New Suffix Array Algorithms — Linear But Not Fast?

Antonitio[1], P. J. Ryan[2], W. F. Smyth[1,2⋆], Andrew Turpin[1⋆⋆], and Xiaoyang Yu[2]

[1] Department of Computing, Curtin University of Technology, Perth WA 6845, Australia
andrew@computing.edu.au
[2] Algorithms Research Group, Department of Computing & Software McMaster University, Hamilton, Ontario, Canada L8S 4K1
{ryanpj,smyth}@mcmaster.ca

Abstract. In 2003 three $\Theta(n)$-time algorithms were proposed for the construction of a suffix array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an indexed alphabet, all of them inspired by the methodology of Farach's $\Theta(n)$-time suffix tree construction algorithm. In the same year a $\Theta(m)$-time algorithm was described for computing all the occurrences of a pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in $\boldsymbol{x}$, given a suffix array of $\boldsymbol{x}$ and various other $\Theta(n)$-space data structures. We analyze the effectiveness and limitations of these algorithms, especially in terms of execution time, and we discuss strategies for their improvement.

Keywords: String, suffix array, algorithm, linear, construction, pattern-matching.

1 Introduction

Suffix arrays were introduced in 1993 [10] as a space-saving alternative data structure to suffix trees [14], with nevertheless an equally wide range of applications to pattern-matching and to string algorithms generally [1]. In the 1990s the first truly linear-time algorithm for suffix tree construction on an indexed alphabet was proposed by Farach [4], though the method involved a complex space-consuming recursion that made it unsuitable for long strings. In the same decade several faster suffix array construction algorithms were also proposed, notably that of Sadakane [11]. In 2003 no less than three

⋆ Supported in part by a grant from the Natural Sciences & Engineering Research Council of Canada.
⋆⋆ Communicating author.

linear-time suffix array construction algorithms were proposed [5, 7, 8] — all of them using a recursive approach inspired by [4] —, and in addition a linear-time pattern-matching algorithm using suffix arrays [12]. Thus, apparently, suffix arrays became also a time-saving alternative to suffix trees, and hence the data structure of choice for a very large number of string processing problems.

In this paper we study in particular two [5, 12] of the new algorithms, comparing the running time of various implementations of them with those of established, but in theory supralinear, algorithms proposed earlier. To our surprise, we find that the new linear-time algorithms are not faster in practice than the best of the previously reported algorithms. We discuss the reasons for this phenomenon and the prospects for run time improvement.

In Section 2 we discuss the methodology of the new algorithms and provide a brief outline of two of them. Section 3 gives the results of tests of implementations of the algorithms on a variety of well-known file collections. In Section 4 we discuss the prospects for improvement of the linear-time algorithms or for the derivation of new algorithms from them.

2 Methodology of the New Algorithms

Throughout this paper we assume that a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is given, with elements drawn from an ***indexed*** alphabet — that is, an alphabet in which any letter λ directly accesses position λ in an array. Without loss of generality, then, we assume that the alphabet $\Sigma = \{1, 2, \ldots, k\}$, where k is a constant. In addition, we make use from time to time of a special (and smallest) letter 0 that does not however appear in $\boldsymbol{x}$. When discussing pattern-matching, we assume that the pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ is also drawn from the same alphabet Σ.

Recall that in the ***suffix array*** $\boldsymbol{\sigma_x} = \boldsymbol{\sigma_x}[1..n]$ of $\boldsymbol{x}$, $\boldsymbol{\sigma_x}[i] = j$ if and only if the suffix $\boldsymbol{x}[j..n]$ is the i^{th} smallest in lexicographical order over all the suffixes of $\boldsymbol{x}$. This ordering requirement means that all the suffixes with the same ***longest common prefix*** (LCP) must occur consecutively in $\boldsymbol{\sigma_x}$.

