Closed Factorization*

Golnaz Badkobeh[1], Hideo Bannai[2], Keisuke Goto[2], Tomohiro I[2],
Costas S. Iliopoulos[3], Shunsuke Inenaga[2], Simon J. Puglisi[4], and Shiho Sugimoto[2]

[1] Department of Computer Science,
University of Sheffield
United Kingdom
`g.badkobeh@sheffield.ac.uk`

[2] Department of Informatics,
Kyushu University
Japan
`{bannai,keisuke.gotou,tomohiro.i,inenaga,shiho.sugimoto}@inf.kyushu-u.ac.jp`

[3] Department of Informatics,
King's College London
United Kingdom
`c.iliopoulos@kcl.ac.uk`

[4] Department of Computer Science,
University of Helsinki
Finland
`puglisi@cs.helsinki.fi`

Abstract. A *closed string* is a string with a proper substring that occurs in the string as a prefix *and* a suffix, but not elsewhere. Closed strings were introduced by Fici (Proc. WORDS, 2011) as objects of combinatorial interest in the study of Trapezoidal and Sturmian words. In this paper we consider algorithms for computing closed factors (substrings) in strings, and in particular for greedily factorizing a string into a sequence of longest closed factors. We describe an algorithm for this problem that uses linear time and space. We then consider the related problem of computing, for every position in the string, the longest closed factor starting at that position. We describe a simple algorithm for the problem that runs in $O(n \log n / \log \log n)$ time.

1 Introduction

A *closed string* is a string with a proper substring that occurs as a prefix and a suffix but does not have internal occurrences. Closed strings were introduced by Fici [3] as objects of combinatorial interest in the study of Trapezoidal and Sturmian words. Since then, Badkobeh, Fici, and Liptak [1] have proved a tight lowerbound for the number of closed factors (substrings) in strings of given length and alphabet.

In this paper we initiate the study of algorithms for computing closed factors. In particular we consider two algorithmic problems. The first, which we call the *closed factorization problem*, is to greedily factorize a given string into a sequence of longest closed factors (we give a formal definition of the problem below, in Section 2). We describe an algorithm for this problem that uses $O(n)$ time and space, where n is the length of the given string.

The second problem we consider is the *closed factor array problem*, which requires us to compute the length of the longest closed factor starting at each position in the

* This research is partially supported by the Academy of Finland through grants 258308 and 250345 (CoECGR), and by the Japan Society for the Promotion of Science.

input string. We show that this problem can be solved in $O(n\frac{\log n}{\log\log n})$ time, using techniques from computational geometry.

This paper proceeds as follows. In the next section we set notation, define the problems more formally, and outline basic data structures and concepts. Section 3 describes an efficient solution to the closed factorization problem and Section 4 then considers the closed factor array. Reflections and outlook are offered in Section 5.

2 Preliminaries

2.1 Strings and Closed Factorization

Let Σ denote a fixed integer alphabet. An element of Σ^* is called a string. For any strings $\mathsf{W}, \mathsf{X}, \mathsf{Y}, \mathsf{Z}$ such that $\mathsf{W} = \mathsf{XYZ}$, the strings $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are respectively called a prefix, substring, and suffix of W. The length of a string X will be denoted by $|\mathsf{X}|$. Let ε denote the empty string of length 0, i.e., $|\varepsilon| = 0$. For any non-negative integer n, $\mathsf{X}[1,n]$ denotes a string X of length n. A prefix X of a string W with $|\mathsf{X}| < |\mathsf{W}|$ is called a proper prefix of W. Similarly, a suffix X of W with $|\mathsf{Z}| < |\mathsf{W}|$ is called a proper suffix of W. For any string X and integer $1 \le i \le |\mathsf{X}|$, let $\mathsf{X}[i]$ denote the ith character of X, and for any integers $1 \le i \le j \le |\mathsf{X}|$, let $\mathsf{X}[i..j]$ denote the substring of X that starts at position i and ends at position j. For convenience, let $\mathsf{X}[i..j]$ be the empty string if $j < i$. For any strings X and Y, if $\mathsf{Y} = \mathsf{X}[i..j]$, then we say that i is an occurrence of Y in X.

