Efficient Algorithms for the Spoonerism Problem*

Hans-Joachim Böckenhauer[1], Juraj Hromkovič[1], Richard Královič[1,3], Tobias Mömke[1], and Kathleen Steinhöfel[2]

[1] Department of Computer Science, ETH Zurich, Switzerland,
{hjb, juraj.hromkovic, richard.kralovic, tobias.moemke}@inf.ethz.ch
[2] Department of Computer Science, King's College London, United Kingdom,
kathleen.steinhofel@kcl.ac.uk
[3] Department of Computer Science, Comenius University, Slovakia.

Abstract. A spoonerism is a sentence in some natural language where the swapping of two letters results in a new sentence with a different meaning. In this paper, we give some efficient algorithms for deciding whether a given sentence, made up from words of a given dictionary, is a spoonerism or not.

1 Introduction

It probably happened to most people that when speaking quickly one accidentally swapped two words of a sentence. If the resulting sentence still has a meaning, it might reveal a new meaning and may turn out funny. A *Spoonerism* is such an accidentally transposition of words or parts of words in a sentence. It is named after Reverend William Archibald Spooner (1844-1930). He was an English scholar who attended New College, Oxford, as an undergraduate in 1862, and remained there for over 60 years in various capacities. Before he ultimately became warden or president of the College, he was lecturing subjects such as history, philosophy, and divinity. Spooner was famous for his talks and lectures that are said to be full of these verbal slips in speech. The reason for these substitutions of phonetically similar parts is not silliness or nervousness but rather that the mind is so swift the tongue cannot keep up.

For a detailed biography of Reverend Spooner see [3] and for a brief history and some examples of the spoonerism see the February 1995 edition of the Reader's Digest Magazine. Here it says: 'Reverend Spooner's tendency to get words and sounds crossed up could happen at any time, but especially when he was agitated. He reprimanded one student for "fighting a liar in the quadrangle" and another who "hissed my mystery lecture." To the latter he added in disgust, "You have deliberately tasted two worms and you can leave Oxford by the town drain." (lighting a fire; missed my history lecture; wasted two terms; down train)'.

* Partially supported by VEGA 1/3106/06 and EPSRC EP/D062012/1

Many such examples have been attributed to Spooner. But a new biography of Spooner [3] suggest that most of them were actually invented by Spooner's students. So, the Oxfords Dictionary gives only one example of spoonerism ("weight of rages") that can be tracked back to Spooner and says: 'Many other Spoonerisms, such as those given in the previous editions of O.D.Q., are now known to be apocryphal.'

In French (contrepéterie) this play with word for amusement is also very popular. However traditionally the swap results in an often indecent meaning such that only the original part should be said. The sometimes hard task to find the swap revealing the funny meaning is left to readers or listeners.

As said before, the substitutions often rhyme. In German there are short rhymes that are based on the exchange of the last two stressed syllables (or parts). These are known as "Schüttelreim" (shaken rhyme). Examples are: "Ein Schornsteinfeger gegen Ruß / am besten steht im Regenguß." (A chimney sweeper avoids soot best when standing in the rain) and "Beim Zahnarzt in den Wartezimmern / hört man oft auch Zarte wimmern." (In dentist waiting rooms one often hears tender ones whimper). The latter example (and many more) can be found at de.wikiquote.org.

In Slovak and Czech, the spoonerism is known as "výmenka" (exchange riddle). An example is "úľ bez nálady – ľúbezná lady" (A hive in bad mood – lovable lady).

In psychological tests, spoonerisms have been used to analyze phonological awareness which is related to spelling abilities [1].

In string matching and analysis, transposition of letters is used as metric for the similarity of strings. For instance, the Jaro distance metric [4, 5] and the Jaro-Winkler distance [8] are string comparators that account for transpositions of single letters.

We consider here the problem of deciding whether a given word or sentence is a spoonerism, i.e., whether there exists a transposition of two letters such that the resulting string is a valid word according to a given dictionary or can be decomposed into valid words. The problem was introduced in an unpublished presentation [7]. Some sketch of the ideas of the presentation can be found in an unpublished manuscript [2]. All the algorithms and results achieved and presented there (see the comparison in Section 2) are disjoint from our results.

The problem can be formalized in the following way. A sentence s is given as a string, i.e., as a concatenation of its words. Its length n is the total number of letters in s. The second part of the input is a dictionary $\mathcal{D}$ of valid words. The task is to decide whether there exist two positions in the string such that a swap of the letters at these position leaves a string that can be decomposed into words of the dictionary. For example, by swapping the letters l and p in the phrase "a lack of pies" we get another meaningful phrase "a pack of lies" which is correctly spelt in English.

In this paper, we will give some efficient algorithms for deciding if a given sentence is a spoonerism and analyze their worst-case running times. In Section 2, we will fix our notion and present a formal definition of the problem. In

Section 3, we will present a first dynamic-programming approach for solving the spoonerism problem; two technically more involved algorithms with improved running times will be presented in Sections 4 and 5.

2 Preliminaries

Before we formally define the spoonerism problem, we fix some notation: For any alphabet Σ, we denote by a *dictionary* a finite subset $\mathcal{D}$ of Σ^+. By ε we denote the empty string, by $w^R = a_l \dots a_1$ we denote the reverse of a string $w = a_1 \dots a_l$.

A *word* is a string $w \in \mathcal{D}$, and a *sentence* over an alphabet Σ with respect to a dictionary $\mathcal{D}$ is a string $s \in \mathcal{D}^*$, i.e., a string that can be decomposed into dictionary words.

Using this notation, we can define the spoonerism problem as follows.

Definition 1. *The* spoonerism problem *is the following decision problem:*

Input: *A dictionary* $\mathcal{D} = \{w_1, w_2, \dots, w_l\}$ *and a sentence* $s = s_1 s_2 \dots s_n$ *of length* n *over an alphabet* Σ.

Output: YES *if there exist* $i, j \in \{1, \dots, n\}$ *such that* $s_i \neq s_j$ *and the string*

$$s' = s_1 \dots s_{i-1} s_j s_{i+1} \dots s_{j-1} s_i s_{j+1} \dots s_n$$

is a sentence over Σ, NO *otherwise.*

Informally speaking, we are asked to find out whether we can swap exactly two different symbols in a sentence s and get a new sentence s'.

