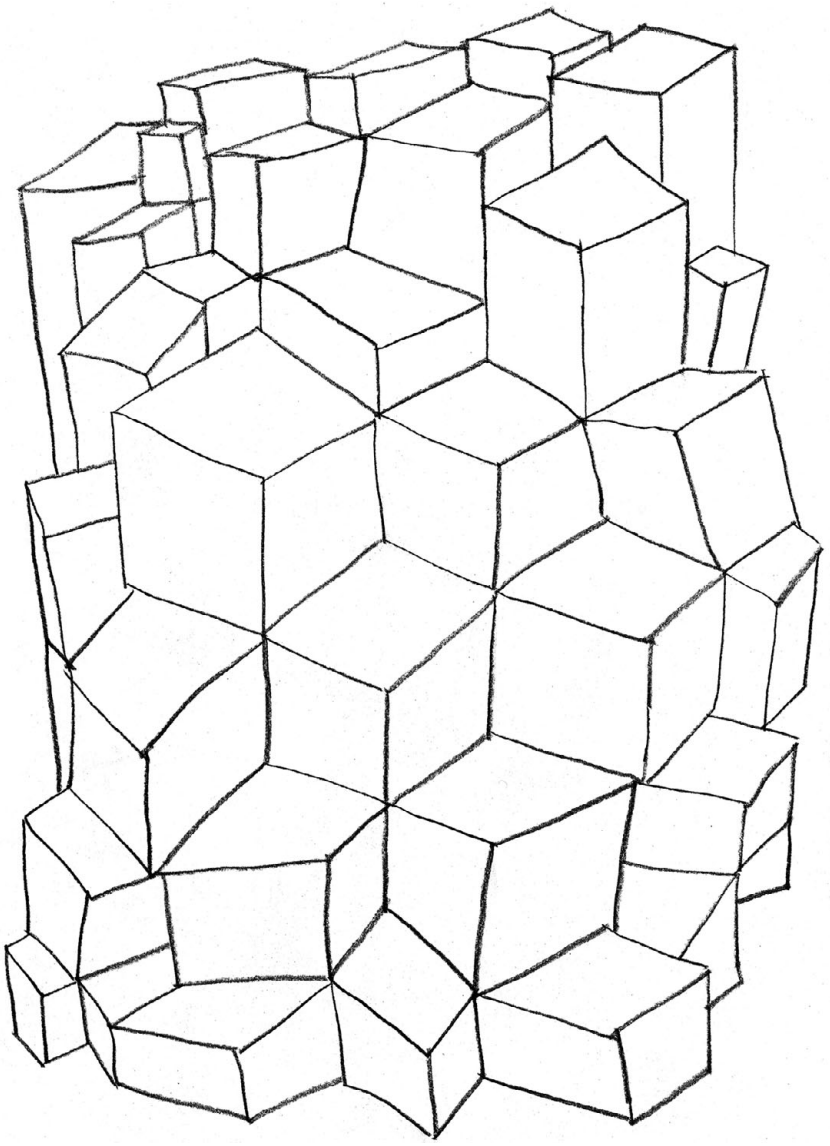
4 Efficient Data Structures



https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

45 List Algorithm for Shortest Cover

A cover of a non-empty word x is one of its factors whose occurrences cover all positions on x. As such it is a repetitive element akin to a repetition. The problem shows how to compute the shortest cover of a word using its prefix table *pref*, instead of its border table as in Problem 20. The algorithm is simpler but uses linear extra memory space.

For each length ℓ of a prefix of x, let $L(\ell) = (i : pref[i] = \ell)$. Algorithm ShortestCover computes the length of the shortest cover of its input.

```
ShortestCover(x non-empty word)
1  L ← (0, 1, ..., |x|)
2  for ℓ ← 0 to |x| − 1 do
3      remove elements of L(ℓ − 1) from L
4      if maxgap(L) ≤ ℓ then
5          return ℓ
```

Simulating a run on $x =$ `abababaaba`, positions in the list L for $\ell = 1, 2, 3$ are shown on the last lines. The associated values of $maxgap(L)$ are respectively 2, 3 and 3. The condition of line 4 is first met when $\ell = 3$, giving the shortest cover `aba` $= x[0\,..\,2]$.

i	0	1	2	3	4	5	6	7	8	9	
$x[i]$	a	b	a	b	a	b	a	a	b	a	
$pref[i]$	10	0	5	0	3	0	1	3	0	1	
$L - L[0]$	0		2		4		6	7		9	10
$L - L[\leq 1]$	0		2		4			7			10
$L - L[\leq 2]$	0		2		4			7			10

Question. Show that Algorithm ShortestCover computes the length of the shortest cover of its input and if properly implemented runs in linear time.

Solution

The correctness of ShortestCover is clear: it removes positions with small *pref* values, since their prefixes are too short and can be ignored. Eventually the condition is satisfied when $\ell = |x|$.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

If lists L and $L[\ell]$ are implemented, for example, by double-linked sorted lists, removing an element and updating *maxgap* simultaneously takes constant time per element. The overall running time is then linear, since each element is removed at most once from L and the total size of disjoint lists $L[\ell]$ is $|x| + 1$.

46 Computing Longest Common Prefixes

The Suffix array of a non-empty word y is a light and efficient solution for text indexing. It consists in using a binary search procedure to locate patterns inside y. To do so the suffixes of y are first sorted in lexicographic order, producing a table SA that lists the starting positions of the sorted suffixes.

But this standard technique is not sufficient to get a powerful search method. This is why the table SA is adjoined to a second table LCP that gives the length of ***longest common prefixes*** between consecutive suffixes in the sorted list (some more values easy to deduce are also needed). Using both tables, searching y for a word x is then achieved in time $O(|x| + \log |y|)$ instead of a straightforward $O(|x| \log |y|)$ time without the table LCP. Here is the Suffix array of `abaababababbabbb`:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$y[j]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b	
Rank r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[r]	2	0	3	5	7	10	13	1	4	6	9	12	8	11	
LCP[r]	0	1	3	4	2	3	0	1	2	3	4	1	2	2	0

where $\text{LCP}[r] = |\text{lcp}(y[\text{SA}[r-1]\,..\,|y|-1], y[\text{SA}[r]\,..\,|y|-1])|$.

Question. Given the table SA for the word y, show that Algorithm LCP computes the associated table LCP in linear time.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

```
LCP(y non-empty word)
 1  for r ← 0 to |y| − 1 do
 2      Rank[SA[r]] ← r
 3  ℓ ← 0
 4  for j ← 0 to |y| − 1 do
 5      ℓ ← max{0, ℓ − 1}
 6      if Rank[j] > 0 then
 7          while max{j + ℓ, SA[Rank[j] − 1] + ℓ} < |y| and
                y[j + ℓ] = y[SA[Rank[j] − 1] + ℓ] do
 8              ℓ ← ℓ + 1
 9      else ℓ ← 0
10      LCP[Rank[j]] ← ℓ
11  LCP[|y|] ← 0
12  return LCP
```

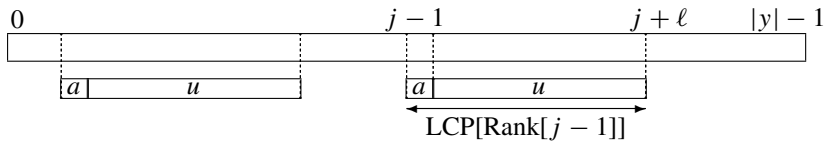
Note the solution is counterintuitive, since it looks natural to compute the values LCP[r] sequentially, that is, by processing suffixes in the increasing order of their ranks. But this does not readily produce a linear-time algorithm. Instead, Algorithm LCP processes the suffixes from the longest to the shortest, which is its key feature and is more efficient.

Solution

The correctness of the algorithm relies on the inequality

$$\text{LCP[Rank}[j-1]]-1 \le \text{LCP[Rank}[j]]$$

illustrated by the picture below.



Assume $\ell = \text{LCP[Rank}[j-1]]$ has just been computed and the longest common prefix associated with position $j-1$ is au for a letter a and a word u, that is, $\text{LCP[Rank}[j-1]] = |au|$. Then the longest common prefix associated with position j cannot be shorter than u. Therefore comparisons to compute LCP[Rank[j]] by extending u can start at position $j+\ell$. This is what the algorithm does at lines 7–8. Line 5 rules out the case when the longest common prefix is empty.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

As written the computation requires the table Rank, inverse of the table SA, which is computed at lines 1–2. It is used to retrieve the suffix immediately before the suffix $y[j\,..\,|y|-1]$ in the sorted list of all suffixes.

As for the running time of the procedure, it mostly depends on the number of tests at line 7. If the letters match, the value of $j+\ell$ increases and never decreases later. So, there are no more than $|y|$ such cases. There is at most one mismatch for each value of the variable j, then again no more than $|y|$ such cases. This proves the algorithm runs in linear time and executes no more than $2|y|$ letter comparisons.

Notes

The solution presented here is by Kasai et al. [155]. See also [74], where it is shown how to compute table SA in linear time on a linear-sortable alphabet.

47 Suffix Array to Suffix Tree

The goal of the problem is to transform the Suffix array of a word x into its Suffix tree. Despite the fact that both data structures infer essentially the same types of indexing operations, some come more readily from the Suffix tree structure.

The interest in designing a linear-time algorithm to do it is interesting when the alphabet is linearly sortable. Indeed, with this hypothesis, there are many linear-time algorithms to build the Suffix array of a word, although there is mostly one method to build its Suffix tree in the same time. Moreover, techniques used for the former construction are way easier to develop.