As noted in Section 1, three new suffix array construction algorithms were proposed in 2003 [5, 7, 8]. Of these, the second [7] is according to its authors of theoretical interest, not for implementation, while we were not able to obtain an implementation of the third [8]. Thus in this paper, we consider the construction algorithm [5] only, which we refer to as Algorithm KS. In addition, we consider the pattern-matching algorithm [12], referred to here as Algorithm SKPP. The remainder of this section provides an overview of these two algorithms.

2.1 Algorithm KS

We are given a string $\boldsymbol{x}$ on Σ:

$$\begin{array}{c} \begin{array}{rcccccccccccc} & & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} \\ \boldsymbol{x} & = & a & b & r & a & c & a & d & a & b & r & a\ \$ \end{array} \\ \text{or} \\ 1\ 2\ 5\ 1\ 3\ 1\ \ 4\ \ 1\ 2\ 5\ \ 1\ \ 0. \end{array} \tag{1}$$

We wish to compute its suffix array:

$$\begin{array}{rcccccccccccc} & & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} \\ \boldsymbol{\sigma_x} & = & 11 & 8 & 1 & 4 & 6 & 9 & 2 & 5 & 7 & 10 & 3. \end{array} \tag{2}$$

Algorithm KS first radix-sorts the letters in positions $i \equiv 1 \bmod 3$:

$$C = \{1, 1, 4, 5\} \text{ in } \{1, 4, 7, 10\}.$$

The observation is made that, given the suffixes s_i at positions $i \not\equiv 1 \bmod 3$ sorted in lexicographic order, $i \not\equiv 1 \bmod 3$ sorted in ***lexorder***,

$$S = \{11, 8, 6, \underline{9, 2}, 5, 3\}, \tag{3}$$

it is easy in $O(n)$ time to produce a merged list of *all* the suffixes of $\boldsymbol{x}$ (thus, the suffix array $\boldsymbol{\sigma_x}$).

KS *approximates* the sorted suffix array S by an array T of sorted ***triples*** $\boldsymbol{t_i} = \boldsymbol{x}[i..i{+}2]$, $i \not\equiv 1 \bmod 3$:

$$T = \{11, 8, 6, \underline{2, 9}, 5, 3\}.$$

Because the alphabet is indexed, a stable radix sort can be used to sort the triples in $\Theta(2n/3)$ time; on the other hand, if two triples are identical (for example, those in positions 2 and 9), sorting triples cannot be guaranteed to correctly represent the order of the corresponding suffixes. To get around this problem, KS assigns to each triple $\boldsymbol{t_i}$ a rank r_i:

i	$\boldsymbol{t_i}$	r_i
11	$10\cdot$	1
11	$a\$\cdot$	1
8	125	2
6	141	3
2	251	4
9	251	4
5	314	5
3	513	6

Then a new string $\boldsymbol{x}^{(1)}$ is formed using the ranks r_i as alphabet, while retaining the order already computed for T:

$$\begin{aligned}\boldsymbol{x}^{(1)} &= (r_i,\ i \equiv 2 \bmod 3)\ (r_i,\ i \equiv 0 \bmod 3)\\ &= \overset{1}{(1}\ \overset{2}{2}\ \overset{3}{\underline{4}}\ \overset{4}{5)}\ \overset{5}{(3}\ \overset{6}{\underline{4}}\ \overset{7}{6)}\ \overset{8}{\$}\end{aligned}$$

The formation of $\boldsymbol{x}^{(1)}$ is justified by

Theorem 1. *The order of the suffixes of $\boldsymbol{x}^{(1)}$ corresponds to the order of the suffixes in S:*

$$\begin{aligned} S &= \{11, 8, 6, \underline{9, 2}, 5, 3\}\\ \boldsymbol{\sigma}_{\boldsymbol{x}^{(1)}} &= \quad 1 \ \ 2\ 5\ \ \underline{3\ 6}\ 4\ 7\end{aligned}$$

This result is the basis of the recursive step in KS: apply the *same process* (sort letters $\equiv 1 \bmod 3$ and triples $\not\equiv 1 \bmod 3$) to $\boldsymbol{x}^{(1)}$ (and, if necessary, to $\boldsymbol{x}^{(2)}$, $\boldsymbol{x}^{(3)}$, ...) until all triples are distinct. Then do a merge at each level of recursion to determine each $\boldsymbol{\sigma}_{\boldsymbol{x}^{(i)}}$, and finally $\boldsymbol{\sigma}_{\boldsymbol{x}}$.