Throughout the paper, we will use $a, b, c, \dots$ to denote single letters from Σ and we will use $u, v, w, \dots$ to denote (possibly empty) strings over Σ. Furthermore, let $n = |s|$ be the length of the input sentence s, let $k = \max_{u \in \mathcal{D}} |u|$ be the length of the longest word in the given dictionary and let $m = \sum_{u \in \mathcal{D}} |u|$ be the total size of the dictionary.

The running time and space complexities of all of our algorithms will depend on the four parameters $|\Sigma|$, m, n and k. It is obvious that $m \geq k$, furthermore, we will assume in the following that $n \geq k$ (otherwise we can just ignore longer dictionary words).

We present two algorithms for the spoonerism problem based on the idea of dynamic programming. The complexity of these algorithms (under the assumption of fixed $|\Sigma|$) is summarized in Table 1. These results improve the results claimed[4] by [2, 7], which use a graph-theoretic approach combined with a divide and conquer technique and achieve time complexity $O(n^2 k + nk^2 \log n)$ for processing the input sentence.

[4] The complete proofs of these results are not given in [2,7] and therefore we have not been able to check their correctness.

	Preprocessing the dictionary	Processing the sentence
Basic algorithm	$O(m)$	$O(nk^3)$
Improved algorithm	$O(mk)$	$O(nk^2)$

Table 1. Time complexity of presented algorithms.

3 The Basic Dynamic-Programming Algorithm

In this section, we will present an algorithm for solving the spoonerism problem that works in time $O(|\Sigma|m + nk\,(|\Sigma| + k)^2)$.

The main idea of the algorithm is to preprocess the input dictionary (in $O(|\Sigma|m)$ time) and to use dynamic programming to process the input sentence, processing each letter in $O(k\,(|\Sigma| + k)^2)$ time.

Definition 2. *We denote the set of all prefixes of all words from the dictionary $\mathcal{D}$ as* $\mathrm{Pref}\,(\mathcal{D})$. *Formally,* $\mathrm{Pref}\,(\mathcal{D}) = \{u \mid \exists v : uv \in \mathcal{D}\}$.

It is easy to see that $\varepsilon \in \mathrm{Pref}\,(\mathcal{D})$ and $\mathcal{D} \subseteq \mathrm{Pref}\,(\mathcal{D})$.

Definition 3. *Let $u, v \in \Sigma^*$. We say that v is a* live suffix *of u w.r.t. the dictionary $\mathcal{D}$ if and only if there exists a partition $u = u'v$ such that*

- *u' is a sentence w.r.t. $\mathcal{D}$, i. e., u' can be represented as a sequence of words from $\mathcal{D}$, and*
- *v is a prefix of some word from the dictionary $\mathcal{D}$, i. e., $v \in \mathrm{Pref}\,(\mathcal{D})$.*

We denote the set of all live suffixes of word u w.r.t. the dictionary $\mathcal{D}$ as $\mathcal{L}^S_{\mathcal{D}}(u)$. If the dictionary is clear from the context, we also write $\mathcal{L}^S(u)$.

Intuitively, the idea behind our algorithm can be described as follows: We want to process the input sentence s sequentially from left to right, and, for any prefix u of s, we want to keep track of all possible partitions of u into dictionary words (plus one prefix of a dictionary word at the end). Actually, we will not need to remember the complete partition, two partitions ending with the same live suffix can be treated as equivalent; thus, we only need to store information about the live suffixes. Here, we have to distinguish between three possible situations: The desired swap of two letters can occur completely inside u, completely outside u, or it can exchange a letter from u with a letter from the remainder of s. In the latter case, we also need to remember the letters exchanged. Formally, we can define these sets of live suffixes as follows.

Definition 4. *Let u be a string over some alphabet Σ, let $\mathcal{D}$ be some dictionary over Σ.*

- *$\mathcal{S}_0(u)$ is the set of all live suffixes of u, i.e. $\mathcal{S}_0(u) = \mathcal{L}^S(u)$.*
- *For all $a, b \in \Sigma$ such that $a \neq b$, $\mathcal{S}_1^{a \to b}(u)$ is the set of all live suffixes of all strings u' obtained by replacing a single letter a by letter b in the string u. More formally, $\mathcal{S}_1^{a \to b}(u) = \bigcup_{vav' = u} \mathcal{L}^S(vbv')$.*

– $\mathcal{S}_2(u)$ *is the set of all live suffixes of all strings* u' *obtained by swapping two letters* $a \neq b$ *in the string* u*. Formally,* $\mathcal{S}_2(u) = \bigcup_{vav'bv''=u \wedge a \neq b} \mathcal{L}^S(vbv'av'')$.

We will call the sets $\mathcal{S}_0(u)$*,* $\mathcal{S}_1^{a\to b}(u)$ *for all* $a \neq b$*, and* $\mathcal{S}_2(u)$ *the* $\mathcal{S}$-sets *for* u *or the* $\mathcal{S}(u)$-sets *for short.*

It is easy to see that there is a solution for the spoonerism problem on the input sentence s if and only if $\varepsilon \in \mathcal{S}_2(s)$.