Here are tables SA and LCP of the Suffix array of `aacab`:

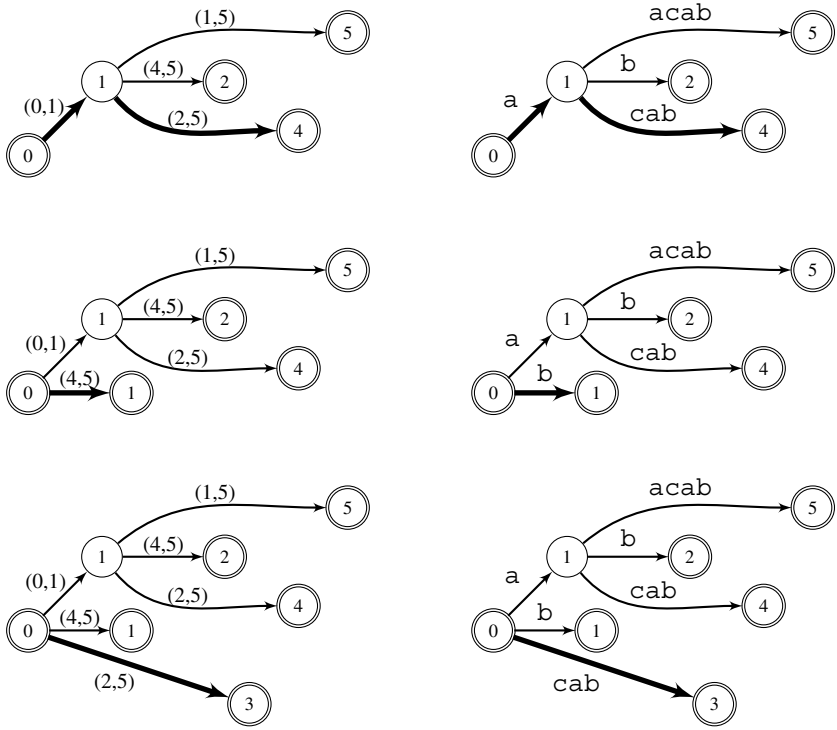
r	SA	LCP	
0	0	0	`aacab`
1	3	1	`ab`
2	1	1	`acab`
3	4	0	`b`
4	2	0	`cab`

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Table SA stores the starting position of non-empty suffixes according to their rank r in lexicographic order. Suffixes themselves are not part of the structure. Table LCP[r] gives the longest common prefix between rank-r and rank-$(r-1)$ suffixes.

Question. Show how to build the Suffix tree of a word in linear time given its Suffix array.

The pictures below illustrate three first steps of a possible Suffix tree construction for the example `aacab`. The first picture is when suffixes `aacab`, `ab` and `acab` have been treated. Labels of nodes are their word depth and labels of arcs are in the form (i, j) (on the left) representing factors $x[i \,..\, j-1]$ (on the right) of the word x. Doubly circled nodes are terminal states and thick paths show last inserted suffixes.



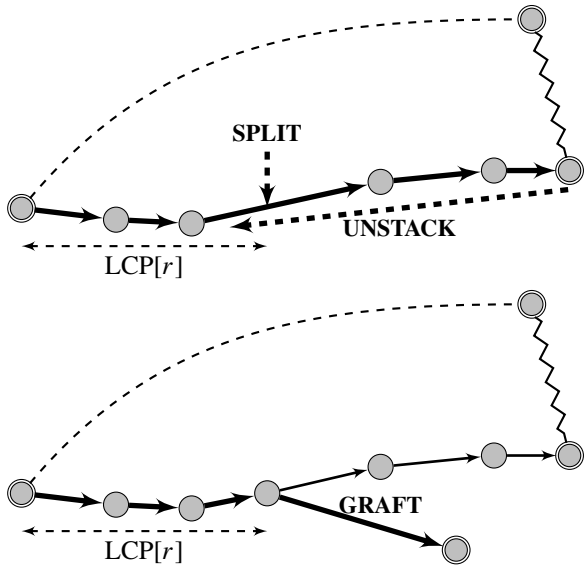
https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Solution

```
SArray2STree(Suffix array of a non-empty word x)
 1  ▷ (SA, LCP) Suffix array of x
 2  (q, d[q]) ← (New-Terminal-State(), 0)
 3  Initial ← q
 4  S ← ∅
 5  Push(S, (q, 0, 0, q))
 6  for r ← 0 to |x| − 1 do
 7      do (p, i, j, q) ← Pop(S)
 8      while LCP[r] < d[p]
 9      if LCP[r] = d[q] then
10          Push(S, (p, i, j, q))
11          s ← q
12      elseif LCP[r] = d[p] then
13          s ← q
14      else (s, d[s]) ← (New-State(), LCP[r])
15          Split(p, i, i + LCP[r] − d[p], s, i + LCP[r] − d[p], j, q)
16          Push(S, (p, i, i + LCP[r] − d[p], s))
17      (t, d[t]) ← (New-Terminal-State(), |x| − SA[r])
18      (s, SA[r] + LCP[r], |x|, t) ← New-Arc()
19      Push(S, (s, SA[r] + LCP[r], |x|, t))
20  return (Initial, nodes and arcs)
```

Algorithm SArray2STree processes the suffixes of the underlying word in alphabetic order, that is, according to table SA, and inserts them in the tree. Recall that arcs are labelled as explained above to ensure the linear space of the whole structure. Table LCP is used in conjunction with the depth of nodes $d[\,]$ (displayed on nodes in the above pictures). At a given step a stack S stores the arcs along the path associated with the last inserted suffix (thick path in pictures). The operation Split (line 15) inserts a node s at the middle of an arc and re-labels the resulting arcs accordingly.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press



Instructions in the for loop, illustrated by the above pictures, consist of three main steps: **UNSTACK**, an optional **SPLIT** and **GRAFT**. Step **UNSTACK** is realised by the while loop at lines 7–8. Then, the found arc is split at lines 14–15 if necessary, that is, if the split operation has to be done in the middle of the arc, not at one extremity. Eventually a new arc is grafted at lines 17–18. Meanwhile new arcs along the path labelled by the current suffix are pushed on the stack.

The correctness of the algorithm can be elaborated from the given indications. For the running time, mostly the analysis of the while loop, the value relies on the time to traverse the tree, which is realised with the help of the stack. Since the size of the tree is linear according to the word length the algorithm runs in linear time. Note there is no condition on the word alphabet.

Notes

The first algorithm to build a Suffix tree in linear time on a linearly sortable alphabet was developed by Farach [110]. The present algorithm provides another solution from any Suffix array construction having the same characteristics. The historically first such construction was by Kärkkäinen and Sanders [153, 154] (see [74]), then by Ko and Aluru [163] and by Kim et al. [159], followed by several others.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

48 Linear Suffix Trie

The Suffix trie of a word can be of quadratic size according to the word length. On the contrary, its Suffix tree requires only a linear amount of space for its storage, but the space should include the word itself.

The goal is to design a Suffix trie with edges labelled by single letters and that can be stored in linear space without the word itself. This is done by adding extra nodes and a few elements to the Suffix tree.

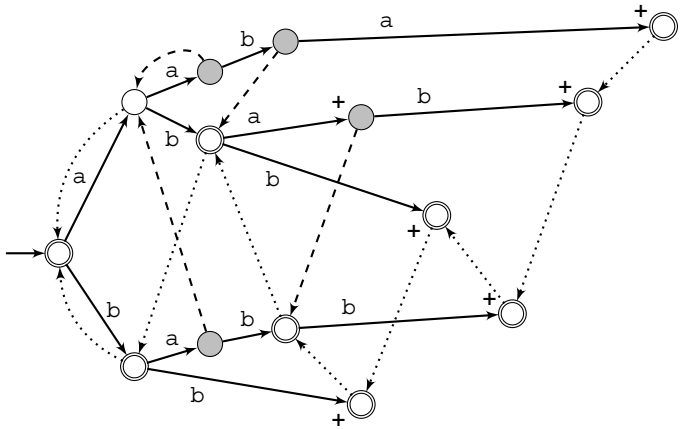
A node of the Suffix trie of y is identified to the factor of y that labels the path from the root to the node. Nodes in the linear Suffix trie $\mathcal{LST}(y)$ that are not in the Suffix tree $\mathcal{ST}(y)$ are of the form au, where a is a letter and u is a node of $\mathcal{ST}(y)$. That is, denoting by s the suffix link of the tree, $s(au) = u$. When nodes are added to $\mathcal{ST}(y)$ to create $\mathcal{LST}(y)$ edges are relabelled accordingly.

Question. Show the number of extra nodes added to the Suffix tree of a word y to create its linear Suffix trie is less than $|y|$.

Labels of edges in $\mathcal{LST}(y)$ are reduced to the first letter of the corresponding factor as follows. If v, $|v| > 1$, labels the edge from u to uv in $\mathcal{ST}(y)$, the label of the associated edge in $\mathcal{LST}(y)$ is the first letter of v and the node uv is marked with the $+$ sign to indicate the actual label is longer.

