1 have (A) a mode in which they ignore case but favor strings of lowercase letters and (B) a mode in which e.g. excerpted strings `"vitraž"` and `"Vitraž"` would be processed as separate words (see Table 7). The reason is the justified comment from (Алексић and Шандрих 2021, 574) which raises the question of the importance of case. For example,

5. For example, the Cartesian product of the set of strings `{"broj", "ulica"}` and the set of strings `{"MESTO", "OPŠTINA"}` is the set of tuples of strings `{("broj", "MESTO"), ("broj", "OPŠTINA"), ("ulica", "MESTO"), ("ulica", "OPŠTINA")}`.

in teaching Serbian as a foreign language proper nouns sometimes have priority over non-proper words. The name *Čak* (*Beri*, *Noris*...) is suitable for a pronunciation exercise with photos; what kind of photo would depict the meaning of the uninflected word *čak*? In the mode B, from the corpus `POL`, KaMP 2 and KaMP 2.1 extract not only the pair `"čak ∼ Žak"`, but also the pair `"Čak ∼ Žak"` (alongside `"Čak ∼ ŽAK"` etc.).[6]

Content of the input file	`"EUPRAVE, eUprave, euprave, EUPRAVA, eUprava, euprava"`
String for the output of KaMP	`"EUPRAVA ∼ EUPRAVE"` (or `"EUPRAVE ∼ EUPRAVA"`)
String for the output of KaMP 2 and KaMP 2.1 in the mode A	`"euprava ∼ euprave"`
Strings for the output of KaMP 2 and KaMP 2.1 in the mode B	`"euprava ∼ euprave", "euprava ∼ eUprave", "euprava ∼ EUPRAVE", "eUprava ∼ euprave", "eUprava ∼ eUprave", "eUprava ∼ EUPRAVE", "EUPRAVA ∼ euprave", "EUPRAVA ∼ eUprave", "EUPRAVA ∼ EUPRAVE"`

Table 7. KaMP and KaMP 2 / KaMP 2.1: Case (in)sensitivity (if the selected substrings are `"a"` and `"e"`)

The function `corpus_segmentation()`[7] has been reorganized. It no longer reads the corpus using an infinite `while`-loop,[8] but, following the

6. Capital first letter is no guarantee that `"Čak"` can be a name in `POL` (because of sentences like `"Čak sam pronašao i kupca."`). An additional search proves that it can (`"Čak Blekvel"`, `"Čak Dejli"`, `"Čak Noris"`...).

7. This is a generator function which partitions the input corpus so that little RAM is used, in such a way as not to cut individual words apart (Алексић and Шандрих 2021, 572, 581).

8. Cf. the code which was added on May 14th 2021 to the answer which was posted to (Stack Overflow 2021b) on June 11th 2015.

example of the recommended way to call a function until a sentinel value from (Hettinger 2021, 12.27 and onwards), using a `for`-loop, which is “fast and beautiful”.

KaMP sorts the found pairs by the Unicode code positions of single letters, while KaMP 2 and KaMP 2.1 sort the found pairs by the positions of single letters in the strings `lower_alphabet` and `upper_alphabet` (see Table 8).

```
# The sorting strings in KaMP 2 and KaMP 2.1
lower_alphabet = "- ~abcčćdđefghijklmnopqrsštuvwxyzž"
upper_alphabet = "- ~ABCČĆDĐEFGHIJKLMNOPQRSŠTUVWXYZŽ"
```

Input list	`[ "nota ∼ note", "đaka ∼ đake", "Bač ∼ Beč" ]`
Input list after being sorted in the way KaMP sorts pairs	`[ "Bač ∼ Beč", "nota ∼ note", "đaka ∼ đake" ]`
Input list after being sorted in the way KaMP 2 and KaMP 2.1 sort pairs	`[ "Bač ∼ Beč", "đaka ∼ đake", "nota ∼ note" ]`

Table 8. KaMP and KaMP 2 / KaMP 2.1: Sorting of the pairs

Truth be told, the new KaMPs use the Unicode code positions as well when they sort pairs, but only for characters which the sorting strings do not contain (see Table 9).