The idea of our algorithm is to compute the sets $\mathcal{S}_0$, $\mathcal{S}_1^{a\to b}$, and $\mathcal{S}_2$ incrementally by using the following equations:

– To compute $\mathcal{S}_0(uc)$, it is sufficient to augment all possible elements of $\mathcal{S}_0(u)$ with letter c. Furthermore, if the obtained set contains a word from the dictionary, then we can add ε into $\mathcal{S}_0(uc)$. Formally,

$$\mathcal{S}_0(uc) = \begin{cases} \{vc \mid v \in \mathcal{S}_0(u), vc \in \text{Pref}\,(\mathcal{D})\} \cup \{\varepsilon\} & \text{iff } \exists v \in \mathcal{S}_0(u) : vc \in \mathcal{D} \\ \{vc \mid v \in \mathcal{S}_0(u), vc \in \text{Pref}\,(\mathcal{D})\} & \text{otherwise} \end{cases} \tag{1}$$

– For computing $\mathcal{S}_1^{a\to b}(uc)$, we distinguish two cases. Either the replacement of the letters occurs in u, or (in case $a = c$) the letter c is replaced. Each of these cases yields a subset of Pref $(\mathcal{D})$ and the resulting set $\mathcal{S}_1^{a\to b}(uc)$ is the union of these subsets. In the former case, the situation is analogous to the one described in previous paragraph. Let $X^{a\to b}(uc)$ be the subset of Pref $(\mathcal{D})$ obtained from the first case:

$$X^{a\to b}(uc) = \{vc \mid v \in \mathcal{S}_1^{a\to b}(u), vc \in \text{Pref}\,(\mathcal{D})\}$$

In the latter case (occurring only if $c = a$), we use our knowledge of $\mathcal{S}_0(u)$. Let $Y^{a\to b}(uc)$ be the subset of Pref $(\mathcal{D})$ obtained from the second case:

$$Y^{a\to b}(uc) = \begin{cases} \{vb \mid v \in \mathcal{S}_0(u), vb \in \text{Pref}\,(\mathcal{D})\} & \text{iff } a = c \\ \emptyset & \text{otherwise} \end{cases}$$

If the obtained set $X^{a\to b}(uc) \cup Y^{a\to b}(uc)$ contains a word from the dictionary, we have to add the empty string:

$$\mathcal{S}_1^{a\to b}(uc) = \begin{cases} X^{a\to b}(uc) \cup Y^{a\to b}(uc) \cup \{\varepsilon\} & \text{iff } \exists v \in X^{a\to b}(uc) \cup Y^{a\to b}(uc) : \\ & v \in \mathcal{D} \\ X^{a\to b}(uc) \cup Y^{a\to b}(uc) & \text{otherwise} \end{cases} \tag{2}$$

– Computing $\mathcal{S}_2(uc)$ is analogous to computing $\mathcal{S}_1^{a\to b}(uc)$. Again, we distinguish two cases. Either the swap occurs in u, or the last letter c is one of the swapped letters. For the first case, we have

$$X(uc) = \{vc \mid v \in \mathcal{S}_2(u), vc \in \text{Pref}\,(\mathcal{D})\}.$$

For the second case, we have

$$Y(uc) = \{va \mid a \in \Sigma, v \in \mathcal{S}_1^{a\to c}(u), va \in \text{Pref}\,(\mathcal{D})\}.$$

Algorithm 1 Basic dynamic programming for the spoonerism problem

Input: A dictionary $\mathcal{D}$ of total size m with maximum word length k over an alphabet Σ and a sentence $s = s_1 \dots s_n$ w.r.t. $\mathcal{D}$.

1. Construct a trie from $\mathcal{D}$.
2. **for** $i := 1$ **to** n **do**
   Compute the $\mathcal{S}$-sets for $s_1 \dots s_i$ according to Equations (1), (2), and (3), using the trie for the look-up operations in the dictionary.
3. **if** $\varepsilon \in \mathcal{S}_2(s_1 \dots s_n)$ **then**
   Output YES
   else
   Output NO

Output: YES if there exists a sentence s' that can be constructed from s by swapping exactly two different letters, NO otherwise.

Finally, we augment the set by ε if necessary:

$$\mathcal{S}_2(uc) = \begin{cases} X(uc) \cup Y(uc) \cup \{\varepsilon\} & \text{iff } \exists v \in X(uc) \cup Y(uc) : v \in \mathcal{D} \\ X(uc) \cup Y(uc) & \text{otherwise} \end{cases} \tag{3}$$

It is easy to see that these equations are correct. Hence, after computing $\mathcal{S}_2(s)$, the algorithm can decide if there exists a solution for the given input sentence s just by checking if $\varepsilon \in \mathcal{S}_2(s)$.

The resulting algorithm is summarized as Algorithm 1.

Theorem 1. *Algorithm 1 can be implemented to run in* $O(|\Sigma|m + nk(|\Sigma| + k)^2)$ *time and* $O(|\Sigma|m + k(|\Sigma| + k)^2)$ *space.*

Proof. For implementing the algorithm efficiently, we use a trie T representing all words from the dictionary $\mathcal{D}$.[5] Each vertex of this trie uniquely represents one element from Pref $(\mathcal{D})$, the root vertex represents ε and the parent of the vertex representing ua represents u. For each vertex, it is sufficient to remember pointers to its children and a flag whether it represents a word from the dictionary. The total size of T is $O(|\Sigma|m)$ and it can also be built in $O(|\Sigma|m)$ time. Moreover, once built, the trie can be reused for different runs of the main part of the algorithm on different input sentences.

The main part of the algorithm processes each letter from the input sentence and computes the corresponding $\mathcal{S}$-sets. Each of these sets (there are $|\Sigma|^2 - |\Sigma| + 2$ of them) can be represented as a list of vertices of the trie T. Suppose the algorithm has computed the $\mathcal{S}(u)$-sets for some prefix u of the input sentence. To enumerate all members of $\mathcal{S}(uc)$-sets for the prefix augmented by one letter c, it is sufficient to iterate through all elements of the $\mathcal{S}(u)$-sets and apply the rules described in Equations (1), (2), and (3). Using the trie representation, it is possible to process one element of the $\mathcal{S}(u)$-sets in constant time as follows:

[5] For a detailed description of the trie data structure see e.g. Section 6.3 in [6].

Since a string v is in the $\mathcal{S}(u)$-sets represented by a pointer to a vertex in the trie, also any string vd, for $d \in \Sigma$, can be looked up in the trie by traversing only one of its edges. This way, the time complexity required to process the letter c of the input word is linear in the total size of the $\mathcal{S}(u)$-sets.

However, there may be some duplicate elements in the newly created $\mathcal{S}(uc)$-sets, which have to be removed. This can be done in the following way. For each set (possibly containing duplicates), we iterate through its elements and mark them directly in the trie. When finding an already marked element, we remove it as a duplicate. After finishing this, we iterate through the elements once more and unmark them in the trie. Such duplication removal requires only linear running time with respect to the number of elements of the $\mathcal{S}(uc)$-sets, regardless of the size of the dictionary.