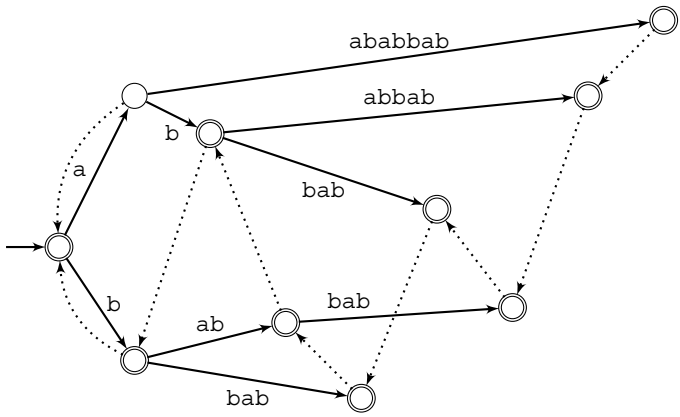
Question. Design an algorithm that checks if x occurs in y using the linear Suffix trie $\mathcal{LST}(y)$ and runs in time $O(|x|)$ on a fixed alphabet.

[**Hint:** Edge labels can be recovered using suffix links.]



https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

The above picture illustrates the linear Suffix trie of aababbab. White-coloured nodes are those of its Suffix tree (below with explicit edge labels), doubly circled when they are suffixes. Dotted edges form the suffix links of the Suffix tree. Grey-coloured nodes are the extra nodes with the dashed edges for the suffix links from them.



Solution

Few extra nodes. To answer the first question let u be a node of $\mathcal{LST}(y)$ that is not in $\mathcal{ST}(y)$. By definition $s(u)$ is a node of $\mathcal{ST}(y)$. Any proper suffix of u is of the form $s^k(u)$, which means it is also a node of $\mathcal{ST}(y)$. Therefore, two distinct nodes like u cannot share the same right position and there are no more than $|y|$ such nodes.

Note that a word whose letters are pairwise distinct has exactly $|y|-1$ extra nodes. If a letter has two occurrences (two distinct right positions) at least, it is a node of $\mathcal{ST}(y)$, then there are no more than $|y|-2$ extra nodes. Overall the number of extra nodes is less than $|y|$.

On the example of $\mathcal{LST}$(aababbab), the right positions of added nodes (grey-coloured in the picture) are 1 for aa, 2 for aab, 4 for abab, 3 and 6 for ba.

Searching $\mathcal{LST}(y)$. Checking if x is a factor of y is done by calling SEARCH($root, x$), where *root* is the root of $\mathcal{LST}(y)$. The main point, dealing with uncomplete edges, that is, edges whose target has a $+$ sign, relies mostly on the following observation.

Observation. Let au, a a letter, be a node of $\mathcal{LST}(y)$. If auv is also a node then uv is as well. This means that if v can be read from au in the tree, it can also be read from $s(au) = u$. (The converse does not hold.)

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

This leads to the sketch of Algorithm SEARCH that returns TRUE if ux is a factor of y.

```
SEARCH(u node of LST(y), x a word)
 1  if x = ε then
 2      return TRUE
 3  elseif no label of edges from u is x[0] then
 4      return FALSE
 5  else let (u, uv) be the edge whose label v is x[0]
 6      if uv has no + sign then
 7          return SEARCH(uv, x[1..|x| − 1])
 8      elseif SEARCH(s(u), v) then
 9          return SEARCH(uv, v^{-1}x)
10      else return FALSE
```

A straight implementation of the above scheme may not run in linear time due to non-explicit labels of some edges. To cope with it another suffix link, denoted by $\bar{s}$, is used.

First note that for any edge (u, uv) of $\mathcal{LST}(y)$ the pair $(s^k(u), s^k(uv))$ is defined for $0 \leq k \leq |u|$ but nodes $s^k(u)$ and $s^k(uv)$ may not be connected by a single edge. The suffix link $\bar{s}$ is defined on edges of $\mathcal{LST}(y)$ corresponding to edges of the Suffix tree having a label longer than a unique letter. If (u, uv) if such an edge of $\mathcal{LST}(y)$, that is, $|v| > 1$, $\bar{s}(u, uv) = (s^k(u), s^k(uv))$, where k is the smallest integer for which nodes $s^k(u)$ and $s^k(uv)$ are not connected by an edge. This definition is valid because all words of length 1 are nodes of $\mathcal{LST}(y)$ (not necessarily of $\mathcal{ST}(y)$). Note $\bar{s}$ can be computed in time proportional to the number of edges of $\mathcal{LST}(y)$.

Using $\bar{s}$ the implementation runs in linear time. Indeed, each time $\bar{s}(u, uv)$ is used to find the explicit label v of the edge, a letter of v is recovered. Then it cannot be used more than $|v|$ times, which yields a linear amortised running time. On a general alphabet A the implementation runs in time $O(|x| \log |A|)$.

Notes

The linear Suffix trie of a word and the associated searching techniques are described in [71]. The linear Suffix trie can be built by a mere post-processing of the Suffix tree of the word.

Hendrian et al. designed a right-to-left online construction of $\mathcal{LST}(y)$ running in time $O(|y| \log |A|)$ in [140]. They also produced a left-to-right online construction running in time $O(|y|(\log |A| + \log |y| / \log \log |y|))$.

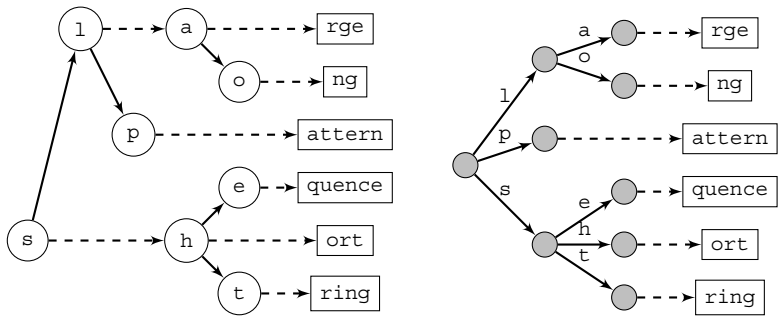
https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

49 Ternary Search Trie

Ternary search tries provide an efficient data structure to store and search a set of words. It figures a clever implementation of the trie of the set in the same way as the Suffix array does for the set of suffixes of a word.

Searching a trie for a pattern starts at the initial state (the root) and proceeds down following the matching arcs until the end of the pattern is met or until no arc matches the current letter. When the alphabet is large, representing arcs outgoing a state can lead to either a waste of space because many arcs have no target, or to a waste of time if linear lists are used. The goal of ternary search tries is to represent them by binary search trees on the outgoing letters.

To do so, each node of the trie has three outgoing arcs: left and right (up and down on the picture) for the binary search tree at the current trie node, and a middle arc to the next trie node. Below are the ternary search trie (left) and the trie (right) of the set
{`large`, `long`, `pattern`, `sequence`, `short`, `string`}.



Question. Describe the data structure to implement a ternary search trie storing a set of n words and show how to search it for a word of length m. Analyse the running time.

[**Hint:** Note the analogy with searching a Suffix array.]

Notice on the above example that the binary search tree corresponding to the arcs outgoing the initial node of the trie has its root labelled `s` and not the middle letter `p`. Indeed, to make the search more efficient binary search trees are weight balanced. The weight corresponds to the number of elements in the subtree. This is why the `s` starting letter of the majority of words is chosen for the root of the binary search tree.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Solution

The data structure of a ternary search tree T is composed of nodes linked in a tree manner. Each node q stores three pointers to other nodes, denoted by $q.left$, $q.right$ and $q.mid$, which have the functions described above. Some nodes are terminal (no outgoing arc). Each node also stores in $q.val$ either a suffix of a word in T if q is terminal or a letter.

```
TST-SEARCH(T a TST and x a non-empty word)
 1  q ← initial node of T
 2  for i ← 0 to |x| − 1 do
 3      if q is a terminal node then
 4          if x[i . . |x| − 1] = q.val then
 5              return TRUE
 6          else return FALSE
 7      q ← BST-SEARCH((q, x[i]))
 8      if q undefined then
 9          return FALSE
10      q ← q.mid
11  return FALSE        ▷ x prefix of a word in T
```

The BST search at line 7 is done in the subtree rooted at q using only the pointers *left* and *right*, and the field *val* compared to $x[i]$.

Let $n > 0$ be the number of words stored in T. A rough worst-case analysis shows the running time is $O(|x| \log n)$. But the role of the TST search is analogous to the binary search in a Suffix array to locate the current letter $x[i]$, leading to a tighter $O(|x| + \log n)$ time. More accurately, each negative letter comparison done during the TST search reduces the interval of words to be searched, which gives $O(\log n)$ such comparisons. And each positive comparison ends instructions in the for loop, thus a total of $O(|x|)$ such comparisons. Then overall there are $O(|x| + \log n)$ comparisons, including those at line 4, which is representative of the running time.

Notes

The notion of a ternary search trie is by Bentley and Sedgewick [31]. Clément et al. [57] give a thorough analysis of the structure according to several probabilistic conditions.

Applied to the suffixes of a word, the ternary search trie is the data structure that corresponds to algorithms associated with the Suffix array of the word.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

50 Longest Common Factor of Two Words

The problem deals with common factors of two words. It serves as a basis to compare texts and extends to applications such as bio-sequence alignment or plagiarism detection.