The first step requires $O(n)$ time, the i^{th} recursive step $O(2^i n/3^i)$ time, $i = 1, 2, \ldots$ — overall $O(n)$ time. KS is a linear time suffix array construction algorithm!

2.2 Algorithm SKPP

Now we suppose that $\boldsymbol{x}$ and $\boldsymbol{\sigma}_{\boldsymbol{x}}$ are given, as in (1) and (2). We seek to compute every occurrence of a pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in $\boldsymbol{x}$; for example, $\boldsymbol{p} = 251$ (bra). Algorithm SKPP requires $O(m \log k)$ time;

it depends upon $O(n)$ time preprocessing of $\boldsymbol{x}$ and $\boldsymbol{\sigma_x}$ to compute four arrays ($b = \lceil n/k \rceil$):

$$W = W[1..k]; \quad X = X[1..b, 0..k];$$
$$Y = Y[1..b, 1..k{+}1]; \;\; Z = Z[1..b, 1..k{+}1].$$

We recast our example slightly to explicitly include position 12:

$$\begin{array}{rcccccccccccccc} & & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \boldsymbol{x} & = & 0 & 1 & 2 & 5 & 1 & 3 & 1 & 4 & 1 & 2 & 5 & 1 & 0 \\ \boldsymbol{\sigma_x} & = & 12 & 11 & 8 & 1 & 4 & 6 & 9 & 2 & 5 & 7 & 10 & 3 & \end{array}$$

We can now define W for every $j \in 1..k$ as follows:

$W[j] = i$ iff $\boldsymbol{x}\big[\boldsymbol{\sigma_x}[i]\big] = j$ and $\boldsymbol{x}\big[\boldsymbol{\sigma_x}[i']\big] \neq j$ for every $i' < i$ (the leftmost reference to letter j in $\boldsymbol{\sigma_x}$); zero if no j in $\boldsymbol{x}$.

$$\begin{array}{rcccccc} & & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} \\ W & = & 1 & 6 & 8 & 9 & 10 \end{array}$$

If every j occurs in $\boldsymbol{x}$, then $W[j{+}1] - W[j]$ is the number of occurrences of j, $1 \le j < k$.

The arrays X, Y & Z are computed in order to permit efficient implementation of a ***virtual array*** $N = N[-1..n, 0..k]$. We describe N because it is easier!

For every $j \in 0..k$, $N[-1, j] = 0$, while for $i \ge 0$, $N[i, j]$ counts the occurrences of $j\boldsymbol{x}\big[\boldsymbol{\sigma_x}[i']..n\big]$ in $\boldsymbol{x}$ over all $i' \in 0..i$ — thus occurrences of j that precede consecutive (in lexorder) suffixes of $\boldsymbol{x}$:

$$N = \begin{array}{r|cccccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} \\ {\scriptstyle -1} & 0 & 0 & 0 & 0 & 0 & 0 \\ {\scriptstyle 0} & 0 & 1 & 0 & 0 & 0 & 0 \\ {\scriptstyle 1} & 0 & 1 & 0 & 0 & 0 & 1 \\ {\scriptstyle 2} & 0 & 1 & 0 & 0 & 1 & 1 \\ {\scriptstyle 3} & 1 & 1 & 0 & 0 & 1 & 1 \\ {\scriptstyle 4} & 1 & 1 & 0 & 0 & 1 & 2 \\ {\scriptstyle 5} & 1 & 1 & 0 & 1 & 1 & 2 \\ {\scriptstyle 6} & 1 & 2 & 0 & 1 & 1 & 2 \\ {\scriptstyle 7} & 1 & 3 & 0 & 1 & 1 & 2 \\ {\scriptstyle 8} & 1 & 4 & 0 & 1 & 1 & 2 \\ {\scriptstyle 9} & 1 & 5 & 0 & 1 & 1 & 2 \\ {\scriptstyle 10} & 1 & 5 & 1 & 1 & 1 & 2 \\ {\scriptstyle 11} & 1 & 5 & 2 & 1 & 1 & 2 \end{array}$$

Algorithm SKPP first finds all matches in $\boldsymbol{x}$ for $\boldsymbol{p}[m]$ — this is a range $[i_{\mathrm{S}}..i_{\mathrm{e}}]$ in $\boldsymbol{\sigma_x}$. Then it finds the subrange of positions of $\boldsymbol{p}[m]$ that are preceded by $j = \boldsymbol{p}[m-1]$ — in general, the subrange of positions of $\boldsymbol{p}[h+1]\boldsymbol{p}[h+2]\cdots\boldsymbol{p}[m]$ that are preceded by $j = \boldsymbol{p}[h]$:

procedure SKPP
$h \leftarrow m$; $i_{\mathrm{S}} \leftarrow 0$; $i_{\mathrm{e}} \leftarrow n$
while $h > 0$ **and** $i_s \leq i_e$ **do**
  $j \leftarrow \boldsymbol{p}[h]$
  $i_{\mathrm{S}} \leftarrow W[j] + N[i_{\mathrm{S}}-1, j]$
  $i_{\mathrm{e}} \leftarrow W[j] + N[i_{\mathrm{e}}, j] - 1$
  $h \leftarrow h-1$
if $i_s \leq i_e$ **then output** (i_s, i_e)

Note that the algorithm actually yields all occurrences of every suffix of $\boldsymbol{p}$. For $\boldsymbol{p} = 251$,

$$\begin{aligned}
h = 3 &\Rightarrow i_{\mathrm{S}} \leftarrow W[1]+N[-1,1] = 1+0 = 1;\\
&\Rightarrow i_{\mathrm{e}} \leftarrow W[1]+N[11,1]-1 = 1+5-1 = 5;\\
h = 2 &\Rightarrow i_{\mathrm{S}} \leftarrow W[5]+N[0,5] = 10+0 = 10;\\
&\Rightarrow i_{\mathrm{e}} \leftarrow W[5]+N[5,5]-1 = 10+2-1 = 11;\\
h = 1 &\Rightarrow i_{\mathrm{S}} \leftarrow W[2]+N[9,2] = 6+0 = 6;\\
&\Rightarrow i_{\mathrm{e}} \leftarrow W[2]+N[11,2]-1 = 6+2-1 = 7.
\end{aligned}$$

The calculation requires $\Theta(m)$ time, but the preprocessing for N requires $O(nk)$ time; using X, Y, Z reduces preprocessing to $\Theta(n)$ time, while increasing calculation time to $O(m \log k)$. The standard approach requires $\Theta(n)$ time for preprocessing together with $O(m+\log n)$ time for each search.

3 Tests on the New Algorithms

In this section we describe the results of tests carried out on implementations of KS (respectively, SKPP) in order to compare efficiency with other suffix structure construction (respectively, pattern-matching) algorithms. The 82 test files used ranged in size from 3.7 Kbytes to 10.4 Mbytes, and were drawn from the collections identified in Table 1. In addition 25 random files were generated (using the C pseudorandom number generator `rand`) on alphabets of size 2, 4, 26, 85 and 128. Most of the tests were carried out both

collection	URL	# files
Calgary	`www.data-compression.info/Corpora/CalgaryCorpus`	18
Canterbury	`www.data-compression.info/Corpora/CanterburyCorpus`	11
Large Canterbury	`www.data-compression.info/Corpora/CanterburyCorpus`	3
DNA	`www.ebi.ac.uk/embl`	19
Gutenberg	`www.promo.net/pg`	6