Since each element can be processed in constant time, the key part of the complexity analysis of Algorithm 1 is to find an upper bound on the size of the $\mathcal{S}$-sets. We will give such a bound in what follows.

All elements in $\mathcal{S}_0(u)$ are suffixes of the word u of length at most k (recall that k is the length of the longest word in the dictionary). Hence, $|\mathcal{S}_0(u)| \in O(k)$.

Similarly, each element of $|\mathcal{S}_2(u)|$ is a suffix of u of length at most k, possibly with two letters swapped. Since there are $O(k^2)$ possibilities for this swap, $|\mathcal{S}_2(u)| \in O(k^3)$.

Now we analyze $\sum_{a,b\in\Sigma\wedge a\neq b} |\mathcal{S}_1^{a\to b}(u)|$ by considering each word from the set $\bigcup_{a,b\in\Sigma\wedge a\neq b} \mathcal{S}_1^{a\to b}(u)$ and estimating in how many $\mathcal{S}_1$-sets it can be: There are at most k different words that are suffixes of u and each of them can be in at most $O(|\Sigma|^2)$ sets: A suffix of u of length $l \leq k$ can be included in at most $|\Sigma|\cdot(|\Sigma|-1)$ sets corresponding to the $|\Sigma|\cdot(|\Sigma|-1)$ possible letter replacements in one of the first $|u|-l$ positions of u. There are at most $|\Sigma|k^2$ other strings in $\bigcup_{a,b\in\Sigma\wedge a\neq b} \mathcal{S}_1^{a\to b}(u)$, since there are k possibilities for the length of the word, at most k possibilities for the location of replacement and $|\Sigma|$ possibilities for the replacement letter. Each of these strings can belong to only one $\mathcal{S}_1(u)$-set, since both the replaced and the replacing letter are uniquely determined by the string itself and the string u. Hence, $\sum_{a,b\in\Sigma\wedge a\neq b} |\mathcal{S}_1^{a\to b}(u)| \in O(|\Sigma|^2k + |\Sigma|k^2)$.

Putting this together, there can be at most $O(k + |\Sigma|^2k + |\Sigma|k^2 + k^3) = O\left(k(|\Sigma| + k)^2\right)$ elements in the $\mathcal{S}(u)$-sets, which proves the time complexity $O\left(nk\left(|\Sigma| + k\right)^2\right)$ of step 2 of Algorithm 1.

As for the memory complexity, observe that only the $\mathcal{S}$-sets for the latest prefix have to be stored, which requires $O(k\left(|\Sigma| + k\right)^2)$ space. Furthermore, the trie representing the dictionary has to be stored, thus the total memory complexity is $O(|\Sigma|m + k\left(|\Sigma| + k\right)^2)$. □

It is not difficult to show that the complexity analysis presented in this theorem is tight. To show that $|\mathcal{S}_2(u)| = \Theta(k^3)$, consider a word $u = a^ib^ic^i$ such that $i = k/3$. There are exactly i^2 different words w that can be obtained from b^ic^i by switching one letter b with a letter c. Hence there are exactly $i^3 = \Theta(k^3)$ different words a^jw such that $j \leq i$. All of these words (if contained in the dictionary) also belong to the set $\mathcal{S}_2(u)$, so, for an appropriate dictionary, the

size of $\mathcal{S}_2(u)$ is indeed $\Theta(k^3)$. Similar reasoning can be used for the $\mathcal{S}_1(u)$-sets, too.

4 An Improved Algorithm

In this section, we will present an improved algorithm with a faster running time for processing the input sentence at the expense of a slower preprocessing phase. This algorithm will consider two separate cases, depending on whether the swapped letters occur inside the same dictionary word or not.

It will turn out that the case of letters that occur in two different dictionary words after swapping can be handled with asymptotically the same preprocessing time as in the previous algorithm and $O(n(|\Sigma|^2k + |\Sigma|k^2))$ time for processing the input sentence.

But handling the case of swapping inside a dictionary word with an improved time complexity will require a more expensive preprocessing in $O(mk^2|\Sigma|)$ time.

4.1 Swapping Across Dictionary Word Boundaries

First we present an algorithm which can detect in $O(n(|\Sigma|^2k+|\Sigma|k^2))$ time, after a preprocessing in $O(|\Sigma|m)$ time, if there is a possibility to swap two letters in the input sentence that are in different dictionary words in some partition of the so constructed sentence. We can formally define this special case of the spoonerism problem as follows.

Definition 5. *The* separated spoonerism problem *is the following decision problem:*

Input: *A dictionary* $\mathcal{D} = \{w_1, w_2, \ldots, w_l\}$ *and a sentence* $s = s_1 s_2 \ldots s_n$ *over an alphabet* Σ*.*

Output: YES *if there exist* $i, j, l \in \{1, \ldots, n\}$ *such that* $i < l < j$, $s_i \neq s_j$, *and the strings* $x' = s_1 \ldots s_{i-1} s_j s_{i+1} \ldots s_l$ *and* $y' = s_{l+1} \ldots s_{j-1} s_i s_{j+1} \ldots s_n$ *are sentences over* Σ*,* NO *otherwise.*

To solve the separated spoonerism problem, we will use a similar idea as in the previous section, processing the input sentence not only in forward direction but also backwards.

The algorithm tries to find a solution for all possible pairs of swapped letters $a, b \in \Sigma$ separately. Such a solution exists if and only if it is possible to decompose the input sentence $s = xy$ into parts x and y such that the following holds:

C1 It is possible to replace some letter a with b in the part x such that the result can be decomposed into dictionary words.

C2 It is possible to replace some letter b with a in the part y such that the result can be decomposed into dictionary words.

It is obvious that condition *C1* can be easily checked using the $\mathcal{S}$-sets described in Algorithm 1: *C1* holds if and only if $\varepsilon \in \mathcal{S}_1^{a\to b}(x)$. To check condition *C2*, we need to process the word s backwards and compute sets analogous to $\mathcal{S}$, but with reversed roles of prefixes and suffixes.