Pair	**Sorting list**	**The origin of the number**
α	945	Unicode
-	0	The string
z	32	`lower_alphabet`
r	23	
a	3	
č	6	
e	10	
n	19	
j	15	
e	10	
	1	
∼	2	
	1	
β	946	Unicode
-	0	The string
z	32	`lower_alphabet`
r	23	
a	3	
č	6	
e	10	
n	19	
j	15	
e	10	

Table 9. KaMP 2 / KaMP 2.1: An example of the sorting list

KaMP 2 and KaMP 2.1 sort the members of every pair before joining them into an output string (e.g. `["knjigu", "Knjiga"]` → `["Knjiga", "knjigu"]`).

4 Execution speed

Speed was measured in Python 3.8.2, on Manjaro Linux, with a computer with the processor i5-11600K and two DDR4-3200 CL16 SDRAMs (16 GB each) and on the `POL` corpus, which has around 117,900,900 words from 223,308 texts from the *Politika* website (Алексић and Шандрих 2021, 575).[9]

5 A short assessment of the efficiency of KaMP 2[10]

The function which finds pairs in KaMP 2 is based on the Cartesian product of two lists. This approach is an elegant solution in terms of layout and complexity of the code, but it is not efficient enough in the case of lists with large numbers of elements, because of quadratic behavior. The problem is easily noticed in the experimental results, where significantly longer execution time is observed in the case of more frequent substrings (ma-va; cf. Table 10).

6 Further work

In real conditions, which may demand that this program be run on weaker computers, just reading the corpus and excerpting the words which contain the selected substrings can last for too long. For example, reading the corpus POL.xml on an older laptop computer (Acer Aspire 3, Intel Quad Core N3710, 4GB RAM) lasts up to approximately 15 minutes. The recommendation is that the option to create a disk-stored dictionary be added. This processing of the corpus would be conducted just once, and the dictionary would later be used to search for pairs by new substrings.

The next possible step in shortening the execution time is search parallelization. Nowadays, even the weaker computers have several "cores" in their processors (for example, the computer from the previous paragraph has 4 cores). That is why it is possible to execute some parts of the code in parallel and thus additionally speed up the program. A suggestion for simple parallelization is the division of one of the lists into n parts, and then the processing of those parts on separate processors (cores) in parallel. Seeing that the sequential version with hashing is already very efficient, the problem of the slow reading of the corpus should be solved first, and only then should further speeding up be considered.

9. All the results come from measuring the execution speed of the code from the Serbian version of this paper.

10. Sections 5 and 6 were written by L. Mrkela. The other sections (without the two sentences in Section 3 which only concern KaMP 2.1), Appendix 2 and Appendix 3 were written by D. Aleksić.

Selected substrings	**Speed in seconds (the average of five successive measurements)**				
	KaMP	**KaMP 2**		**KaMP 2.1**	
		Mode A	**Mode B**	**Mode A**	**Mode B**
`"č", "đ"`	334	199	246	67	66
`"dž", "đ"`	135	81	85		
`"nadnad", "supersuper"`	6639	**65.79**	**65.51**	66.38	66.09
`"ir", "zir"`	143	86	90	**67** (!)	66
`"ma", "va"`	2153	1257	1516		

Table 10. KaMP, KaMP 2 and KaMP 2.1: Execution speed

7 Conclusion

In comparison to KaMP, KaMP 2 and KaMP 2.1 achieve more in less time – they find virtually the same pairs and sort the pairs and the words inside the pairs in a better way.

KaMP 2.1 was inarguably faster than KaMP 2 in the majority of the investigated cases.

Appendix 1. KaMP 2 and KaMP 2.1[11]

```
"""KaMP 2.1 is a modified version of KaMP 2. KaMP 2 is
a modified version of KaMP.

The pairing in the functions KaMP_2_1_a() and KaMP_2_1_b()
was written by L. Mrkela, and the rest of the code
(with the functions excerp_(), proc_words_1() and proc_words_2())
was written by D. Aleksić.
"""


def main():
    from functools import partial
    from itertools import product
    import re
    import sys

    sys.stdout.reconfigure(encoding="utf-8")
    """V. (Алексић and Шандрих 2021, 580)."""
    letter_1 = "ma".casefold()
    letter_2 = "va".casefold()
    overlap_ = False
    case_diff = False
    sort_char = chr(1114111)
    lower_alphabet = "- ~abcčćdđefghijklmnopqrsštuvwxyzž"
    upper_alphabet = "- ~ABCČĆDĐEFGHIJKLMNOPQRSŠTUVWXYZŽ"

    def join_strings(*strings):
        return "".join(strings)

    def corpus_segmentation(
            corpus_, size_=8192, separator_="\n"):
        """Cf. (581).
        The function returns parts of the corpus which have
        the specified size. The cuts between the parts are
        made at the specified separator.
        """
```