If a non-empty string X is both a proper prefix and suffix of string W, then, X is called a *border* of W. A string W is said to be *closed*, if there exists a border X of W that occurs exactly twice in W, i.e., $\mathsf{X} = \mathsf{W}[1..|\mathsf{X}|] = \mathsf{W}[|\mathsf{W}| - |\mathsf{X}| + 1..|\mathsf{W}|]$ and $\mathsf{X} \neq \mathsf{W}[i..i + |\mathsf{X}| - 1]$ for any $2 \le i \le |\mathsf{W}| - |\mathsf{X}|$. We suppose that any single character $\mathsf{C} \in \Sigma$ is closed, assuming that the empty string ε occurs exactly twice in C. A string X is a *closed factor* of W, if X is closed and is a substring of W. Throughout we consider a string $\mathsf{X}[1,n]$ on Σ. We define the *closed factorization* of string $\mathsf{X}[1,n]$ as follows.

Definition 1 (Closed Factorization). *The closed factorization of string* $\mathsf{X}[1,n]$, *denoted* $\mathsf{CF}(\mathsf{X})$, *is a sequence* $(\mathsf{G}_0, \mathsf{G}_1, \ldots, \mathsf{G}_k)$ *of strings such that* $\mathsf{G}_0 = \varepsilon$, $\mathsf{X}[1,n] = \mathsf{G}_1 \cdots \mathsf{G}_k$ *and, for each* $1 \le j \le k$, G_j *is the longest prefix of* $\mathsf{X}[|\mathsf{G}_1 \cdots \mathsf{G}_{j-1}| + 1..n]$ *that is closed.*

Example 2. For string $\mathsf{X} = \mathtt{ababaacbbbcbcc\$}$, $\mathsf{CF}(\mathsf{X}) = (\varepsilon, \mathtt{ababa}, \mathtt{a}, \mathtt{cbbbcb}, \mathtt{cc}, \$)$.

We remark that a closed factor G_j is a single character if and only if $|\mathsf{G}_1 \cdots \mathsf{G}_{j-1}|+1$ is the rightmost (last) occurrence of character $\mathsf{X}[|\mathsf{G}_1 \cdots \mathsf{G}_{j-1}| + 1]$ in X.

We also define the *longest closed factor array* of string $\mathsf{X}[1,n]$.

Definition 3 (Longest Closed Factor Array). *The longest closed factor array of* $\mathsf{X}[1,n]$ *is an array* $\mathsf{A}[1,n]$ *of integers such that for any* $1 \le i \le n$, $\mathsf{A}[i] = \ell$ *if and only if* ℓ *is the length of the longest prefix of* $\mathsf{X}[i..n]$ *that is closed.*

Example 4. For string $\mathsf{X} = \mathtt{ababaacbbbcbcc\$}$, $\mathsf{A} = [5,4,3,5,2,1,6,3,2,4,3,1,2,1,1]$.

Clearly, given the longest closed factor array $\mathsf{A}[1,n]$ of string X, $\mathsf{CF}(\mathsf{X})$ can be computed in $O(n)$ time. However, the algorithm we describe in Section 4 to compute $\mathsf{A}[1,n]$ requires $O(n\frac{\log n}{\log\log n})$ time, and so using it to compute $\mathsf{CF}(\mathsf{X})$ would also take $O(n\frac{\log n}{\log\log n})$ time overall. In Section 3 we present an optimal $O(n)$-time algorithm to compute $\mathsf{CF}(\mathsf{X})$ that does not require $\mathsf{A}[1,n]$.