Definition 6. *We denote the dictionary containing reverses of all words from the dictionary $\mathcal{D}$ as $\mathcal{D}^R$. Formally, $\mathcal{D}^R = \{u^R \mid u \in \mathcal{D}\}$.*

We denote the set of all suffixes of all words from the dictionary $\mathcal{D}$ as $\mathrm{Suff}(\mathcal{D})$. *Formally,* $\mathrm{Suff}(\mathcal{D}) = \{u \mid \exists v : vu \in \mathcal{D}\}$. *It is easy to see that* $\mathrm{Suff}(\mathcal{D}) = \left(\mathrm{Pref}\left(\mathcal{D}^R\right)\right)^R$ *and* $\varepsilon \in \mathrm{Suff}(\mathcal{D}) \supseteq \mathcal{D}$.

Let $u, v \in \Sigma^*$. We call v a live prefix *of u w.r.t. the dictionary $\mathcal{D}$, if and only if v^R is a live suffix of u^R w.r.t. the dictionary $\mathcal{D}^R$. We denote the set of all live prefixes of word u as $\mathcal{L}^P(u)$.*

By $\mathcal{P}_0(u)$ we denote the set of all live prefixes of u, i. e., $\mathcal{P}_0(u) = \mathcal{L}^P(u)$.

For all $a \neq b \in \Sigma$, $\mathcal{P}_1^{a\to b}(u)$ is the set of all live prefixes of all strings u' obtained by replacing a single letter a by letter b in the string u. Formally, $\mathcal{P}_1^{a\to b}(u) = \bigcup_{vav'=u \wedge a\neq b} \mathcal{L}^P(vbv')$.

Hence, for checking condition *C2*, it is sufficient to check if $\varepsilon \in \mathcal{P}_1^{b\to a}(y)$. To do so, the elements of the $\mathcal{P}$-sets can be computed similarly as the elements of the $\mathcal{S}$-sets, processing the input word backwards.

- To compute $\mathcal{P}_0(cu)$, it is sufficient to prepend all possible elements of $\mathcal{P}_0(u)$ with letter c. Furthermore, if the obtained set contains a word from the dictionary, then we can add ε into $\mathcal{P}_0(cu)$. Formally,

$$\mathcal{P}_0(cu) = \begin{cases} \{cv \mid v \in \mathcal{P}_0(u), cv \in \mathrm{Suff}(\mathcal{D})\} \cup \{\varepsilon\} & \text{iff } \exists v \in \mathcal{P}_0(u) : cv \in \mathcal{D} \\ \{cv \mid v \in \mathcal{P}_0(u), cv \in \mathrm{Suff}(\mathcal{D})\} & \text{otherwise} \end{cases} \tag{4}$$

- For computing $\mathcal{P}_1^{a\to b}(cu)$, we distinguish two cases. Either the replacement of the letters occurs in u, or (in case $a = c$) the letter c is replaced. Each of these cases yields a subset from $\mathrm{Suff}(\mathcal{D})$ and the resulting set $\mathcal{P}_1^{a\to b}(cu)$ is the union of these subsets. In the former case, the situation is analogous to the one described in the previous paragraph. Let $X^{a\to b}(cu)$ be the subset of $\mathrm{Suff}(\mathcal{D})$ obtained from the first case:

$$X^{a\to b}(cu) = \{cv \mid v \in \mathcal{P}_1^{a\to b}(u), cv \in \mathrm{Suff}(\mathcal{D})\}$$

  In the latter case (occurring only if $c = a$), we use our knowledge of $\mathcal{P}_0(u)$. Let $Y^{a\to b}(cu)$ be the subset of $\mathrm{Suff}(\mathcal{D})$ obtained from the second case:

$$Y^{a\to b}(cu) = \begin{cases} \{bv \mid v \in \mathcal{P}_0(u), bv \in \mathrm{Suff}(\mathcal{D})\} & \text{iff } c = a \\ \emptyset & \text{otherwise} \end{cases}$$

If the obtained set $X^{a\to b}(cu)\cup Y^{a\to b}(cu)$ contains a word from the dictionary, we have to add the empty string:

$$\mathcal{P}_1^{a\to b}(cu) = \begin{cases} X^{a\to b}(cu) \cup Y^{a\to b}(cu) \cup \{\varepsilon\} & \text{iff } \exists v \in X^{a\to b}(cu) \cup Y^{a\to b}(cu) : \\ & v \in \mathcal{D} \\ X^{a\to b}(cu) \cup Y^{a\to b}(cu) & \text{otherwise} \end{cases} \tag{5}$$

The resulting strategy is summarized in Algorithm 2.

Algorithm 2 Solving the separated spoonerism problem

Input: A dictionary $\mathcal{D}$ of total size m with maximum word length k over an alphabet Σ and a sentence $s = s_1 \dots s_n$ w.r.t. $\mathcal{D}$.

1. Construct a trie T from $\mathcal{D}$ and a trie T^R from $\mathcal{D}^R$.
2. **for** $i := 1$ **to** n **do**
   Compute the $\mathcal{S}_0$- and $\mathcal{S}_1$-sets for $s_1 \dots s_i$ according to Equations (1) and (2), using T for the look-up operations in the dictionary.
   Compute the $\mathcal{P}_0$- and $\mathcal{P}_1$-sets for $s_{n-i} \dots s_n$ according analogous equations, using T^R for the look-up operations in the dictionary.
3. **for** $a, b \in \Sigma$, $a \neq b$ **do**
   for $l := 1$ **to** $n-1$ **do**
   if $\varepsilon \in \mathcal{S}_1^{a\to b}(s_1 \dots s_l)$ **and** $\varepsilon \in \mathcal{P}_1^{b\to a}(s_{l+1} \dots s_n)$ **then**
   Output YES and stop.
   Output NO

Output: YES if there exists a solution to the separated spoonerism problem, NO otherwise.

Lemma 1. *Algorithm 2 solves the separated spoonerism problem in* $O(m|\Sigma| + n(|\Sigma|^2 k + |\Sigma|k^2))$ *time and* $O(m|\Sigma| + |\Sigma|k^2 + n|\Sigma|^2)$ *space.*

Proof. Obviously, Algorithm 2 correctly solves the separated spoonerism problem.