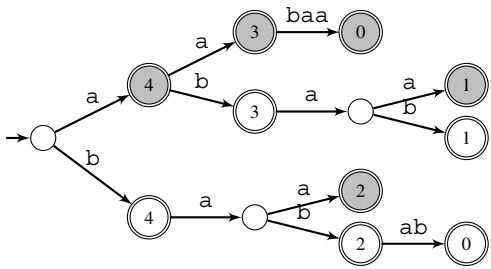
Let $LCF(x, y)$ denote the maximal length of factors that appear in two given words x and y drawn from the alphabet A. A straightforward solution to compute it is to build the common Suffix tree of x and y. Nodes are prefixes of their suffixes. A deepest node whose subtree contains both suffixes of x and suffixes of y gives the answer, its depth. This can also be done with the Suffix tree of $x\#y$, where # is a letter that does not appear in x nor in y.

The time to compute the tree is $O(|xy| \log |A|)$, or $O(|xy|)$ on linearly sortable alphabets (see Problem 47), and the required space is $O(|xy|)$.

Below is the common Suffix tree of $x = \texttt{aabaa}$ and $y = \texttt{babab}$. Grey (resp. white) doubly circled nodes are non-empty suffixes of x (resp. y). The node `aba` gives $LCF(\texttt{aabaa}, \texttt{babab}) = |\texttt{aba}| = 3$.

i	0	1	2	3	4
$x[i]$	a	a	b	a	a

j	0	1	2	3	4
$y[j]$	b	a	b	a	b



The goal of the problem is to reduce the size of the data structure to that of only one word, contrary to the above solution.

Question. Design an algorithm to compute $LCF(x, y)$ using the Suffix automaton (or the Suffix tree) of only one word. Analyse the time and space complexity of the whole computation.

[**Hint:** Use the indexing structure as a search machine.]

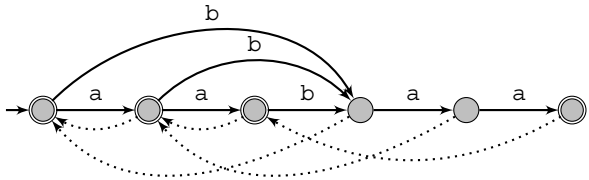
Solution

We assume $|x| \leq |y|$ and consider the Suffix automaton $\mathcal{S}(x)$ of x. Its size is known to be $O(|x|)$ independently of the alphabet. In addition to its states and

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

labelled arcs, the automaton is equipped with two functions defined on states: failure link *fail* and maximal depth L. For a state q associated with a non-empty word v (i.e., $q = goto(\text{initial}, v)$), $fail[v]$ is the state $p \neq q$ associated with the longest possible suffix u of v. And $L[q]$ is the maximal length of words associated with q.

Below is the Suffix automaton of the example word `aabaa` with the failure links (dotted arcs) on its states.



Algorithm LCF solves the question by using $\mathcal{S}(x)$ as a search engine.

```
LCF(S(x) Suffix automaton of x, y non-empty word)
 1  (m, ℓ, q) ← (0, 0, initial state of S(x))
 2  for j ← 0 to |y| − 1 do
 3      if goto(q, y[j]) defined then
 4          (ℓ, q) ← (ℓ + 1, goto(q, y[j]))
 5      else do  q ← fail[q]
 6          while q defined and goto(q, y[j]) undefined
 7          if q defined then
 8              (ℓ, q) ← (L[q] + 1, goto(q, y[j]))
 9          else (ℓ, q) ← (0, initial state of S(x))
10      m ← max{m, ℓ}
11  return m
```

At each step, the algorithm computes the length ℓ of the longest match between a factor of x and a suffix of $y[0\,..\,j]$. To do so, it proceeds like string-matching algorithms based on the use of a failure link. The only details specific to the algorithm is the faculty to reset properly the length ℓ to $L[q] + 1$ after following a series of links (see notes).

As for the whole running time, it is linear on a linearly sorted alphabet. Indeed, building the Suffix automaton of x can be done in linear time; and the above algorithm also runs in the same time because any computation of $goto(q, y[j])$ leads to an increase of either the variable j or the expression $j - \ell$, quantities that vary from 0 to $|y|$.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Note that, in fact, the algorithm finds the longest factor of x that ends at any position on y.

Notes

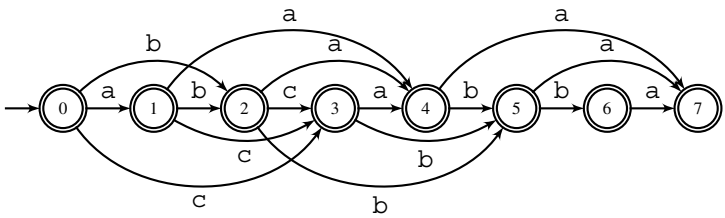
The method developed in the problem is by Crochemore [68] (see also [74, Chapter 6]. A similar method using a Suffix tree is by Hartman and Rodeh [138]. The technique adapts to locate a conjugate of x inside y with the Suffix automaton of xx.

51 Subsequence Automaton

Subsequences (or subwords) occurring in a word are useful elements to filter series of texts or to compare them. The basic data structure for developing applications related to subsequences is an automaton accepting them due to its reasonable size.

For a non-empty word y, let $\mathcal{SM}(y)$ be the minimal (deterministic) automaton accepting the subsequences of y. It is also called the Deterministic Acyclic Subsequence Graph (DASG). Below is the subsequence automaton of `abcabba`. All its states are terminal and it accepts the set

$$\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{aa}, \texttt{ab}, \texttt{ac}, \texttt{ba}, \texttt{bb}, \texttt{bc}, \texttt{ca}, \texttt{cb}, \texttt{aaa}, \texttt{aab}, \texttt{aba}, \texttt{abb}, \texttt{abc}, \ldots\}.$$



Question. Show how to build the subsequence automaton of a word and analyse the time and space complexity of the construction according to the size of the automaton.

Subsequence automata are an essential structure to find words that discriminate two words because they provide a direct access to all their subsequences.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

A word u distinguishes two distinct words y and z if it is a subsequence of only one of them, that is, if it is in the symmetric difference of their associated sets of subsequences.

Question. Show how to compute a shortest subsequence distinguishing two different words y and z with the help of their automata $\mathcal{SM}(y)$ and $\mathcal{SM}(z)$.

To design the algorithm it is interesting to note the following property of the subsequence automaton of y: if there is a path from state 0 to state i labelled by the word u, then $y[0\,..\,i-1]$ is the shortest prefix of y containing u as a subsequence.

Solution

Subsequence automaton construction. States of automaton $\mathcal{SM}(y)$ are $0, 1, \ldots, |y|$ and its transition table is goto. Let us assume that the alphabet of the word y is fixed, of size σ, and that it indexes a table t storing states. Algorithm DASG below processes y online. When its non-empty prefix w has just been processed, $t[a]-1$ is the rightmost position on w of letter a. Equivalently, it is also the rightmost state target of an arc labelled by letter a.

```
DASG(y)
1  for each letter a ∈ alph(y) do
2      t[a] ← 0
3  for i ← 0 to |y| − 1 do
4      for j ← t[y[i]] to i do
5          goto(j, y[i]) ← i + 1
6      t[y[i]] ← i + 1
```

Since the automaton is deterministic, its number of arcs is less than $\sigma|y|$. In fact it is no more than $\sigma|y| - \sigma(\sigma-1)/2$. Therefore the instruction at line 5 is executed less than $\sigma|y|$ times, which shows that the running time is $O(\sigma|y|)$. The extra space used by table t is $O(\sigma)$.

If the alphabet is not fixed, letters occurring in y can be first sorted in $O(alph(y) \log alph(y))$ time to get the above hypothesis. This adds to the total running time.

Distinguishing words. To find the shortest subsequence distinguishing two different words, one can use the general algorithm to test the equivalence of two deterministic finite automata. The algorithm is a standard application of

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

the UNION-FIND data structure and runs in time $O(n \log^* n)$, where n is the smaller length of the two words.

Notes

The notion of a subsequence automaton was first introduced by Baeza-Yates [21] and later on called a DASG by Troníček and Melichar [231]. Baeza-Yates's construction processes the word from right to left contrary to the above algorithm. The extension of the automaton to a finite set of words can be found in [21, 100]. The size of a DASG is analysed in [232].

Testing the equivalence of deterministic automata is by Hopcroft and Karp (1971), see [4], as an application of the UNION-FIND data structure. Another description and analysis of the structure appears in [63].

52 Codicity Test

Sets of words, especially binary words, are used to encode information. They may be related to transmission protocols, to data compression or mere texts. Streams of data need to be parsed according to the set to retrieve the original information. Parsing is a simple operation when codewords have the same length, like ASCII and UTF-32 codes for characters, and gives a unique factorisation of encoded data.

A code is a set of words that features a uniquely decipherable property. The question of having a unique parsing concerns mostly variable-length codes. The goal of the problem is to test whether a set of words is a code.