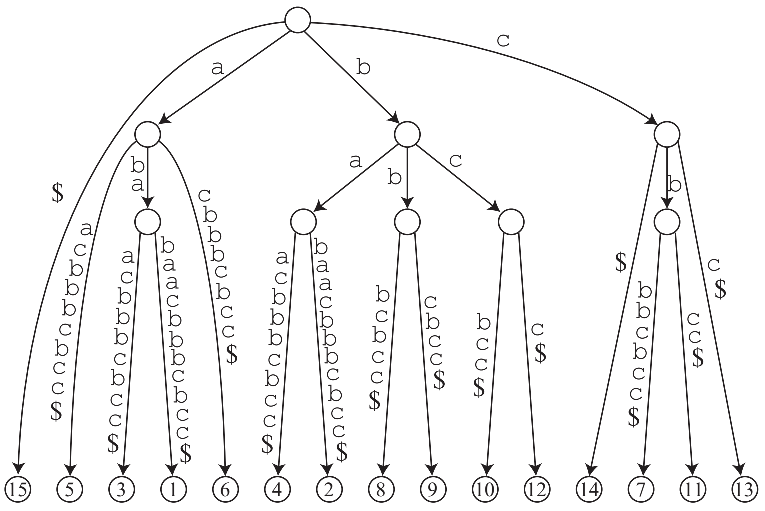

Figure 1. The suffix tree of string X = `ababaacbbbcbcc$`, where each leaf stores the beginning position of the corresponding suffix. All the branches from an internal node are sorted in ascending lexicographical order, assuming \$ is the lexicographically smallest. The suffix array SA of X is $[15, 5, 3, 1, 6, 4, 2, 8, 9, 10, 12, 14, 7, 11, 13]$, which corresponds to the sequence of leaves from left to right, and thus can be computed in linear time by a depth first traversal on the suffix tree.

2.2 Tools

The suffix array [5] SA_X (we drop subscripts when they are clear from the context) of a string X is an array $\mathsf{SA}[1..n]$ which contains a permutation of the integers $[1..n]$ such that $\mathsf{X}[\mathsf{SA}[1]..n] < \mathsf{X}[\mathsf{SA}[2]..n] < \cdots < \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n]$ is the j^{th} suffix of X in ascending lexicographical order.

The suffix tree [8] of a string $\mathsf{X}[1, n]$ is a compacted trie consisting of all suffixes of X. Suffix trees can be represented in linear space, and can be constructed in linear time for integer alphabets [2]. Figure 1 illustrates the suffix tree for an example string.

For a node w in the suffix tree let $\mathsf{pathlabel}(w)$ be the string spelt out by the letters on the edges on the path from the root to w. If there is a branch from node u to node v and u is an ancestor of v then we say $u = \mathsf{parent}(v)$. Assuming that every string X terminates with a special character \$ which occurs nowhere else in X, there is a one-to-one correspondence between the suffixes of X and the leaves of the suffix tree of X. We assume the branches from a node u to each child v of u are stored in ascending lexicographical order of $\mathsf{pathlabel}(v)$. When this is the case, SA is simply the leaves of the suffix tree when read during a depth-first traversal. At each internal node v in the suffix tree we store two additional values $v.s$ and $v.e$ such that $\mathsf{SA}[v.s..v.e]$ contains the beginning positions of all the suffixes in the subtree rooted at v.

3 Greedy Longest Closed Factorization in Linear Time

In this section, we present how to compute the closed factorization $\mathsf{CF}(\mathsf{X})$ of a given string $\mathsf{X}[1, n]$. Our high level strategy is to build a data structure that helps us to efficiently compute, for a given position i in X, the longest closed factor starting at i. The core of this data structure is the suffix tree for X, which we decorate in various ways.

Let S be the set of the beginning positions of the longest closed factors in $\mathsf{CF}(\mathsf{X})$. For any $i \in S$, let $\mathsf{G} = \mathsf{X}[i..i + |\mathsf{G}| - 1]$ be the longest closed factor of X starting at position i in X.