11. The official documentation, from the site (Python 2021a), was the primary source. The other sources were indicated by directing to links from the References, by directing to the corresponding parts of (Алексић and Шандрих 2021) and by directing to a part of this paper.

```
        remainder_ = ""
        for piece_ in iter(
                partial(corpus_.read, size_), ""):
            """V. (Hettinger 2021, 12.27 and onwards)."""
            piece_ = join_strings(remainder_, piece_)
            if separator_ in piece_:
                pieces_ = piece_.rsplit(separator_, 1)
                """V. (W3Schools 2021)."""
                yield pieces_[0]
                remainder_ = pieces_[1]
            else:
                remainder_ = piece_
        if remainder_:
            yield remainder_

    def lower_first_1(word_):
        """The key for word sorting which favors the words
        consisting of lowercase letters.
        """
        if word_.islower():
            return "!"
        elif word_.istitle():
            return sort_char
        else:
            for letter_ in word_:
                if letter_.isupper():
                    word_ = word_.replace(
                        letter_, sort_char)
            return word_

    def lower_first_2(word_):
        """The key for sorting the found pairs.
        The pairs which contain less uppercase letters will
        be at the top of the list.
        """
        if word_.islower():
            return "!"
        else:
            word_ = word_.replace("~", "")
            if word_.istitle():
                return sort_char
            else:
                for letter_ in word_:
                    if letter_.isupper():
                        word_ = word_.replace(
                            letter_, sort_char)
                return word_

    def indexing_for_list(word_):
        """Cf. (Stack Overflow 2021c).
        The key for sorting in the specified order.
        """
        sort_list = []
        for letter_ in word_:
            if letter_ in lower_alphabet:
                sort_list.append(
                    lower_alphabet.index(letter_))
            elif letter_ in upper_alphabet:
                sort_list.append(
                    upper_alphabet.index(letter_))
            else:
                sort_list.append(ord(letter_))
        return sort_list

    def simple_word_repl(word_):
        """The selected substrings of words are replaced by
        the special string.
        """
        if letter_1 in word_ and letter_2 not in word_:
            word_with_repl = word_.replace(
                letter_1, "\u23B2")
        elif letter_1 not in word_ and letter_2 in word_:
            word_with_repl = word_.replace(
                letter_2, "\u23B2")
        elif letter_1 in word_ and letter_2 in word_:
            word_with_repl = word_.replace(
                letter_1, "\u23B2")
            word_with_repl = word_with_repl.replace(
                letter_2, "\u23B2")
        return (word_with_repl,)

    def complex_word_repl(word_, letter_):
        """The specified substrings in words are replaced
        with the special string.
        The cases when there is overlap between the selected
        substrings are covered.
        V. (Алексић and Шандрих 2021, 580--81).
        """
        output_set = set()
        word_for_proc = word_.replace(letter_, "{}")
        for combination_ in product(
                [letter_, "\u23B2"],
                repeat=word_for_proc.count("{}")):
            word_with_repls = word_for_proc.format(
                *combination_)
            if word_with_repls != word_:
                output_set.add(
                    word_with_repls)
        return output_set

    def letter_repl(word_):
        if not overlap_:
            return simple_word_repl(word_)
        else:
            set_ = set()
            set_.update(
                complex_word_repl(word_, letter_1),
                complex_word_repl(word_, letter_2))
            return set_

    def tokenization_():
        """The corpus is transformed into a dictionary of
        words which were gathered by means of a regular
        expression. The function enables the program to
        avoid the use of very complex regular expressions
        in the cases when the selected substrings are
        long (see Table 10).
        Cf. Section 6.
        """
        dict_ = {}
        with open(r"/home/.../POL.xml",
                  "r", encoding="utf-8") as corpus_:
            pieces_ = corpus_segmentation(corpus_)
            for piece_ in pieces_:
                matches_ = re.findall(
                    "[A-Za-zĆ-ž-\u00ad]+", piece_)
                """V. (573--74)."""
                for match_ in matches_:
                    word_ = match_.strip("-")
                    if "\u00ad" in word_:
                        word_ = word_.replace(
                            "\u00ad", "")
                        """V. (Алексић and Шандрих 2021, 581)."""
                    dict_[word_] = word_.casefold()
        return dict_

    def excerp_(letter_):
        """The words which contain the selected substring
        are taken from the dictionary into which the corpus
        has been transformed.
        """
        return (key_
                for key_, value_
                in corpus_dict.items()
                if letter_ in value_)

    def descartes(list_1, list_2):
        """The unwanted pairs are eliminated from the
        Cartesian product of the processed words.
        """
        return (
            (*sorted([b, e]), a, d)
            for (a, b, c), (d, e, f)
