Algorithms Seminar WS 2019/20 10.12.2019

Direct Suffix Array Construction in Linear Time

Irina Makarenko

1 Introduction

A suffix array is the lexicographically sorted array of the suffixes of a string. It holds the same information as a suffix tree, though in a more implicit form, but is a much simpler and more compact data structure for many applications. For a long time, the only linear-time suffix array construction algorithm was based on a lexicographic traversal of the suffix tree, the *indirect* suffix array construction. One of the more prominent linear-time algorithms for constructing a suffix tree, Farach's algorithm (1997), adopts the following recursive divide-and-conquer approach:

1. Construct the suffix tree of the suffixes starting at odd positions. This is done by reduction to the suffix tree construction of a string of half the length, which is solved recursively.
2. Construct the suffix tree of the remaining suffixes using the result of the first step.
3. Merge the two suffix trees into one.

But the merge in the third step is a complicated procedure. The direct linear-time suffix array construction algorithm of Kim, Sim, Park and Park (2003) has the same structure and therefore also includes a very tedious merge process. Another approach for a direct linear-time suffix array construction follows the general idea of Farach's algorithm but uses a 2/3-recursion instead of a half-recursion:

1. Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Such a simple modification is enough to allow for basic comparison-based merging in the last step. This algorithm is called the DC3 algorithm and was developed by Kärkkäinen, Sanders and Burkhardt (2006).

In this paper, the individual steps of algorithm DC3 and its runtime are presented. Afterwards difference cover samples are introduced and building on that a general form of the DC3 algorithm is discussed.

2 Notations & Definitions

The **input** of the suffix array construction algorithm is a string $T = T[0, n) = t_0 t_1 \dots t_{n-1}$ over the integer alphabet $[1, n]$, that is, a sequence of n integers from the range $[1, n]$. [1] We assume

[1] The restriction to the alphabet $[1, n]$ is not a serious one. See Kärkkäinen et al. (2006: 5) for details.

that $t_j = 0$ for $j \geq n$. For $i \in [0, n]$, let S_i denote the *suffix* $T[i, n) = t_i t_{i+1} \dots t_{n-1}$. For $C \subseteq [0, n], S_C = \{\mathrm{S}_i \mid i \in C\}$ is the set of suffixes given by C. The goal is to sort the set $S_{[0,n]}$ of suffixes of T in lexicographic order. The **output** is the *suffix array* $SA[0, n]$ of T, a permutation of $[0, n]$ satisfying $\mathrm{S}_{SA[0]} < \mathrm{S}_{SA[1]} < \dots < \mathrm{S}_{SA[n]}$.

3 Suffix Array Construction Algorithm DC3

A detailed description of the linear-time algorithm DC3 as presented by Kärkkäinen et al. (2006) follows. The execution of the algorithm is illustrated using the example string

$$\begin{array}{rc} & \scriptstyle 0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10\ 11 \\ T[0,n) = & \texttt{y}\ \texttt{a}\ \texttt{b}\ \texttt{b}\ \texttt{a}\ \texttt{d}\ \texttt{a}\ \texttt{b}\ \texttt{b}\ \texttt{a}\ \texttt{d}\ \texttt{o}\ \texttt{.} \end{array}$$

We are looking for the suffix array $SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$.

Step 0: Construct a sample.

For $k = 0, 1, 2$, define $B_k = \{i \in [0, n] \mid i \bmod 3 = k\}$. Let $C = B_1 \cup B_2$ be the set of sample positions and S_C the set of sample suffixes.

Example: $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Step 1: Sort sample suffixes.

For $k = 1, 2$, construct the strings $R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \dots [t_{\max B_k} t_{\max B_k + 1} t_{\max B_k + 2}]$ consisting of the triples $[t_i t_{i+1} t_{i+2}]$, so-called characters. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. Then the suffixes of R correspond to the set S_C of sample suffixes: $[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \dots$ corresponds to S_i. The correspondence preserves the order, by sorting the suffixes of R we get the order of the sample suffixes S_C.

Example: $R = [\texttt{abb}][\texttt{ada}][\texttt{bba}][\texttt{do0}][\texttt{bba}][\texttt{dab}][\texttt{bad}][\texttt{o00}]$.

Sort the characters of R and rename them with their ranks to obtain the string R'. If all ranks are unique, the order of the characters can directly be transferred to the order of suffixes. If not, sort the suffixes of R' by recursively calling algorithm DC3.

Example: $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ and $SA_R = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.

Use the previous result to assign ranks to the sample suffixes. For $i \in C$, let $rank(\mathrm{S}_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $rank(\mathrm{S}_{n+1}) = rank(\mathrm{S}_{n+2}) = 0$. For $i \in B_0$, $rank(\mathrm{S}_i)$ is undefined.

Example:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(\mathrm{S}_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0 .

Step 2: Sort nonsample suffixes.

Represent each nonsample suffix $\mathrm{S}_i \in S_{B_0}$ with the tuple $(t_i, rank(\mathrm{S}_{i+1}))$. To sort all suffixes S_i, S_j with $i, j \in B_0$, we use:

$$\mathrm{S}_i \leq \mathrm{S}_j \iff (t_i,\, rank(\mathrm{S}_{i+1})) \leq (t_j,\, rank(\mathrm{S}_{j+1})).$$

Example: Since $(0,0) \leq (a,5) \leq (a,7) \leq (b,2) \leq (y,1)$, we have $S_{12} \leq S_6 \leq S_9 \leq S_3 \leq S_0$.