Let G' be the unique border of the longest closed factor G starting at position i of X, and b_i be its length, i.e., $\mathsf{G}' = \mathsf{G}[1..b_i] = \mathsf{X}[i..i + b_i - 1]$ (if G is a single character, then $\mathsf{G}' = \varepsilon$ and $b_i = 0$). The following lemma shows that we can efficiently compute $\mathsf{CF}(\mathsf{X})$ if we know b_i for all $i \in S$.

Lemma 5. *Given b_i for all $i \in S$, we can compute $\mathsf{CF}(\mathsf{X})$ in a total of $\mathrm{O}(n)$ time and space independently of the alphabet size.*

Proof. If $b_i = 0$, then $\mathsf{G} = \mathsf{X}[i]$. Hence, in this case it clearly takes $\mathrm{O}(1)$ time and space to compute G.

If $b_i > 1$, then we can compute G in $\mathrm{O}(|\mathsf{G}|)$ time and $\mathrm{O}(b_i)$ space, as follows. We preprocess the border G' of G using the Knuth-Morris-Pratt (KMP) string matching algorithm [4]. This preprocessing takes $\mathrm{O}(b_i)$ time and space. We then search for the first occurrence of G' in $\mathsf{X}[i + 1..n]$ (i.e. the next occurrence of the longest border of $\mathsf{G}[1, m]$ to the right of the occurrence $\mathsf{X}[i..i + b_i - 1]$). The location of the next occurrence tells us where the end of the closed factor is, and so it also tells us $\mathsf{G} = \mathsf{X}[i..i + |\mathsf{G}| - 1]$. The search takes $\mathrm{O}(|\mathsf{G}|)$ time — i.e. time proportional to the length of the closed factor. Because the sum of the lengths of the closed factors is n, over the whole factorization we take $\mathsf{O}(n)$ time and space. The running time and space usage of the algorithm are clearly independent of the alphabet size. $\square$

What remains is to be able to efficiently compute b_i for a given $i \in S$. The following lemma gives an efficient solution to this subproblem:

Lemma 6. *We can preprocess the suffix tree of string $\mathsf{X}[1, n]$ in $\mathrm{O}(n)$ time and space, so that b_i for each $i \in S$ can be computed in $\mathrm{O}(1)$ time.*

Proof. In each leaf of the suffix tree, we store the beginning position of the suffix corresponding to the leaf. For any internal node v of the suffix tree of X, let $\max(v)$ denote the maximum leaf value in the subtree rooted at v, i.e.,

$$\max(v) = \max\{i \mid \mathsf{X}[i..i + \mathsf{pathlabel}(v) - 1], 1 \leq i \leq n - \mathsf{pathlabel}(v) - 1\}.$$

We can compute $\max(v)$ for every v in a total of $\mathrm{O}(n)$ time total via a depth first traversal. Next, let $\mathsf{P}[1, n]$ be an array of pointers to suffix tree nodes (to be computed next). Initially every $\mathsf{P}[i]$ is set to null. We traverse the suffix tree in pre-order, and for each node v we encounter we set $\mathsf{P}[\max(v)] = v$ if $\mathsf{P}[\max(v)]$ is null. At the end of the traversal $\mathsf{P}[i]$ will contain a pointer to the highest node w in the tree for which i is the maximum leaf value (i.e., i is the rightmost occurrence of $\mathsf{pathlabel}(w)$).

We are now able to compute b_i, the length of the unique border of the longest closed factor starting at any given i, as follows. First we retrieve node $v = \mathsf{P}[i]$. Observe that, because of the definition of $\mathsf{P}[i]$, there are no occurrences of substring $\mathsf{X}[i..i + |\mathsf{pathlabel}(v)|]$ to the right of i. Let $u = \mathsf{parent}(v)$. There are two cases to consider:

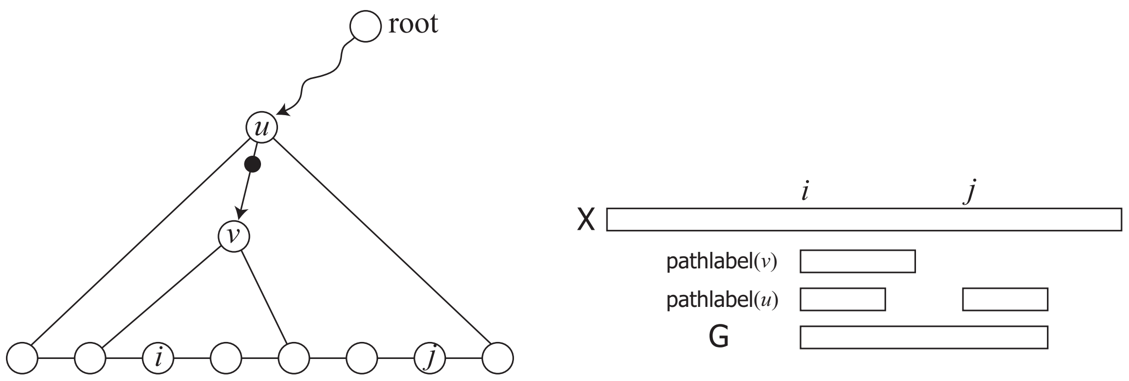

Figure 2. Illustration for Lemma 6. We consider the longest closed factor G starting at position i of string X. We retrieve node $\mathsf{P}[i] = v$, which implies $\max(v) = i$. Let u be the parent of v. The black circle represents a (possibly implicit) node which represents $\mathsf{X}[i..i + \mathsf{pathlabel}(u)]$, which has the same set of occurrences as $\mathsf{pathlabel}(v)$. Hence $b_i = |\mathsf{pathlabel}(u)|$, and therefore $\mathsf{G} = \mathsf{X}[i..j + \mathsf{pathlabel}(u) - 1]$, where j is the leftmost occurrence of $\mathsf{pathlabel}(u)$ with $j > i$.

- If u is not the root, then observe that there always exists an occurrence of substring $\mathsf{pathlabel}(u)$ to the right of position i (otherwise i would be the rightmost occurrence of $\mathsf{pathlabel}(u)$, but this cannot be the case since u is higher than v, and we defined $\mathsf{P}[i]$ to be the highest node w with $\max(w) = i$). Let j be the the leftmost occurrence of $\mathsf{pathlabel}(u)$ to the right of i. Then, the longest closed factor starting at position i is $\mathsf{X}[i..j + |\mathsf{pathlabel}(u)| - 1]$ (this position j is found by the KMP algorithm as in Lemma 5).
- If u is the root, then it turns out that i is the rightmost occurrence of character $\mathsf{X}[i]$ in X. Hence, the longest closed factor starting at position i is $\mathsf{X}[i]$.

The thing we have not shown is that $|\mathsf{pathlabel}(u)| = b_i$. This is indeed the case, since the set of occurrences of $\mathsf{X}[i..j + |\mathsf{pathlabel}(u)|]$ (i.e., leaves in the subtree corresponding to the string) is equivalent to that of $\mathsf{pathlabel}(v)$, any substring starting at i that is longer than $|\mathsf{pathlabel}(u)|$ does not occur to the right of i and thus b_i cannot be any longer. Hence $|\mathsf{pathlabel}(u)| = b_i$. (See also Figure 2).

Clearly $v = \mathsf{P}[i]$ can be retrieved in $\mathrm{O}(1)$ time for a given i, and then $u = \mathsf{parent}(v)$ can be obtained in $\mathrm{O}(1)$ time from v. This completes the proof. □

The main result of this section follows:

Theorem 7. *Given a string* $\mathsf{X}[1, n]$ *over an integer alphabet, the* closed factorization $\mathsf{CF}(\mathsf{X}) = (\mathsf{G}_1, \ldots, \mathsf{G}_k)$ *of* X *can be computed in* $\mathrm{O}(n)$ *time and space.*