The preprocessing of the dictionary in step 1 can be done in $O(m|\Sigma|)$ time as already explained in the analysis of Algorithm 1.

Calculating the $\mathcal{S}_0$- and $\mathcal{S}_1$-sets in step 2 is possible in $O(n(|\Sigma|^2 k + |\Sigma|k^2))$ time, as already shown in the proof of Theorem 1. Calculating the $\mathcal{P}_0$- and $\mathcal{P}_1$-sets obviously takes the same time.

The test in each single iteration of step 3 can be implemented to take constant time; thus, step 3 needs $O(n|\Sigma|^2)$ time overall.

Summarizing, the total time required by Algorithm 2 is in $O(m|\Sigma|+n(|\Sigma|^2 k+ |\Sigma|k^2))$. To prove the claimed space complexity, we note that the tries need $O(m|\Sigma|)$ space, and the $\mathcal{S}(u)$- and $\mathcal{P}(u)$-sets need $O(|\Sigma|^2 k + |\Sigma|k^2)$ space for one prefix u. There is no reason in storing all these sets for every prefix; only

the information whether $\varepsilon \in \mathcal{S}_1^{a\to b}(u)$ and $\varepsilon \in \mathcal{P}_1^{a\to b}(u)$ needs to be stored for each prefix u and for each $a, b \in \Sigma$, hence requiring $O(n|\Sigma|^2)$ space. □

4.2 Swapping Inside a Dictionary Word

If the separated spoonerism problem has no solution, we proceed to the case where the swapped letters are located in the same dictionary word. The main idea of the algorithm is to try all possible decompositions of the input sentence $s = w_1w_2w_3$ into three parts w_1, w_2, and w_3 such that w_1 and w_3 can be decomposed into dictionary words and w_2 is a string such that a swap of two different letters makes it a dictionary word.

It is easy to see that w_1 and w_3 can be decomposed if and only if $\varepsilon \in \mathcal{S}_0(w_1)$ and $\varepsilon \in \mathcal{P}_0(w_3)$. Since the sets $\mathcal{S}_0$ and $\mathcal{P}_0$ have already been precomputed in Algorithm 2 for the separated spoonerism problem, each of these checks can be made in constant time.

Now we describe how to check whether two different letters in w_2 can be swapped as to yield a dictionary word.

Our algorithm constructs another trie T' which contains all strings that can be reached from a dictionary word by swapping two different letters. For a dictionary word w of length l, there are obviously at most $l^2 \leq k^2$ different reachable strings. Thus, for each dictionary word, the resulting trie T' contains at most k^2 strings of the same length. The total size of T' is thus in $O(k^2m|\Sigma|)$.

After we have constructed this additional trie, we can process an input sentence as follows: There are $O(nk)$ possible partitions $s = w_1w_2w_3$ as described above, the consistency check for w_1 and w_3 can be done in constant time for each of these partitions. By enumerating the possible partitions in a suitable way, also the look-up of w_2 in the additional trie T' can be done in amortized constant time.

This strategy is summarized in Algorithm 3.

Theorem 2. *Algorithm 3 solves the spoonerism problem in* $O(mk^2|\Sigma|+n(|\Sigma|^2k+|\Sigma|k^2))$ *time and* $O(mk^2|\Sigma| + |\Sigma|k^2 + n|\Sigma|^2)$ *space.*

Proof. From the discussion above it is clear that Algorithm 3 solves the spoonerism problem.

We will now analyze its time complexity. Step 1 is possible in $O(m|\Sigma| + n(|\Sigma|^2k+|\Sigma|k^2))$ time according to Lemma 1. The trie T' has a size of $O(k^2m|\Sigma|)$ as discussed above, and it can obviously also be constructed in $O(k^2m|\Sigma|)$ time. The tests in each iteration of step 3 can be performed in constant time: We have already discussed this for the tests on the sets $\mathcal{S}_0$ and $\mathcal{P}_0$ above; for testing the membership of $s_{i+1} \dots s_{i+l}$ in $\mathcal{D}'$ after having tested the membership of $s_{i+1} \dots s_{i+l-1}$ in the previous iteration, only one step along one edge of the trie is needed, hence this can also been done in constant time. This leads to an overall running time in $O(nk)$ for step 3. The total running time of the algorithm is thus in $O(m|\Sigma| + n(|\Sigma|^2k + |\Sigma|k^2) + k^2m|\Sigma| + nk) = O(k^2m|\Sigma| + n(|\Sigma|^2k + |\Sigma|k^2))$.

The space complexity of Algorithm 3 is obviously determined by the size of the additional trie T' and the space complexity of Algorithm 2. □

Algorithm 3 Solving the spoonerism problem with extensive preprocessing

Input: A dictionary $\mathcal{D}$ of total size m with maximum word length k over an alphabet Σ and a sentence $s = s_1 \dots s_n$ w.r.t. $\mathcal{D}$.

1. Use Algorithm 2 to check whether the separated spoonerism problem for $\mathcal{D}$ and s has a solution. If so, output YES and stop.
2. Construct a trie T' for the set $\mathcal{D}'$ of all strings that can be reached from a dictionary word by swapping two different letters.
3. **for** $i := 0$ **to** $n - 1$ **do**
   for $l := 1$ **to** $\min(k, n - i)$ **do**
   if $\varepsilon \in \mathcal{S}_0(s_1 \dots s_i)$ **and** $\varepsilon \in \mathcal{P}_0(s_{i+l+1} \dots s_n)$ **and** $s_{i+1} \dots s_{i+l} \in \mathcal{D}'$ **then**
   Output YES and stop.
   Output NO

Output: YES if there exists a solution to the spoonerism problem, NO otherwise.

5 A Further Improvement for Small Alphabets

In this section, we will describe another algorithm which can, at least in the case where $k \gg |\Sigma|$, save some preprocessing time at the expense of a slightly higher time complexity for processing an input sentence.

This algorithm in a first phase also uses Algorithm 2 to solve the separated spoonerism problem. In a second phase, it again considers all possible partitions $s = w_1 w_2 w_3$ of the input sentence s into two sentences w_1 and w_3 and a string w_2 which becomes a dictionary word by swapping two different letters.