            in filter(
            lambda tuple_: tuple_[0][2] == tuple_[1][2]
                          and tuple_[0][1]
                          != tuple_[1][1],
            product(list_1, list_2, repeat=1)))

    def proc_words_1(gen):
        """The function returns tuples which contain words,
        casefolded words and words with replacement.
        This is a case insensitive function.
        """
        tuple_list = []
        counter_ = set()
        word_list = sorted(
            list(gen), key=lower_first_1)
        for word_ in word_list:
            casefold_word = corpus_dict[word_]
            if casefold_word not in counter_:
                counter_.add(casefold_word)
                for word_with_repl in letter_repl(
                        casefold_word):
                    tuple_list.append(
                        (word_, casefold_word,
                         word_with_repl))
        return tuple_list

    def proc_words_2(gen):
        """The function returns tuples which contain words,
        casefolded words and words with replacement.
        This is a case sensitive function.
        """
        tuple_list = []
        for word_ in gen:
            casefold_word = corpus_dict[word_]
            for word_with_repl in letter_repl(
                    casefold_word):
                tuple_list.append(
                    (word_, casefold_word,
                     word_with_repl))
        return tuple_list

    def KaMP_2_a():
        """Final case insensitive processing in KaMP 2.
        """
        counter_ = set()
        for tuple_ in descartes(
                proc_words_1(excerp_(
                    letter_1)),
                proc_words_1(excerp_(
                    letter_2))):
            if (tuple_[0], tuple_[1]) not in counter_:
                counter_.add((tuple_[0], tuple_[1]))
                final_set.add((tuple_[2], tuple_[3]))
        pair_list = [
            " ~ ".join(sorted(list(tuple_),
                              key=indexing_for_list))
            for tuple_ in final_set]
        pair_list.sort(key=indexing_for_list)
        for pair_ in pair_list:
            print(pair_)
        print("\n\tTHE NUMBER OF PAIRS:")
        print("\t\t", len(pair_list))
```

```python

    def KaMP_2_b():
        """Final case sensitive processing in KaMP 2.
        """
        final_set = {(tuple_[2], tuple_[3])
                        for tuple_ in descartes(
                    proc_words_2(excerp_(
                        letter_1)),
                    proc_words_2(excerp_(
                        letter_2)))}
        pair_list = [
            " ~ ".join(sorted(list(tuple_),
                              key=indexing_for_list))
            for tuple_ in final_set]
        pair_list = list(set(pair_list))
        pair_list.sort(key=lower_first_2)
        pair_list.sort(key=indexing_for_list)
        for pair_ in pair_list:
            print(pair_)
        print("\n\tTHE NUMBER OF PAIRS:")
        print("\t\t", len(pair_list))

    def KaMP_2_1_a():
        """Case insensitive pairing and final processing
        in KaMP 2.1.
        """
        list1 = proc_words_1(excerp_(
            letter_1))
        list2 = proc_words_1(excerp_(
            letter_2))
        map_ = {}
        for x in list2:
            if x[2] not in map_:
                map_[x[2]] = {}
            map_[x[2]][x[0]] = x[1]
        for tuple_ in list1:
            result = map_.get(tuple_[2])
            if result is not None:
                for k, v in result.items():
                    if (tuple_[1] != v and (k, tuple_[0])
                            not in final_set):
                        final_set.add((tuple_[0], k))
        list_ = []
        for pair_ in final_set:
            pair_ = list(pair_)
            pair_.sort(key=indexing_for_list)
            output_ = join_strings(
                pair_[0], " ~ ", pair_[1])
            list_.append(output_)
        list_.sort(key=indexing_for_list)
        for pair_ in list_:
            print(pair_)
        print("The number of pairs: ", len(list_))

    def KaMP_2_1_b():
        """Case sensitive pairing and final processing
        in KaMP 2.1.
        """
        list1 = proc_words_2(excerp_(
            letter_1))
        list2 = proc_words_2(excerp_(
            letter_2))
        map_ = {}
        for x in list2:
            if x[2] not in map_:
                map_[x[2]] = {}
            map_[x[2]][x[0]] = x[1]
        for tuple_ in list1:
            result = map_.get(tuple_[2])
            if result is not None:
                for k, v in result.items():
                    if (tuple_[1] != v and (k, tuple_[0])
                            not in final_set):
                        final_set.add((tuple_[0], k))
        list_ = []
        for pair_ in final_set:
            pair_ = list(pair_)
            pair_.sort(key=indexing_for_list)
            output_ = join_strings(
                pair_[0], " ~ ", pair_[1])
            list_.append(output_)
        list_.sort(key=lower_first_2)
        list_.sort(key=indexing_for_list)
        for pair_ in list_:
            print(pair_)
        print("The number of pairs: ", len(list_))

    if (letter_1[-1:] == letter_2[:1]
            or letter_1[:1] == letter_2[-1:]
            or (letter_1 in letter_2
                or letter_2 in letter_1)):
        overlap_ = True
    final_set = set()
    corpus_dict = tokenization_()
    if case_diff:
        KaMP_2_1_b()  # KaMP 2 calls the function KaMP_2_b().
    else:
        KaMP_2_1_a()  # KaMP 2 calls the function KaMP_2_a().


if __name__ == "__main__":
    main()
```