Table 1. File Collections Used for Testing

in Australia (Curtin University) and Canada (McMaster University), in some cases using different implementations; at McMaster all tests were run on two SunW Ultra_4 sparc workstations with differing CPU speeds, cache sizes and memory sizes, and in addition the source code was compiled using two different C compilers. The Unix `times()` utility was used to give average user time over 1000 runs on each input data set. There was very good agreement among test results across variations in location, platform and compiler.

3.1 Algorithm KS

Among suffix structure construction algorithms currently available, Sadakane's suffix array algorithm [11] and Kurtz's implementation of McCreight's suffix tree algorithm [9] are perhaps the most efficient, both in terms of time and space. Sadakane's algorithm (Algorithm S) requires only $9n$ bytes of storage and executes in $O(n \log^2 n)$ worst case time; Kurtz's algorithm (Algorithm KM) uses as little as $20n$ bytes while executing in $\Theta(n)$ time. In practice Algorithm S appears generally to have an advantage over KM [16, p. 91], and so we confined ourselves to testing S.

In the first instance we compared an implementation of Algorithm KS, downloaded from

`www.mpi-sb.mpg.de/~sanders/program/suffix`,

with one of Algorithm S, available from `sada@is.s.u-tokyo.ac.jp`. Our basic finding was that Algorithm S was 2–3 times faster than Algorithm KS, a result consistent over all sizes and types of data file except for random files on alphabets of sizes 2 and 4; on these random files, KS was nearly twice as fast as S. Both implementations behaved in practice as if they were $\Theta(n)$, but with KS having a constant of proportionality generally 2–3 times that of S. The anomaly for random files on small alphabets seems to derive from the observation [6] that the expected length of the longest repeating substring in a random file is $O(\log_k n)$ — hence the expected depth of the recursion for KS applied to a random file will be $O(\log_{k^3} n)$, but as a rule very much larger for a nonrandom file. Thus KS has an advantage on random files, particularly those on a small alphabet, that it loses on nonrandom ones.

By profiling the execution of KS, we discovered that about 40% of the execution time was absorbed by nested indirect addressing in the radix sorts. Furthermore, the depth of recursion for nonrandom files was often large, indicating the frequent occurrence of at least one pair of long repeating substrings. These observations led us to experiment with variant implementations, as follows:

- replacing the array-based radix sort with a list-based radix sort;
- replacing the recursive steps with quicksort;
- using alphabet size k^3 in the radix sort.

None of these variants performed better than the original downloaded KS implementation.

3.2 Algorithm SKPP

In 1992 two papers were published [2, 15] that described a fast bitmapping approach to pattern-matching that exploited the inherent parallelism of the bits in a computer word. (As noted in [13], these papers essentially rediscovered a methodology originally proposed [3] in the mid-1960s.) The `agrep` utility, based primarily on [15], has for 10 years or so provided a standard for exact and approximate pattern-matching that is both fast and flexible.

To test the effectiveness of Algorithm SKPP, therefore, we conducted comparisons of our own implementations (one at Curtin, one at McMaster) of SKPP against `agrep` downloaded from

`www.manber.com/software.html`.

For SKPP there were two basic cases to consider:

SKPP-1 The preprocessing time for the four arrays W, X, Y, Z was included in the timing for the algorithm. (Generally Algorithm KS's time requirement for suffix array construction was 3–10 times that required to compute all of W, X, Y, Z.)

SKPP-2 Preprocessing time was not included in the timing — thus only the $\Theta(m)$ time for the pattern-matching algorithm (**procedure** SKPP) itself.