More precisely, a set $C = \{w_1, w_2, \ldots, w_n\}$ of words drawn from an alphabet A is a code if for every two sequences (noted as words) $i_1 i_2 \ldots i_k$ and $j_1 j_2 \ldots j_\ell$ of indices from $\{1, 2, \ldots, n\}$ we have

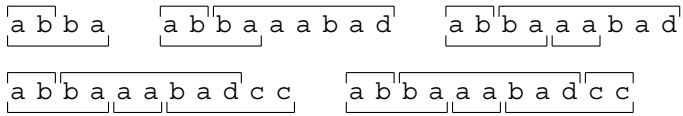
$$i_1 i_2 \ldots i_k \neq j_1 j_2 \ldots j_\ell \;\Rightarrow\; w_{i_1} w_{i_2} \ldots w_{i_k} \neq w_{j_1} w_{j_2} \ldots w_{j_\ell}.$$

In other words, if we define the morphism h from $\{1, 2, \ldots, n\}^*$ to A^* by $h(i) = w_i$, for $i \in \{1, 2, \ldots .n\}$, the condition means h is injective.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

The set $C_0 = \{\mathtt{ab}, \mathtt{abba}, \mathtt{baccab}, \mathtt{cc}\}$ is not a code because the word $\mathtt{abbaccab} \in C_0^*$ has two factorisations, $\mathtt{ab} \cdot \mathtt{baccab}$ and $\mathtt{abba} \cdot \mathtt{cc} \cdot \mathtt{ab}$, on the words of C_0. On the contrary, the set $C_1 = \{\mathtt{ab}, \mathtt{bacc}, \mathtt{cc}\}$ is a code because a word in C_1^* can start by only one word in C_1. It is said to be a prefix code (no $u \in C$ is a proper prefix of $v \in C$).

To test if $C_2 = \{\mathtt{ab}, \mathtt{abba}, \mathtt{baaabad}, \mathtt{aa}, \mathtt{badcc}, \mathtt{cc}, \mathtt{dccbad}, \mathtt{badba}\}$ is a code we can try to build a word in C_2^* with a double factorisation. Here is a sequence of attempts:



At each step we get a remainder, namely $\mathtt{ba}$, $\mathtt{aabad}$, $\mathtt{bad}$ and $\mathtt{cc}$, that we try to eliminate. Eventually we get a double factorisation because the last remainder is the empty word. Then C_2 is not a code.

The size N of the codicity testing problem for a finite set of words is the total length $||C||$ of all words of C.

Question. Design an algorithm that checks if a finite set C of words is a code and that runs in time $O(N^2)$.

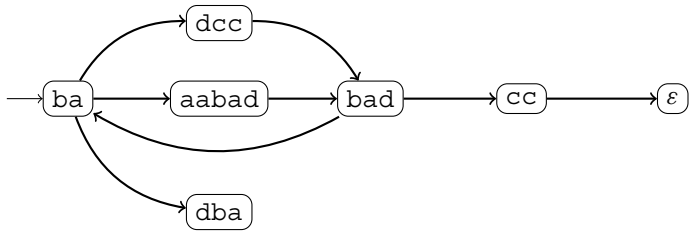
Solution

To solve the question, testing the codicity of C is transformed into a problem on a graph $G(C)$. Nodes of $G(C)$ are the remainders of attempts at a double factorisation, and as such are suffixes of words in C (including the empty word).

Nodes of $G(C)$ are defined in a width-first manner. Initial nodes at level 0 are those of the form $u^{-1}v$ for two distinct words $u, v \in C$. Their set may be empty if C is a prefix code. Then nodes at level $k+1$ are words of $C^{-1}D_k \cup D_k^{-1}C$, where D_k are nodes at level k. The set of nodes includes the empty word called the sink. There is an edge in $G(C)$ from u to v when $v = z^{-1}u$ or when $v = u^{-1}z$, for $z \in C$.

The picture below shows the graph $G(C_2)$ in which there is only one initial node and where columns correspond to node levels. The set C_2 is not a code because there is a path from the initial node to the sink. The middle such path corresponds to the above double factorisation. In fact, there is an infinity of words with a double factorisation due to the loop in the graph.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press



Observation. The set C is a code if and only if there is no path in $G(C)$ from an initial node to the sink.

The size of graph $G(C)$ is $O(N^2)$, since nodes are suffixes of words in C. Therefore the observation leads to an effective test of codicity. And since building the graph and exploring it can be done in time proportional to the size of the graph the solution runs in $O(N^2)$ time.

Notes

The algorithm to test the codicity of a finite set of words has been invented by Sardinas and Paterson [217]. A formal proof of the observation appears in [175, chapter 1] and in [36, chapter 1].

The algorithm can be implemented with the trie of the set, equipped with appropriate links, to obtain a $O(nN)$ running time, where n is the maximal length of words; see [15].

53 LPF Table

The problem deals with yet another table on words called abusively the ***longest previous factor*** table. It is a useful tool to factorise words for data compression (see Problem 97) and more generally to design efficient algorithms for finding repeats in texts.

For a non-empty word y, its table LPF stores lengths of repeating factors. More precisely, for a position j on y, LPF[j] is the maximal length of factors that starts both at position j and at a previous (i.e., smaller) position. Here is the table for `abaababababbabbb`.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$y[j]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b
LPF$[j]$	0	0	1	3	2	4	3	2	1	4	3	2	2	1

The next algorithm computes the table LPF for its input word y. It utilises the Suffix array of y and the table Rank that gives ranks of its suffixes in lexicographic order. Tables *prev* and *next* are links for a list representation of suffix ranks.

```
LPF(y non-empty word)
 1  for r ← 0 to |y| − 1 do
 2      (prev[r], next[r]) ← (r − 1, r + 1)
 3  for j ← |y| − 1 downto 0 do
 4      r ← Rank[j]
 5      LPF[j] ← max{LCP[r], LCP[next[r]]}
 6      LCP[next[r]] ← min{LCP[r], LCP[next[r]]}
 7      if prev[r] ≥ 0 then
 8          next[prev[r]] ← next[r]
 9      if next[r] < |y| then
10          prev[next[r]] ← prev[r]
11  return LPF
```

Question. Show that Algorithm LPF correctly computes the table LPF and works in linear time.

Looking accurately at the algorithm proves more than what it is designed for: lengths in LPF form a permutation of lengths in LCP.

Question. Show both that values in the LPF table are permuted from values in the LCP table and that the LCP table can be transformed into the LPF table.

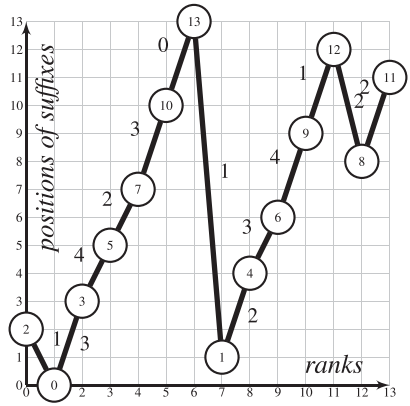
Solution

The analysis of Algorithm LPF becomes obvious when the Suffix array of its input is displayed graphically. The Suffix array of `abaababababbabbb` and the ranks of its suffixes are as follows.

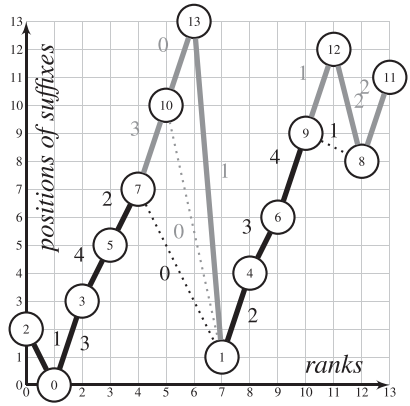
https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
$y[j]$	a	b	a	a	b	a	b	a	b	b	a	b	b	b	
Rank$[j]$	1	7	0	2	8	3	9	4	12	10	5	13	11	6	
r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA$[r]$	2	0	3	5	7	10	13	1	4	6	9	12	8	11	
LCP$[r]$	0	1	3	4	2	3	0	1	2	3	4	1	2	2	0

The display (below top) shows a graphic representation of the Suffix array of the above word. Positions are displayed according to their ranks (x-axis) and of their values (y-axis). The link between positions at ranks $r-1$ and r is labelled by LCP$[r]$.



Observation. The LCP length between the position at rank $r-1$ (resp. r) and any position of higher (resp. smaller) rank is not larger than LCP$[r]$.



A straight consequence of the observation gives the clue of the technique. When the position j at rank r occurs at a peak on the graphic, its associated

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

LPF length is the larger value of LCP[r] and LCP[$r+1$]. And the LCP length between its previous and next positions is the smaller of the two values. This is exactly the role of comparisons at lines 5–6.

It also explains why positions are treated from the largest to the smallest because then each position appears at a graphic peak in turn.

Next instructions of LPF manage the list of positions like a doubly linked list thanks to *prev* and *next*. The role of instructions at lines 8 and 10 is to remove the position j, of rank r, from the list.

The picture (above bottom) illustrates the situation just after positions 13 to 10 (in grey) have been treated. Dotted links are still labelled by LCP values.

This shows Algorithm LPF correctly computes the sought LPF table.

Solution to the second question. The above argument also shows that the values in the LPF table are permuted values of those in the LCP table of the Suffix array of y.

To transform the LCP table into the LPF table of the input, lines 5–6 of Algorithm LPF are changed to:

```
5  if LCP[r] < LCP[next[r]] then
6      (ℓ, LCP[r], LCP[next[r]]) ← (LCP[r], LCP[next[r]], ℓ)
```

where line 6 exchanges two values of the table. The algorithm produces the table LCP′ corresponding to LPF, since LPF[SA[r]] = LCP′[r], or equivalently LPF[j] = LCP′[Rank[j]]. Sorting pairs (SA[r], LCP′[r]) according to their first component produces values of the table LPF as their second component.

r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA[r]	2	0	3	5	7	10	13	1	4	6	9	12	8	11	
LCP′[r]	1	0	3	4	2	3	1	0	2	3	4	2	1	2	0

In the example, the algorithm produces the above table LCP′ from which we deduce for example LPF[2] = 1 (corresponding to the second occurrence of a) because 2 and 1 are aligned at rank $r = 0$.