Appendix 2. Minpair vs. KaMP 2.1: Pairing speed

```
"""The Minpair approach.
"""
from collections import defaultdict

map_1 = defaultdict(lambda: {})
for x in list_2:
    map_1[x[2]][x[0]] = x[1]

"""The KaMP 2.1 approach.
"""
map_2 = {}
for x in list_2:
    if x[2] not in map_2:
        map_2[x[2]] = {}
    map_2[x[2]][x[0]] = x[1]

"""The input was a list of tuples like ("subsidiaries",
"subsidiaries", "subsidi.ri.s"). The list was made of
the words from cmudict which contain the substring "a"
and/or the substring "e".

The KaMP 2.1 approach proved itself around 7% faster in
```

```
Python 3.8.2 on the system which was described in
Section 4. The averages of 500 successive measurements
were compared (55 ms : 51 ms).
"""
```

Appendix 3. Minpair vs. KaMP 2 / KaMP 2.1: Replacement speed

```
"""The Minpair approach.
"""
vowels_regex = re.compile(r'^(?:%s)' % '|'.join(vowels))
matches = [vowels_regex.search(phone) for phone in word]
list_with_repl = []
for i, character in enumerate(word):
    for j, match in enumerate(matches):
        if i == j:
            if match:
                list_with_repl.append(".")
            else:
                list_with_repl.append(character)
string_with_repl = "".join(list_with_repl)

"""The KaMP 2 and KaMP 2.1 approach.
"""
word_with_repl = word_.replace(
    letter_1, ".")
word_with_repl = word_with_repl.replace(
    letter_2, ".")

"""The input consisted of the words from cmudict
which contain the substring "a" and/or the substring "e".

The KaMP 2 and KaMP 2.1 approach proved itself around
95% faster in Python 3.8.2 on the system which was described
in Section 4. The averages of 500 successive measurements
were compared (472 ms : 23 ms). However, it must be pointed out
that the code for replacing in Minpair gets a list and returns
a tuple (e.g. ["L", "UW", "S"] → ("L", ".", "S")), while both
the input and the output of the code for replacing in KaMP 2
and KaMP 2.1 are a string (e.g "teorijska" → "t◥orijsk◥").
"""
```

References

Antić, Zhenya. 2021. *Python Natural Language Processing Cookbook: Over 50 recipes to understand, analyze, and generate text for implementing language processing tasks.* Birmingham: Packt Publishing.