As expected SKPP-2 was much faster than `agrep`, ranging from 100 times as fast for $n \approx 10^4$ to $100,000$ times as fast for $n \approx 10^7$. On the other hand, SKPP-1 generally ran 10–50 times as slowly as `agrep`, so that if all the preprocessing were included (time for suffix array construction as well), the time requirement could easily be 100 times or more that of `agrep`. Thus an approach to pattern-matching based on the use of Algorithms KS and SKPP could only be justified over use of `agrep` if there were a requirement to search for multiple patterns $\boldsymbol{p}$, perhaps 100 or so, in $\boldsymbol{x}$.

A final remark: we did tests on the use of the array N defined in Subsection 2.2 instead of X, Y, Z, but found an advantage in overall performance only for alphabet size $k = 2$.

4 Conclusions & Future Prospects

Our results suggest that

- the linear-time recursive algorithm KS for suffix array construction is not as fast in practice as Sadakane's $O(n \log^2 n)$ algorithm;
- an approach to pattern-matching based on suffix arrays should be considered only if multiple searches (≥ 100) of the text string $\boldsymbol{x}$ are expected.

The basic problem with Algorithm KS (as with the other two algorithms [7, 8] derived from Farach's algorithm [4]) is that, in order to distinguish among what may be a small number of identical triples, it is necessary to involve *all* the triples in a recursive process. In our example, this means that, even though in (3) only two of the triples (9 and 2) need to be distinguished in order to obtain the correct ordering of the suffixes, nevertheless all of the triples given in 3 need to be considered in the recursive step. It appears that only by somehow avoiding this duplication of effort can a major improvement be made in the efficiency of suffix array construction.

As perhaps a step in this direction, we observe that the problem of ordering identical triples to conform to the ordering of suffixes can be formulated as a graph traversal problem. Suppose that the triples $x[i..i+2]$, $i \equiv 0$ or $2 \pmod 3$, have been radix-sorted and assigned ranks $1, 2, \ldots, \rho$. This information can be represented as a digraph $G = (V, A)$ with vertices V of two kinds:

- ρ ***rank*** vertices R with distinct labels $1, 2, \ldots, \rho$;
- $2n/3$ ***position*** vertices P with distinct labels $2, 3, 5, 6, \cdots$.

These vertices would then give rise to arcs of three kinds:

- arcs (r, p) from $r \in R$ to $p \in P$ if and only if $x[p..p+2]$ is of rank r;
- arcs $(r, r+1)$ for every $r < \rho$;
- arcs $(p, p+3)$ for every $p \leq n-3$.

Thus, for the string

$$x = \overset{1}{a}\,\overset{2}{b}\,\overset{3}{c}\,\overset{4}{a}\,\overset{5}{b}\,\overset{6}{d}\,\overset{7}{r}\,\overset{8}{s}\,\overset{9}{t}\,\overset{10}{a}\,\overset{11}{b}\,\overset{12}{c}\,\overset{13}{a}\,\overset{14}{b}\,\overset{15}{d}\,\overset{16}{s}\,\overset{17}{t}\,\overset{18}{r},$$

we would form G as shown in Figure 1.

Observe that, in this example, a short, simple search would permit the two pairs $(2, 11)$ and $(3, 12)$ of identical rank to be separated according to the lexicographic order of the suffixes. And, in general, it turns out that corresponding to every arc followed in such a graph the ordering of one suffix can be determined, essentially a linear time operation. The advantage of this formulation is that it is linear and requires a search only for each triple that is not unique; the disadvantage is that the processing can be $O(n^2)$ in the worst case.

References

1. Alberto Apostolico, **The myriad virtues of subword trees**, *Combinatorial Algorithms on Words (NATO ASI Series F12)*, Alberto Apostolico & Zvi Galil (eds.), Springer-Verlag (1985) 85–96.