For testing whether the swapping of letters may occur inside w_2, this algorithm will combine a dynamic programming approach similar to the one of Algorithm 1 with the idea of an expanded preprocessed trie from Algorithm 3. More precisely, in a preprocessing step, the algorithm will construct $O(|\Sigma|^2)$ new dictionaries, where the dictionary $\mathcal{D}^{a \to b}$ contains all strings that can be reached from a dictionary word from $\mathcal{D}$ by replacing a letter a by a letter b, i.e., $\mathcal{D}^{a \to b} = \{xby \mid xay \in \mathcal{D}\}$. Using dynamic programming, the algorithm further constructs the set $\mathcal{T}(w_2)$ of all strings from $\mathcal{D}^{a \to b}$ that can be obtained from w_2 by replacing an a by a b. In other words, the set $\mathcal{T}(w_2)$ contains all strings which can be obtained both from w_2 and from some dictionary word $w \in \mathcal{D}$ by replacing a letter a by a letter b.

If $\mathcal{T}^{a \to b}(w_2)$ is non-empty for some $a, b \in \Sigma$, it contains some string $u = xby = x'by'$, such that $w_2 = xay$ and $v = x'ay' \in \mathcal{D}$. As long as we can guarantee that $x \neq x'$, this string from $\mathcal{T}^{a \to b}(w_2)$ gives us a positive solution to the spoonerism problem. To ensure this, we have to store some information about the positions where the replacements may occur. This will be done both while constructing the tries representing the dictionaries $\mathcal{D}^{a \to b}$ and while computing the set $\mathcal{T}^{a \to b}(w_2)$. For every string in $\mathcal{T}^{a \to b}(w_2)$, the replacement position has to be unique, since we are starting with the unique string w_2.

For the strings in $\mathcal{D}^{a\to b}$, the situation is slightly more complicated. If there are two dictionary words $v = x'ay'$ and $z = x''ay''$, both mapped to the same string $u = x'by' = x''by'' \in \mathcal{D}^{a\to b}$ where $x' \neq x''$, i.e, via replacements in different positions, the presence of u in $\mathcal{T}^{a\to b}(w_2)$ already ensures a positive solution to the spoonerism problem. This means that we have to store, for each string in $\mathcal{D}^{a\to b}$, either the unique replacement position or just the information that the position is not unique.

The $\mathcal{T}^{a\to b}$-sets can be computed in a similar way as the $\mathcal{S}^{a\to b}$-sets in Algorithms 1 and 2. Creating the modified dictionaries $\mathcal{D}^{a\to b}$ in the preprocessing phase can be implemented in $O(mk|\Sigma|^2)$ time and space. The processing of the input sentence itself takes $O(nk|\Sigma|^2)$ time – considering $O(|\Sigma|^2)$ different letter pairs, $O(nk)$ partitions $s = w_1w_2w_3$ and $O(k)$ elements of the $\mathcal{T}^{a\to b}$-sets and processing each element in constant time. The space complexity (not counting the preprocessed dictionaries and information reused from Algorithm 2) is $O(k)$, required for storing the $\mathcal{T}^{a\to b}$-sets. Adding the complexity of Algorithm 2, which is used for deciding whether the swapping of letters may occur across dictionary word boundaries, a time complexity in $O(mk|\Sigma|^2 + nk^2|\Sigma|^2)$ and a space complexity in $O(mk|\Sigma|^2 + |\Sigma|k^2 + n|\Sigma|^2)$ is obtained.

Now we present the description and analysis of the algorithm in more detail. At first we provide the formal definition of the set $\mathcal{T}^{a\to b}(u)$ and describe how to compute it for any string $u \in \Sigma^*$. The idea is analogous to that of computing the set $\mathcal{S}_1^{a\to b}(u)$ used in Algorithms 1 and 2.

Definition 7. *For all $a, b \in \Sigma$, $a \neq b$, the set $\mathcal{T}^{a\to b}(u)$ is the set of all pairs (w, l), where w is a string obtained by replacing a single letter a by a letter b in the string u that also is a prefix of some word from the dictionary $\mathcal{D}^{a\to b}$, and where $l \in \{1, \ldots, |u|\}$ denotes the position of the letter replacement. Formally,*

$$\mathcal{T}^{a\to b}(u) = \left\{(xby, |x| + 1) \mid u = xay \wedge xby \in \mathrm{Pref}\left(\mathcal{D}^{a\to b}\right)\right\}.$$

Slightly abusing notation, in the following we will also say that $w \in \mathcal{T}^{a\to b}(u)$, if there exists an l such that $(w, l) \in \mathcal{T}^{a\to b}(u)$.

The sets $\mathcal{T}^{a\to b}$ can be computed similarly as the sets $\mathcal{S}_1^{a\to b}$, except that the empty string ε is not added to the sets and the dictionary $\mathcal{D}^{a\to b}$ is used instead of $\mathcal{D}$. For computing $\mathcal{T}^{a\to b}(uc)$, we distinguish two cases. Either the replacement of the letters occurs in u, or (in case $a = c$) the letter c is replaced. Let $X^{a\to b}(uc)$ be the set of strings corresponding to the former case:

$$X^{a\to b}(uc) = \left\{(vc, l) \mid (v, l) \in \mathcal{T}^{a\to b}(u), vc \in \mathrm{Pref}\left(\mathcal{D}^{a\to b}\right)\right\}$$

In the latter case (that occurs only if $c = a$), we obtain at most one string:

$$Y^{a\to b}(uc) = \begin{cases} \{(ub, |ub|)\} & \text{iff } c = a \\ \emptyset & \text{otherwise} \end{cases}$$

Putting this together yields

$$\mathcal{T}^{a\to b}(uc) = X^{a\to b}(uc) \cup Y^{a\to b}(uc). \tag{6}$$

The complete strategy of this approach is summarized in Algorithm 4.

Algorithm 4 Refined dynamic programming for the spoonerism problem

Input: A dictionary $\mathcal{D}$ of total size m with maximum word length k over an alphabet Σ and a sentence $s = s_1 \dots s_n$ w.r.t. $\mathcal{D}$.