Bird, Steven, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python.* Sebastopol, CA: O'Reilly Media.

Böckenhauer, Hans-Joachim, and Dirk Bongartz. 2007. *Algorithmic Aspects of Bioinformatics.* Berlin: Springer.

Bugarski, Ranko. 2003. *Uvod u opštu lingvistiku.* 2nd ed. Beograd: Čigoja štampa.

Cicolani, Jeff. 2021. *Beginning Robotics with Raspberry Pi and Arduino: Using Python and OpenCV.* 2nd ed. Berkeley, CA: Apress.

Deza, Michel Marie, and Elena Deza. 2016. *Encyclopedia of Distances.* 4th ed. Berlin: Springer.

Hettinger, Raymond. 2021. "Transforming Code into Beautiful, Idiomatic Python." Accessed August 21, 2021. https://www.youtube.com/watch?v=OSGv2VnC0go.

Lothaire, M. 2005. *Applied Combinatorics on Words.* Cambridge: Cambridge University Press.

Mairano, Paolo, and Lidia Calabrò. 2016. "Are minimal pairs too few to be used in pronunciation classes?" In *La fonetica nell'apprendimento delle lingue: Phonetics and language learning,* edited by Renata Savy and Iolanda Alfano, 255–268. Milano: Officinaventuno.

Martelli, Alex, Anna Ravenscroft, and Steve Holden. 2017. *Python in a Nutshell.* 3rd ed. Sebastopol, CA: O'Reilly Media.

McEnery, Tony, and Andrew Hardie. 2012. *Corpus Linguistics: Method, Theory and Practice.* Cambridge: Cambridge University Press.

Pajankar, Ashwin. 2020. *Raspberry Pi Computer Vision Programming: Design and implement computer vision applications with Raspberry Pi, OpenCV, and Python 3.* 2nd ed. Birmingham: Packt Publishing.

Partee, Barbara H., Alice ter Meulen, and Robert E. Wall. 1993. *Mathematical Methods in Linguistics.* Dordrecht: Kluwer Academic Publishers.

PyPI. 2021. “minpair 0.1.3.” Accessed October 26, 2021. https://pypi.org/project/minpair.

Python. 2021a. Accessed August 21, 2021. https://www.python.org.

Python. 2021b. “Text Sequence Type — `str`.” Accessed August 21, 2021. https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str.

Rajagopalan, Gayathri. 2021. *A Python Data Analyst’s Toolkit: Learn Python and Python-based Libraries with Applications in Data Analysis and Statistics.* Berkeley, CA: Apress.

Shovic, John C., and Alan Simpson. 2021. *Python All-in-One For Dummies.* 2nd ed. Hoboken, NJ: John Wiley & Sons.

Singh, Arindama. 2009. *Elements of Computation Theory.* London: Springer.

Stack Overflow. 2021a. “Finding (phonological) minimal pairs with python.” Accessed August 31, 2021. https://stackoverflow.com/q/26157361.

Stack Overflow. 2021b. “Lazy Method for Reading Big File in Python?” Accessed August 31, 2021. https://stackoverflow.com/q/519633.

Stack Overflow. 2021c. “Sorting string values according to a custom alphabet in Python.” Accessed October 26, 2021. https://stackoverflow.com/q/26579392.

Unpingco, José. 2021. *Python Programming for Data Analysis.* Cham: Springer.

W3Schools. 2021. “Python String rsplit() Method.” Accessed October 26, 2021. https://www.w3schools.com/python/ref_string_rsplit.asp.

Алексић, Данило, and Бранислава Шандрих. 2021. “Аутоматска ексцерпција парова речи за учење изговора у настави српског као страног језика.” *Српски језик: студије српске и словенске* 26 (1): 567–584. ISSN: 0354-9259. https://doi.org/10.18485/sj.2021.26.1.32. http://doi.fil.bg.ac.rs/pdf/journals/sj/2021-1/sj-2021-26-1-32.pdf.

Ивић, Павле. 1961–1962. “Број прозодијских могућности у речи као карактеристика фонолошких система словенских језика.” *Јужнословенски филолог* 25: 75–113.