Notes

The first linear-time algorithm for computing the LPF table of a word from its Suffix array appears in [76]. More efficient algorithms are designed in [78], where it is shown the computation can be done time–space optimally with an algorithm that runs in linear time with only $O(\sqrt{|y|})$ extra memory space used for a stack.

Three variants of the LPF table are presented in [87] with their corresponding construction algorithms; see also [50, 52, 99].

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

54 Sorting Suffixes of Thue–Morse Words

Thue–Morse words with their special structure provide examples in which some algorithms running in linear time or more can be optimised to run in logarithmic time instead. The problem shows such an example related to the Suffix array of words.

The infinite Thue–Morse word results from the iteration of the Thue–Morse morphism μ from $\{0,1\}^*$ to itself defined by

$$\begin{cases} \mu(0) = 01 \\ \mu(1) = 10 \end{cases}$$

The Thue–Morse word τ_n is $\mu^n(0)$, for a natural integer n. This type of description of Thue–Morse words is suitable to describe recursively the array SA_n that lists the starting positions of non-empty suffixes of τ_n sorted according to the lexicographic order of the suffixes. For example, $\tau_3 = 01101001$ and $\mathrm{SA}_3 = [5,6,3,0,7,4,2,1]$.

Question. Given integers n and k, $0 \leq k < n$, show how to compute $\mathrm{SA}_n[k]$ in time $O(n)$ for the word τ_n of length 2^n.

Solution

Let us start with two observations on word τ_n.

Observation 1. Let i be an even position on τ_n. Then $\tau_n[i] \neq \tau_n[i+1]$.

For $c \in \{0,1\}$, I^c_{odd}, resp. I^c_{even}, is the set of odd, resp. even, positions i for which $\tau_n[i] = c$. The justification of Observation 2, in which suf_i is $\tau_n[i\,..\,2^n-1]$, follows from Observation 1.

Observation 2.
(a) If $i \in I^0_{\mathrm{odd}}$, $j \in I^0_{\mathrm{even}}$ and $suf_i \neq 01$, then $suf_i < suf_j$.
(b) If $i \in I^1_{\mathrm{even}}$, $j \in I^1_{\mathrm{odd}}$ and $suf_j \neq 1$, then $suf_i < suf_j$.
An alternative formulation of Observation 2 is

$$I^0_{\mathrm{odd}} < I^0_{\mathrm{even}} < I^1_{\mathrm{even}} < I^1_{odd}.$$

For a sequence S of integers and two integers p and q, $p \cdot S$ and $S + q$ denote the sequences of elements of S multiplied by p and increased by q respectively. For example, $2 \cdot [1,2,3] = [2,4,6]$ and $[1,2,3] + 3 = [4,5,6]$.

The solution is split into two parts according to the parity of n.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

The case of even n. When n is an even integer the table SA_n is related to SA_{n-1} in the following way. Let α and β be the two halves of SA_{n-1} ($\mathrm{SA}_{n-1} = [\alpha, \beta]$); then

$$(*)\quad \mathrm{SA}_n = [2\cdot\beta + 1, 2\cdot\alpha, 2\cdot\beta, 2\cdot\alpha + 1].$$

Proof Let $sorted(X)$, for a set X of suffix starting positions on a word, denote the sorted list of positions according to the lexicographic order of the suffixes. Let also

$$\gamma_1 = sorted(I^{\mathtt{0}}_{\mathrm{odd}}),\quad \gamma_2 = sorted(I^{\mathtt{0}}_{\mathrm{even}}),$$

$$\gamma_3 = sorted(I^{\mathtt{1}}_{\mathrm{even}}),\quad \gamma_3 = sorted(I^{\mathtt{1}}_{\mathrm{odd}}).$$

Then, due to Observation 2, $\mathrm{SA}_n = [\gamma_1, \gamma_2, \gamma_3, \gamma_4]$.

Fortunately, for even n we do not have bad suffixes `01` nor `1` in τ_n. We can use the morphic representation of Thue–Morse words. First observe that the morphism μ preserves the lexicographic order ($u < v \Leftrightarrow \mu(u) < \mu(v)$). Each suffix at position i on τ_{n-1} is mapped by μ to a suffix at position $2i$ on τ_n. Hence $2\cdot\mathrm{SA}_{n-1} = [2\cdot\alpha, 2\cdot\beta]$ is the sequence of sorted suffixes at even positions in τ_n.

Then due to the previous observation $\mathrm{SA}_n = [\gamma_1, \gamma_2, \gamma_3, \gamma_4]$, where γ_1 corresponds to sorted suffixes starting at second positions of suffixes associated with $2\cdot\beta$. Similarly for γ_4 and $2\cdot\alpha$. Therefore we get $\mathrm{SA}_n = [2\cdot\beta + 1, 2\cdot\alpha, 2\cdot\beta, 2\cdot\alpha + 1]$, as required. ∎

Computing SA_4 from SA_3. $\mathrm{SA}_3 = [5,6,3,0,7,4,2,1]$ is composed of $\alpha = [5,6,3,0]$ and $\beta = [7,4,2,1]$. We have $2\cdot\beta = [14,8,4,2]$, $2\cdot\beta+1 = [15,9,5,3]$, $2\cdot\alpha = [10,12,6,0]$ and $2\cdot\alpha + 1 = [11,13,7,1]$, which gives

$$\mathrm{SA}_4 = [15,9,5,3,\ 10,12,6,0,\ 14,8,4,2,\ 11,13,7,1].$$

The case of odd n. When n is odd we can also apply the formula $(*)$ except that the *bad* suffixes `01` and `1` should be *specially* placed at their correct places: the suffix `1` should be placed in front of all other suffixes starting with `1`. The suffix `01` should be placed immediately after the whole sequence of suffixes starting with `00`. Hence the correction reduces to the computation of the number $p(n)$ of occurrences of `00` in τ_n.

The numbers $p(n)$ for $n = 2, 3, \ldots, 10$ are $0, 1, 2, 5, 10, 21, 42, 85, 170$. These numbers satisfy the recurrence

$$(**)\quad p(1) = 0,\ p(2k+1) = 4\cdot p(2k-1) + 1,\ p(2k+2) = 2\cdot p(2k+1).$$

Consequently $p(2k+1) = (4^k - 1)/3$.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Computing SA_5 from SA_4. To do it, first apply the transformation $(*)$ to get the four blocks:

$$29, 17, 9, 5, 23, 27, 15, 3 \qquad \mathbf{30}, 18, 10, 6, 20, 24, 12, 0,$$

$$28, 16, 8, 4, 22, 26, 14, 2 \qquad \mathbf{31}, 19, 11, 7, 21, 25, 13, 1.$$

The bad suffixes `01` and `1` start at positions 30, 31. The number 31 should be moved after the 5th element 23, since $p(5) = 5$. The number 31 corresponding to a one-letter suffix should be moved to the beginning of the third quarter (it is the smallest suffix starting with letter `1`). We get the final value of the suffix table SA_5 by concatenating:

$$29, 17, 9, 5, 23, \mathbf{30}, 27, 15 \qquad 3, 18, 10, 6, 20, 24, 12, 0,$$

$$\mathbf{31}, 28, 16, 8, 4, 22, 26, 14 \qquad 2, 19, 11, 7, 21, 25, 13, 1.$$

Conclusion. To answer the question, computing quickly $SA_n[k]$, let us summarise how it can be done:

- Identify in which quarter of SA_n the number k is located.
- Reduce the problem to the computation of $SA_{n-1}[j]$, where the corresponding position j (around half of k) is computed using the formula $(*)$ backwards.
- If n is odd, take into account the relocation of the two *bad* suffixes in SA_n. The value of $p(n)$ given by $(**)$ is used for the relocation.
- Iterate such reductions until coming to a constant-sized table.

Altogether $O(n)$ steps are sufficient to compute $SA_n[k]$, which is logarithmic with respect to the size of the table SA_n.

Notes

It seems there is a possible different approach using a compact factor automaton for Thue–Morse words, as described in [204]. However, this leads to an even more complicated solution.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

55 Bare Suffix Tree

Suffix trees provide a data structure for indexing texts. Optimal-time constructions of them suffer from a rather high memory requirement, larger than for Suffix arrays with the same usage. The problem deals with a moderately efficient but not completely naive and very simple construction of Suffix trees.

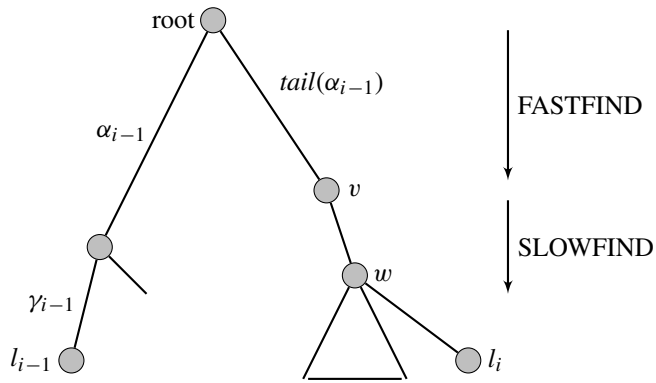
The Suffix tree T of a word x ending with a unique end marker is the compacted trie of suffixes of x. A leaf corresponds to a suffix and an internal node to a factor having at least two occurrences followed by different letters. Each edge is labelled by a factor $x[i\,..\,j]$ of x, represented by the pair (i, j). Its word-length is $|x[i\,..\,j]| = j - i + 1$. The word-length of a path in T is the sum of word-lengths of its edges, while the length of the path is its number of edges. Let $depth(T)$ be the maximum length of a path in T from the root to a leaf. Let l_i be the leaf ending the branch labelled by $x[i\,..\,n-1]$.