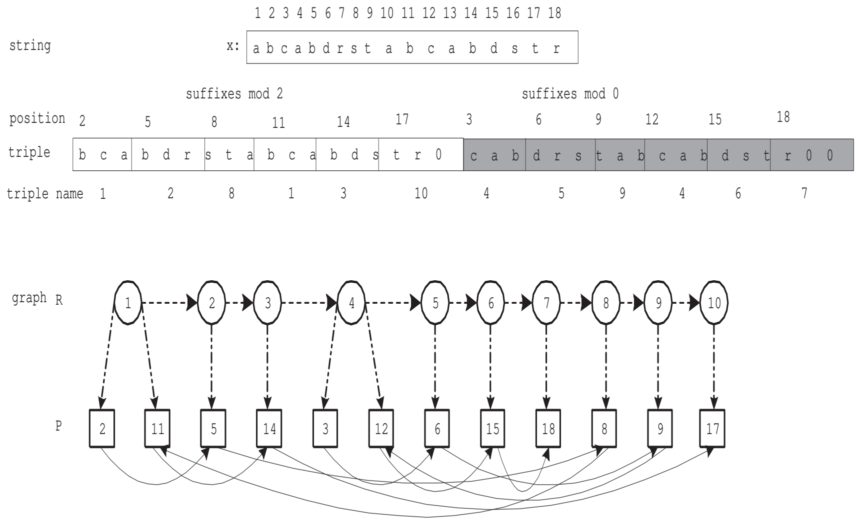

Fig. 1. The graph formulation G of $x = abcabdrstabcabdstr$

2. Ricardo A. Baeza-Yates & Gaston H. Gonnet, **A new approach to text searching**, *CACM 35–10* (1992) 74–82.
3. Bálint Dömölki, **A universal computer system based on production rules**, BIT 8 (1968) 262–275.
4. Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc.* 38$^{\text{th}}$ *Annual IEEE Symp. Foundations of Computer Science* (1997) 137–143.
5. Juha Kärkkäinen & Peter Sanders, **Simple linear work suffix array construction**, *Proc.* 30$^{\text{th}}$ *Internat. Colloq. Automata, Languages & Programming* (2003) 943–955.
6. S. Karlin, G. Ghandour, F. Ost, S. Tavare & L. J. Korn, **New approaches for computer analysis of nucleic acid sequences**, *Proc. Natl. Acad. Sci. USA 80* (1983) 5660–5664.
7. Dong Kyue Kim, Jeong Seop Sim, Heejin Park & Kunsoo Park, **Linear-time construction of suffix arrays**, *Proc.* 14$^{\text{th}}$ *Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003) 186–199.
8. Pang Ko & Srinivas Aluru, **Space efficient linear time construction of suffix arrays**, *Proc.* 14$^{\text{th}}$ *Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003) 200–210.
9. Stefan Kurtz, **Reducing the space requirement of suffix trees**, *Software – Practice & Experience29–13* (1999) 1149–1171.
10. Udi Manber & Gene W. Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM J. Comput. 22–5* (1993) 935–948.

11. Kunihiko Sadakane, **A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation**, *Proc. IEEE Data Compression Conference* (1998) 129–138.
12. Jeong Seop Sim, Dong Kyue Kim, Heejin Park & Kunsoo Park, **Linear-time search in suffix arrays**, *Proc.* 14^{th} *Australasian Workshop on Combinatorial Algorithms*, Mirka Miller & Kunsoo Park (eds.) (2003) 139–146. *Proc.* 14^{th} *Australasian Workshop on Combinatorial Algorithms* (2003) 139–146.
13. Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.
14. Peter Weiner, **Linear pattern matching algorithms**, *Proc. 14th Annual IEEE Symp. Switching & Automata Theory* (1973) 1–11.
15. Sun Wu & Udi Manber, **Fast text searching allowing errors**, *CACM 35–10* (1992) 83–91.
16. Xiangdong Xiao, *Computing Quasi Suffix Arrays*, M.Sc. thesis, McMaster University (2003).