Step 3: Merge.

Combine the two sorted sets of suffixes using a standard comparison-based merging. To compare suffix $\mathrm{S}_i \in S_C$ and $S_j \in S_{B_0}$, distinguish two cases:

$$\begin{aligned} i \in B_1 \;:\; & \mathrm{S}_i \leq \mathrm{S}_j \iff (t_i,\, rank(\mathrm{S}_{i+1})) \leq (t_j,\, rank(\mathrm{S}_{j+1})) \\ i \in B_2 \;:\; & \mathrm{S}_i \leq \mathrm{S}_j \iff (t_i,\, t_{i+1},\, rank(\mathrm{S}_{i+2})) \leq (t_j,\, t_{j+1},\, rank(\mathrm{S}_{j+2})) \end{aligned}$$

3.1 Runtime

Kärkkäinen et al. (2006) established the following theorem regarding the runtime of DC3:

Satz 1. *The time complexity of algorithm DC3 is* $\mathcal{O}(n)$.

Proof. Excluding the recursive call, everything can clearly be done in linear time. The recursion is on a string of length $\lceil 2n/3 \rceil$. Thus the time is given by the recurrence $T(n) = T(2n/3) + \mathcal{O}(n)$, whose solution is $T(n) = \mathcal{O}(n)$. □

Using radix sort for sorting tasks as suggested by the authors every of the above steps except the recursion can indeed be done in linear time.

4 Difference Cover Sample

The sample used by the algorithm DC3 has to satisfy two conditions:

1. The sample itself can be sorted efficiently.
2. The sorted sample helps in sorting the set of all suffixes.

Difference cover samples fulfill both these conditions and so the sample of suffixes in DC3 is a special variant of a difference cover sample.

A difference cover sample is based on difference covers:

Definition 2. *A set* $D \subseteq [0, v)$ *is a* difference cover *modulo* v *if* $\{(i-j) \bmod v \mid i, j \in D\} = [0, v)$.

Definition 3. *A* v*-periodic sample* C *of* $[0, n]$ *with the period* D*, i.e.,* $C = \{i \in [0, n] \mid i \bmod v \in D\}$*, is a* difference cover sample *if* D *is a difference cover modulo* v.

Kärkkäinen et al. (2006) showed that the following lemma holds:

Lemma 4. *If D is a difference cover modulo v, and i and j are integers, there exists $l \in [0, v)$ such that $(i + l) \bmod v$ and $(j + l) \bmod v$ are in D.*

Because of this lemma difference cover samples have the property that for any $i, j \in [0, n - v + 1]$ there is a small l such that both $(i + l)$ and $(j + l)$ are sample positions. This property ensures, that that sample can definitely be used to sort the complete set of suffixes. Also, difference cover samples can be sorted efficiently because they are periodic. So, in fact, both of the initally stated conditions are met and some sort of difference cover sample can be used as sample in DC3.

5 General Algorithm

DC3 means Difference Cover modulo 3. The algorithm DC3 sorts suffixes with starting positions in a difference cover sample with difference cover modulo 3 and then uses the result to sort all suffixes. The central idea behind the DC3 algorithm, difference covers, leads naturally to the generalization of it. Utilizing any difference cover modulo v gives the more general algorithm DC, which is fully described by Kärkkäinen, Sanders and Burkhardt in their paper from 2006.

6 Conclusion

Theoreticians prefer suffix trees over suffix arrays whereas practicians prefer suffix arrays over suffix trees. The reason for this is that suffix arrays are more space-efficient and simpler to implement, while suffix trees have a more explicit structure and direct linear-time construction algorithms. Kärkkäinen, Sanders and Burkhardt wanted to address the needs of theoreticians and practicians equally and designed a direct linear-time suffix array construction algorithm that is simple and space-efficient at the same time. According to them (Kärkkäinen et al. 2006), "the algorithm is easy to implement and [it] can be used as an example for advanced string algorithms even in undergraduate level algorithms courses. Its simplicity also makes it an ideal candidate for implementation on advanced models of computation". In their paper, they provided a realization of DC3 in the language C++, 50 effective lines of code, and also explained implementations of the DC3 algorithm in advanced models of computation to demonstrate their success in providing an algorithm that is as universal as required.

References

M. Farach (1997). "Optimal Suffix Tree Construction with Large Alphabets". In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pp. 137–143.

J. Kärkkäinen and P. Sanders (2003). "Simple Linear Work Suffix Array Construction". In: *Proceedings of the 30th International Conference on Automata, Languages and Programming (ICALP03)*. Ed. by J. C. M. Baeten et al. Vol. 2719. Lecture Notes in Computer Science. Springer, pp. 943–955.

J. Kärkkäinen, P. Sanders and S. Burkhardt (2006). "Linear Work Suffix Array Construction". In: *Journal of the ACM*. Vol. 53. 6, pp. 918–936.

D. K. Kim et al. (2003). "Linear-time Construction of Suffix Arrays". In: *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*. Vol. 2676. Lecture Notes in Computer Science. Springer, pp. 186–199.