Question. Design a construction of the Suffix tree T of a word x using no additional array and running in time $O(|x|depth(T))$ on a fixed-size alphabet.

Solution

The main scheme of the solution is to insert iteratively the suffixes in the tree, from the longest to the shortest suffix of x.

Let T_{i-1} denote the compacted trie of suffixes starting at positions $0, 1, \ldots, i-1$ on x. We show how to update T_{i-1} to get the tree T_i.



The ith suffix can be split into $\alpha_i\gamma_i$ where $\alpha_i = x[i\,..\,i + d_i - 1])$ and $\gamma_i = x[i + d_i\,..\,n-1]$. The word α_i is the path-label from the root to $w = parent(l_i)$ (see picture). In particular $\alpha_i = \varepsilon$ if $parent(l_i) = root$.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

When a is the first letter of the word au, $tail(au) = u$. Note the word $\alpha_k \neq \varepsilon$ has an occurrence starting at k and at some smaller position. Consequently $tail(\alpha_k)$ has occurrences at $k+1$ and at a smaller position. This implies the following crucial fact.

Observation. Assume $\alpha_{i-1} \neq \varepsilon$. Then there is a path in T_{i-1} spelling the word $tail(\alpha_{i-1})\gamma_{i-1}$ (see picture). In other words, a great part of the suffix $x[i\,..\,n]$ that is being inserted is already present in the tree.

The algorithm uses two types of tree traversal:

- *FastFind*(α) assumes that α is present in the current tree (as a path-label). It finds the node v by spelling the word α. If the spelling ends in the middle of an edge-label, the node v is created.

  The traversal is guided by the length d of α. It uses the edges of the tree as *shortcuts*, reading only the first symbol and the length of each edge. The cost of the traversal is $O(depth(T))$.
- *SlowFind*(v, γ) finds the lowest descendant w of v in the current tree following the path labelled by the longest possible prefix β of γ. As above, node w may have to be created.

  The traversal goes symbol by symbol, updating the value of d. Its cost is $O(|\beta|)$.

The whole algorithm starts with the tree composed of a single edge labelled by the whole word $x[0\,..\,n-1]$ and executes the following scheme for suffixes at positions $i = 1, 2, \ldots, n-1$.

One iteration, from T_{i-1} to T_i:

if $\alpha_{i-1} = \varepsilon$ then $v \leftarrow root$ else $v \leftarrow$ *FastFind*$(tail(\alpha_{i-1}))$,
$w \leftarrow$ *SlowFind*(v, γ_{i-1}),
a new leaf l_i and new edge $w \rightarrow l_i$ are created.

Running time of the algorithm. There are $n-1$ iterations. In each iteration the cost of *FastFind* is $O(depth(T))$. The total cost of *SlowFind*s is $O(n)$, since each of their single moves decreases the length of the word γ_i. Altogether the time cost is $O(n \cdot depth(T))$.

Note the algorithm requires no additional array as required.

Notes

The algorithm described here is a simplified version of McCreight's Suffix tree construction [187] that runs on linear time but requires additional arrays to work. The present variant is slightly slower but significantly simpler than

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

McCreight's algorithm. It can be viewed as a first step towards understanding the complete algorithm. Moreover, for many types of words, the coefficient $depth(T)$ is logarithmic.

Ukkonen's algorithm [234] can be modified in a similar way, which gives another simple but not naive construction of Suffix trees.

56 Comparing Suffixes of a Fibonacci Word

The structure of Fibonacci words like that of Thue–Morse words, is an example of a situation in which some algorithms can be optimised to run in logarithmic time with respect to the length of words. The problem is concerned with a fast lexicographic comparison between two suffixes of a finite Fibonacci word, which shows such a phenomenon.

We deal with a slightly shortened version of Fibonacci words to simplify arguments. Let g_n be the nth Fibonacci word fib_n with the last two letters deleted, that is, $g_n = fib_n\{\mathtt{a},\mathtt{b}\}^{-2}$, for $n \geq 2$. Let also $suf(k,n)$ be the kth suffix $g_n[k\,..\,|g_n|-1]$ of g_n. For example, $g_2 = \mathtt{a}$, $g_3 = \mathtt{aba}$, $g_4 = \mathtt{abaaba}$, $g_5 = \mathtt{abaababaaba}$ and $suf(3,5) = \mathtt{ababaaba}$.

The comparison between suffixes of g_n is efficiently reduced to the comparison of their compact representation of logarithmic length implied by their logarithmic-size decomposition (property below). Factors of the decomposition are the reverse $R_n = fib_n^{\mathrm{R}}$ of Fibonacci words. The first factors are $R_0 = \mathtt{a}$, $R_1 = \mathtt{ba}$, $R_2 = \mathtt{aba}$ and $R_3 = \mathtt{baaba}$. Observe that $R_{n+2} = R_{n+1}R_n$ and R_n starts with the letter $\mathtt{a}$ if and only if n is even.

Property. For $n > 2$, $suf(k,n)$ uniquely factorises as $R_{i_0}R_{i_1}\ldots R_{i_m}$, where $i_0 \in \{0,1\}$ and $i_t \in \{i_{t-1}+1,\, i_{t-1}+2\}$ for $t = 1,\ldots,m$.

Related to the factorisation let $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$. For example, $suf(3,5) = \mathtt{ababaaba} = R_0\cdot R_1\cdot R_3 = \mathtt{a}\cdot\mathtt{ba}\cdot\mathtt{baaba}$ and $\mathcal{R}_5(3) = (0,1,3)$.

Question. **(A)** Show how to compare any two suffixes of $g_n = fib_n\{\mathtt{a},\mathtt{b}\}^{-2}$ in time $O((\log|fib_n|)^2)$.
(B) Improve the running time to $O(\log|fib_n|)$.

[**Hint:** For (A) use the algorithm of Problem 6.]

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Solution

Associated with $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$ let

$$\Psi_n(k) = first(R_{i_0})first(R_{i_1})\ldots first(R_{i_m}),$$

where $first(w)$ denotes the first letter of w. It can be verified that

Observation. $suf(p,n) < suf(q,n) \iff \Psi_n(p) < \Psi_n(q)$.

Example. For $g_5 = \texttt{abaababaaba}$, we have $suf(3,5) = \texttt{a} \cdot \texttt{ba} \cdot \texttt{baaba} = R_0R_1R_3$ and $suf(5,5) = \texttt{a} \cdot \texttt{ba} \cdot \texttt{aba} = R_0R_1R_2$. Then $\Psi_5(3) = \texttt{abb}$ and $\Psi_5(5) = \texttt{aba}$. Therefore $suf(5,5) < suf(3,5)$, since $\texttt{aba} < \texttt{abb}$.

The observation reduces the problem to the fast computation of the decomposition in the above property and of the function $\mathcal{R}$.

Point (A). $\mathcal{R}_n(k)$ can be computed as follows, scanning the suffix $suf(n,k)$ from left to right. If $g_n[k] = \texttt{a}$ we know that $i_0 = 0$; otherwise $i_0 = 1$. Then in each iteration $t > 0$ the current position on g_n is increased by the length $F_{i_{t-1}+2} = |R_{i_{t-1}}|$ to point to the next letter of g_n. Depending on this letter we know whether $i_t = i_{t-1} + 1$ or $i_t = i_{t-1} + 2$. In this way $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$ is computed and the process has a logarithmic number of iterations.

Accessing each letter in g_n is done in time $O(\log |fib_n|)$ using the algorithm of Problem 6.

Overall this gives an algorithm running in time $O((\log |fib_n|)^2)$ and solves (A).

Point (B). It does not come as a surprise that Fibonacci words are closely related to the Fibonacci numeration system (see Problem 6). Here we show they are related to a dual version of this system in the context of lexicographic sorting.

Lazy Fibonacci numeration system. Let $LazyFib(k)$ be the lazy Fibonacci representation of the natural number k, starting with least significant digits. In this system a natural number N is represented in a unique way as the sequence of bits $(b_0, b_1, b_2, \ldots)$ for which $N = \sum b_i F_{i+2}$, where F_js are consecutive Fibonacci numbers, and in which no two consecutive zeros appear. This corresponds to the condition $i_{t+1} \in \{i_t + 1, i_t + 2\}$ stated in the factorisation property.

For example, $LazyFib(9) = (\texttt{1 0 1 1})$ and $LazyFib(23) = (\texttt{0 1 1 1 0 1})$, since $9 = F_2 + F_4 + F_5$ and $23 = F_3 + F_4 + F_5 + F_7$.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Fast computation of the decomposition. Let $(b_0, b_1, b_2, \ldots)$ be the representation of the length $|suf(k,n)|$ in the lazy Fibonacci system. Then $\mathcal{R}_n(k) = (i_0, i_1, \ldots, i_m)$, where i_js are the positions of bit `1` in $(b_0, b_1, b_2, \ldots)$.

Since the lazy Fibonacci representation can be computed in logarithmic time with respect to the length of fib_n, this solves (B).

Notes

The proof of the factorisation property can be found in [213, 238].