1. Use Algorithm 2 to check whether the separated spoonerism problem for $\mathcal{D}$ and s has a solution. If so, output YES and stop.
2. For all pairs (a, b) of different letters from Σ, construct a trie for the dictionary $\mathcal{D}^{a\to b}$ where the vertices of the trie are labeled with either the unique position of letter replacement that led to the corresponding string or with MULT if this position is not unique.
3. **for** $a, b \in \Sigma$, $a \neq b$ **do**
   for $i := 0$ **to** $n - 1$ **do**
   for $l := 1$ **to** $\min(k, n - i)$ **do**
   if $\varepsilon \in \mathcal{S}_0(s_1 \dots s_i)$ **and** $\varepsilon \in \mathcal{P}_0(s_{i+l+1} \dots s_n)$ **then**
   Construct the set $\mathcal{T}^{a\to b}(s_{i+1} \dots s_{i+l})$ from the set $\mathcal{T}^{a\to b}(s_{i+1} \dots s_{i+l-1})$ using Equation (6)
   for all $(w, l) \in \mathcal{T}^{a\to b}(s_{i+1} \dots s_{i+l})$ **do**
   Check if the vertex in $\mathcal{D}^{a\to b}$ corresponding to w is labeled with some $l' \neq l$ or with MULT. If so, output YES and stop.
   Output NO

Output: YES if there exists a solution to the spoonerism problem, NO otherwise.

Theorem 3. *Algorithm 4 solves the spoonerism problem in* $O(mk|\Sigma|^2 + nk^2|\Sigma|^2)$ *time and* $O(|\Sigma|^2 km + |\Sigma|k^2 + n|\Sigma|^2)$ *space.*

Proof. It is clear from our discussion above that the algorithm correctly solves the spoonerism problem.

Step 1 again takes $O(m|\Sigma| + n(|\Sigma|^2 k + |\Sigma|k^2))$ time according to Lemma 1. We now analyze the time complexity needed to create and preprocess the modified dictionaries in step 2. Each word $u \in \mathcal{D}$ yields $|\Sigma||u|$ different strings belonging to various of the modified dictionaries. Each of these strings can be inserted into the appropriate dictionary represented by a trie in $O(|\Sigma||u|)$ time. Labeling any vertex of any of the tries requires only constant additional time and space, hence also the overall time and space complexity of step 2 is

$$O\left(\sum_{u\in\mathcal{D}} |\Sigma|^2|u|^2\right) = O\left(|\Sigma|^2 k \sum_{u\in\mathcal{D}} |u|\right) = O\left(|\Sigma|^2 km\right).$$

The inner loop in step 3 is performed $O(nk|\Sigma|^2)$ times. Since obviously $|\mathcal{T}^{a\to b}(u)| \leq k$ for each u, the construction of the $\mathcal{T}$-sets according to Equation (6) can be performed in $O(k)$ time using the trie for the modified dictionary $\mathcal{D}^{a\to b}$. Looking up the middle part $s_{i+1} \dots s_{i+l}$ in the corresponding trie can be done in amortized constant time, with a proof analogous to the one of Theorem 2. Thus, step 3 has a total time complexity in $O(nk^2|\Sigma|^2)$.

Overall, Algorithm 4 has a time complexity in $O(m|\Sigma| + n(|\Sigma|^2 k + |\Sigma|k^2) + mk|\Sigma|^2 + nk^2|\Sigma|^2) = O(mk|\Sigma|^2 + nk^2|\Sigma|^2)$.

Considering the space complexity, the algorithm needs to store the tries for the modified dictionaries, requiring $O(mk|\Sigma|^2)$ space. Moreover, the space requirements of step 1 exceed those for step 3 (not counting the tries). Thus, the overall space complexity is in $O(mk|\Sigma|^2 + |\Sigma|^2 k + |\Sigma|k^2 + n|\Sigma|^2) = O(mk|\Sigma|^2 + |\Sigma|k^2 + n|\Sigma|^2)$. □

6 Conclusion

We have presented some efficient algorithms for the spoonerism problem. The worst-case running time of the basic dynamic-programming algorithm (Algorithm 1) is $O(m|\Sigma|)$ for preprocessing and $O(nk(|\Sigma| + k)^2)$ for processing the input. The improved algorithm (Algorithm 3) reduces the input processing time to $O(nk(|\Sigma|^2 + |\Sigma|k))$, which is asymptotically better even for the case $k = \Theta(n)$. Finally, we have presented a possible improvement of the preprocessing time of Algorithm 3 to $O(mk|\Sigma|^2)$ at the expense of a slightly worse input processing time $O(nk^2|\Sigma|^2)$.

For a variant of the spoonerism problem, where substrings of length greater than 1 may be swapped, these algorithms obviously yield a running time that is exponential in the length of the interchanged substrings. We leave it as an open problem to find more efficient algorithms for this problem.

Acknowledgment

We would like to thank Michael Bender who pointed us to this interesting problem and gave us comments and suggestions helpful for improving the presentation of our paper.

References

1. F. A. Allyn, J. S. Burt: Pinch my wig or winch my pig: Spelling, spoonerisms and other language skills, *Reading and Writing*, 10:51–74 (1998).
2. M. Bender, R. Clifford, K. Steinföfel, K. Tsichlas: The Spoonerism Problem, *Unpublished manuscript*.
3. Sir William Hayter: *Spooner: A Biography*, W. H. Allen, ISBN 0-491-01658-1 (1976).
4. M. A. Jaro: Advances in record linking methodology as applied to the 1985 census of Tampa Florida, *Journal of the American Statistical Society*, 64:1183–1210 (1989).
5. M. A. Jaro: Probabilistic linkage of large public health data file, *Statistics in Medicine*, 14:491–498 (1995).
6. D. E. Knuth: *The Art of Computer Programming (Volume 3) – Sorting and Searching*, Addison-Wesley, 1973.
7. K. Tsichlas, M. Bender, and ADG in King's College: The Spoonerism Problem, *Unpublished presentation at London Stringology Day 2005*.
8. W. E. Winkler: The state of record linkage and current research problems, *Statistics of Income Division*, Internal Revenue Service Publication R99/04 (1999).