Proof. $\mathsf{G}_0 = \varepsilon$ by definition and so does not need to be computed. We compute the other G_j in ascending order of $j = 1, \ldots, k$. Let s_i be the beginning position of G_i in X, i.e., $s_1 = 1$ and $s_i = |\mathsf{G}_1 \cdots \mathsf{G}_{i-1}| + 1$ for $1 < i \leq k$. We compute G_1 in $\mathrm{O}(|\mathsf{G}_1|)$ time and space from b_{s_1} using Lemma 5 and Lemma 6. Assume we have computed the first $j - 1$ factors $\mathsf{G}_1, \ldots, \mathsf{G}_{j-1}$ for any $1 \leq j < k - 1$. We then compute G_j in $\mathrm{O}(|\mathsf{G}_j|)$ time and space from b_{s_j}, again using Lemmas 5 and 6. Since $\sum_{j=1}^{k} |\mathsf{G}_j| = n$, the proof completes. □

The following is an example of how the algorithm presented in this section computes $\mathsf{CF}(\mathsf{X})$ for a given string X.

Example 8. Consider the running example string $\mathsf{X} = \mathtt{ababaacbbbcbcc\$}$, and see Figure 1, which shows the suffix tree of X.

1. We begin with node $\mathsf{P}[1]$ representing $\mathtt{ababaacbbbcbcc\$}$, whose parent represents $\mathtt{aba}$. Hence we get $b_1 = |\mathtt{aba}| = 3$. We run the KMP algorithm with pattern $\mathtt{aba}$ and find the first factor $\mathsf{G}_1 = \mathtt{ababa}$.
2. We then check node $\mathsf{P}[6]$ representing $\mathtt{a}$. Since its parent is the root, we get $b_2 = 0$ and therefore the second factor is $\mathsf{G}_2 = \mathtt{a}$.
3. We then check node $\mathsf{P}[7]$ representing $\mathtt{cbbbcbcc\$}$, whose parent represents $\mathtt{cb}$. Hence we get $b_3 = |\mathtt{cb}| = 2$. We run the KMP algorithm with pattern $\mathtt{cb}$ and find the third factor $\mathsf{G}_3 = \mathtt{cbbbcb}$.
4. We then check node $\mathsf{P}[13]$ representing $\mathtt{cc\$}$, whose parent represents $\mathtt{c}$. Hence we get $b_4 = |\mathtt{c}| = 1$. We run the KMP algorithm with pattern $\mathtt{c}$ and find the fourth factor $\mathsf{G}_4 = \mathtt{cc}$.
5. We finally check node $\mathsf{P}[15]$ representing \$. Since its parent is the root, we get $b_5 = 0$ and therefore the fifth factor is $\mathsf{G}_5 = \$$.

Consequently, we obtain $\mathsf{CF}(\mathsf{X}) = (\mathtt{ababa}, \mathtt{a}, \mathtt{cbbbcb}, \mathtt{cc}, \$)$, which coincides with Example 2.

4 Longest Closed Factor Array

A natural extension of the problem in the previous section is to compute the longest closed factor starting at *every* position in X in linear time — not just those involved in the factorization. Formally, we would like to compute the longest closed factor array of X, i.e., an array $\mathsf{A}[1,n]$ of integers such that $\mathsf{A}[i] = \ell$ if and only if ℓ is the length of the longest closed factor starting at position i in X.

Our algorithm for closed factorization computes the longest closed factor starting at a given position in time proportional to the factor's length, and so does not immediately provide a linear time algorithm for computing A; indeed, applying the algorithm naïvely at each position would take $\mathrm{O}(n^2)$ time to compute A. In what follows, we present a more efficient solution:

Theorem 9. *Given a string* $\mathsf{X}[1,n]$ *over an integer alphabet, the* closed factor array *of* X *can be computed in* $\mathrm{O}(n\frac{\log n}{\log\log n})$ *time and* $\mathrm{O}(n)$ *space.*

Proof. We extend the data structure of the last section to allow A to be computed in $\mathrm{O}(n\frac{\log n}{\log\log n})$ time and $\mathrm{O}(n)$ space. The main change is to replace the KMP algorithm scanning in the first algorithm with a data structure that allows us to find the end of the closed factor in time independent of its length.

We first preprocess the suffix array SA for *range successor* queries, building the data structure of Yu, Hon and Wang [9]. A range successor query $\mathsf{rsq}_{\mathsf{SA}}(s,e,k)$ returns, given a range $[s,e] \subseteq [1,n]$, the smallest value $x \in \mathsf{SA}[s..e]$ such that $x > k$, or null if there is no value larger than k in $\mathsf{SA}[s..e]$. Yu et al.'s data structure allows range successor queries to be answered in $\mathrm{O}(\frac{\log n}{\log\log n})$ time each, takes $\mathrm{O}(n)$ space, and $\mathrm{O}(n\frac{\log n}{\log\log n})$ time to construct.

Now, to compute the longest closed factor starting at a given position i in X (i.e. to compute $\mathsf{A}[i]$) we do the following. First we compute b_i, the length of the border of the longest closed factor starting at i, in $\mathrm{O}(1)$ time using Lemma 6. Recall that in the process of computing b_i we determine the node u having $\mathsf{pathlabel}(u) = \mathsf{X}[i..i+b_i-1]$.

To determine the end of the closed factor we must find the smallest $j > i$ such that $\mathsf{X}[j..j+b_i-1] = \mathsf{X}[i..i+b_i-1]$. Observe that j, if it exists, is precisely the answer to $\mathsf{rsq}_{\mathsf{SA}}(u.s, u.e, i)$. (See also the left diagram of Figure 2. Assuming that the leaves in the subtree rooted at u are sorted in the lexicographical order, the leftmost and rightmost leaves in the subtree correspond to the $u.s$-th and $u.e$-th entries of SA, respectively. Hence, $j = \mathsf{rsq}_{\mathsf{SA}}(u.s, u.e, i)$). For each $\mathsf{A}[i]$ we spend $O(\frac{\log n}{\log\log n})$ time and so overall the algorithm takes $O(n\frac{\log n}{\log\log n})$ time. The space requirement is clearly $O(n)$. □

We note that recently Navarro and Neckrich [7] described range successor data structures with faster $O(\sqrt{\log n})$-time queries, but straightforward construction takes $O(n \log n)$ time [6], so overall this does not improve the runtime of our algorithm.

5 Concluding Remarks

We have considered but two problems on closed factors here, and many others remain. For example, how efficiently can one compute all the closed factors in a string (or, say, the closed factors that occur at least k times)? Relatedly, how many closed factors does a string contain in the worst case and on average?

One also wonders if the closed factor array can be computed in linear time, by somehow avoiding range successor queries.

References

1. G. Badkobeh, G. Fici, and Zs. Lipták: *A note on words with the smallest number of closed factors*. `http://arxiv.org/abs/1305.6395`, 2013.
2. M. Farach: *Optimal suffix tree construction with large alphabets*, in Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 1997, pp. 137–143.
3. G. Fici: *A classification of trapezoidal words*, in Proc. 8th International Conference Words 2011 (WORDS 2011), Electronic Proceedings in Theoretical Computer Science 63, 2011, pp. 129–137, See also `http://arxiv.org/abs/1108.3629v1`.
4. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
5. U. Manber and G. W. Myers: *Suffix arrays: a new method for on-line string searches*. SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
6. G. Navarro and Y. Neckrich: Personal Communication.
7. G. Navarro and Y. Neckrich: *Sorted range reporting*, in Proc. Scandinavian Workshop on Algorithm Theory, LNCS 7357, Springer, 2012, pp. 271–282.
8. P. Weiner: *Linear pattern matching*, in IEEE 14th Annual Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.
9. C.-C. Yu, W.-K. Hon, and B.-F. Wang: *Improved data structures for the orthogonal range successor problem*. Computational Geometry, 44(3) 2011, pp. 148–159.