If we want to compare two suffixes of length larger than 2 of standard (not shortened) Fibonacci words fib_n then the same function Ψ can be used if n is odd. But if n is even we have to replace $\Psi(k)$ by $\Psi(k) \cdot \mathtt{b}$. It is also known that for even n the table SA of the Suffix array of Fibonacci words contains an arithmetic progression (modulo the length of the array) and this gives an alternative comparison test for the case of an even n.

The lazy Fibonacci system allows computation in logarithmic time of the rank of the kth suffix (its position in SA) of a Fibonacci word.

57 Avoidability of Binary Words

Some patterns occur in all long enough words. They are said to be unavoidable. The notion obviously depends on the alphabet size and in the problem we consider binary patterns.

A word w is said to avoid a set X of words if no factor of w belongs to X. The set X is said to be avoidable if there is an infinite word avoiding it, or equivalently on a finite alphabet, if there are infinitely many words avoiding it. The goal is to test if a set of words drawn from the binary alphabet $\mathtt{B} = \{\mathtt{a}, \mathtt{b}\}$ is avoidable.

For example, $\{\mathtt{aa}, \mathtt{abab}, \mathtt{bb}\}$ cannot be avoided by a word of length at least 5. On the contrary, $\{\mathtt{aa}, \mathtt{bb}\}$ is avoided by the infinite word $(\mathtt{ab})^\infty$.

To design the test we define two reductions on a set $X \subseteq \mathtt{B}^+$.

reduce1 (remove super-word): If $x, y \in X$ and x is a factor of y remove y from X.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

reduce2 (chop last letter): If x is a suffix of $y \neq \varepsilon$ and $x\bar{a}, ya \in X$ substitute y for ya in X (the bar morphism exchanges a and b).

AVOIDABLE(X non-empty set of binary words)
1 **while** reduce1 or reduce2 are applicable to X **do**
2 $X \leftarrow$ reduce1(X) or reduce2(X)
3 **if** $X \neq \{\mathtt{a}, \mathtt{b}\}$ **return** TRUE **else return** FALSE

Example. Set $X = \{\mathtt{aaa}, \mathtt{aba}, \mathtt{bb}\}$ is unavoidable because the sequence of reductions yields B (changed words are underlined): $\{\mathtt{aaa}, \underline{\mathtt{aba}}, \mathtt{bb}\} \to \{\underline{\mathtt{aaa}}, \mathtt{ab}, \mathtt{bb}\} \to \{\mathtt{aa}, \underline{\mathtt{ab}}, \mathtt{bb}\} \to \{\underline{\mathtt{aa}}, \mathtt{a}, \mathtt{bb}\} \to \{\mathtt{a}, \underline{\mathtt{bb}}\} \to \mathtt{B}$.

Question. Show that a set X of binary words is avoidable if and only if AVOIDABLE(X) = TRUE.

Question. Show that a set $X \subseteq \mathtt{B}^{\leq n}$ is avoidable if and only if it is avoided by a word of length larger than $2^{n-1} + n - 2$ and that the bound is optimal.

Solution

Correctness of AVOIDABLE. It is a consequence of the following two properties.

Property 1. If $Y =$ reduce1(X) or $Y =$ reduce2(X), X is avoidable if and only if Y is avoidable.

Proof It is clear that a word that avoids Y also avoids X. To prove the converse, let w be an infinite word avoiding X. We show that w also avoids Y. This is obvious if $Y =$ reduce1(X). If $Y =$ reduce2(X), there are two words $x\bar{a}, ya \in X$, x a suffix of y and $Y = X \setminus \{ya\} \cup \{y\}$.

It is then enough to show that w avoids y. If not, yb is a factor of w. Letter b cannot be a because w avoids ya. But it cannot be $\bar{a}$ either because w avoids $x\bar{a}$ suffix of $y\bar{a}$. Then y is not a factor of w. ∎

Property 2. If no reduction is applicable to X and $X \neq \mathtt{B}$, X is avoidable.

Proof To show the conclusion we construct incrementally an infinite word w avoiding X. Let v be a finite word avoiding X (v can be just a letter because $X \neq \mathtt{B}$). We claim that v can be extended by a single letter to va that also avoids X. Indeed, if it cannot, there are two suffixes x and y of v for which

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

$x\bar{a} \in X$ and $ya \in X$. Since one of the words is a suffix of the other, reduction2 applies to X, in contradiction of the hypothesis. Hence v can be extended eventually to an infinite word w, avoiding X. ∎

Length bound on avoiding words. The next property is used to answer the second question.

Property 3. $X \subseteq \mathtt{B}^{\leq n}$ is avoidable if and only if it is avoided by a word having a border in $\mathtt{B}^{n-1}$.

In fact, if X is avoidable, an infinite word avoiding it has a factor satisfying the condition. Conversely, let $w = uv = v'u$, avoiding X with $u \in \mathtt{B}^{n-1}$. Since $uv^i = v'uv^{i-1} = \cdots = v'^i u$, $i > 0$, it is clear that any length-n factor of uv^i is a factor of w. Therefore uv^∞ also avoids X.

To answer the second question, just the converse of the if-and-only-if needs to be proved. If a word of length larger than $2^{n-1} + n - 2$ avoids X it contains at least two occurrences of some word in $\mathtt{B}^{n-1}$ and thus a factor as in property 3 that avoids X. Therefore X is avoidable.

The optimality of the bound relies on de Bruijn (binary) words of order $n - 1$. Such a word w contains exactly one occurrence of each word in $\mathtt{B}^{n-1}$ and has length $2^{n-1} + n - 2$. The word avoids the set X of length-n words that are not factors of it. This achieves the proof, since X is unavoidable.

Notes

Algorithm AVOIDABLE is certainly not the most efficient algorithm to test set avoidability in the binary case but it is probably the simplest one. References on the subject may be found in [175]. The solution of the second question is from [90].

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

58 Avoiding a Set of Words

For a finite set F of words on a finite alphabet A, $F \subseteq A^*$, let $L(F) \subseteq A^*$ be the language of words that avoids F; that is, no word in F appears in words of $L(F)$. The aim is to build an automaton accepting $L(F)$.

Note that if w avoids u it also avoids any word u is a factor of. Thus we can consider that F is an anti-factorial (factor code) language: no word in F is a proper factor of another word in F. On the contrary, $F(L)$ is a factorial language: any factor of a word in $F(L)$ is in $F(L)$.

Examples. For $F_0 = \{\mathtt{aa}, \mathtt{bb}\} \subseteq \{\mathtt{a}, \mathtt{b}\}^*$ we get $L(F_0) = (\mathtt{ab})^* \cup (\mathtt{ba})^*$. For $F_1 = \{\mathtt{aaa}, \mathtt{bbab}, \mathtt{bbb}\} \subseteq \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$ we have $(\mathtt{ab})^* \subseteq L(F_1)$ as well as $(\mathtt{bbaa})^* \subseteq L(F_1)$ and $\mathtt{c}^* \subseteq L(F_1)$.

Question. Show that $L(F)$ is recognised by a finite automaton and design an algorithm to build, from the trie of F, a deterministic automaton that accepts it.

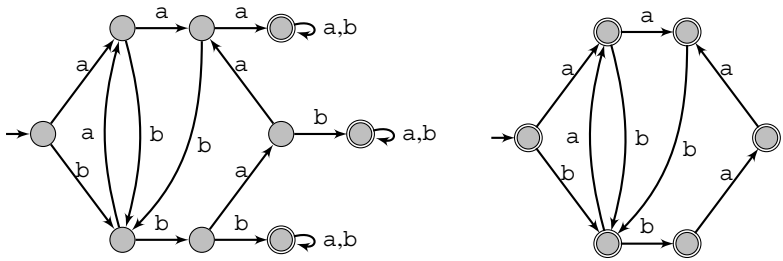
[**Hint:** Use the Aho–Corasick technique.]

The set F is said to be unavoidable if no infinite word avoids it (see Problem 57). For example, the set F_1 is avoidable on the alphabet $\{\mathtt{a}, \mathtt{b}\}$ because it is avoided by the infinite word $(\mathtt{ab})^\infty$.

Question. Show how to test if the set F is unavoidable.

Solution

Assume F is non-empty and anti-factorial (in particular it does not contain the empty word) and consider the automaton accepting A^*FA^*, words having a factor in F. States of the automaton are (or can be identified with) the prefixes of words in F. Indeed, any such prefix can be extended to produce a word of F and these latter words form sink states. Below left is an automaton accepting $\{\mathtt{a}, \mathtt{b}\}^* F_1 \{\mathtt{a}, \mathtt{b}\}^*$, in which doubly circled nodes correspond to words with a factor in F_1.



https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

The language $L(F) = A^* \setminus A^*FA^*$, the complement of A^*FA^*, is accepted by the automaton accepting A^*FA^* after exchanging the role of terminal and non-terminal nodes. This shows $L(F)$ is accepted by a finite automaton. Above right is an automaton accepting $L(F_1)$.

Algorithm AVOIDING follows closely the construction of the dictionary matching automaton of F and eventually reverses the status of states and deletes useless states.

```
AVOIDING(trie of F, alphabet A)
 1  q0 ← initial state (root) of the trie
 2  Q ← ∅          ▷ empty queue
 3  for each letter a ∈ A do
 4      if goto(q0, a) undefined then
 5          add arc (q0, a, q0)
 6      else append (goto(q0, a), q0) to Q
 7  while Q not empty do
 8      remove (p, r) from Q
 9      if r terminal then
10          set p a terminal state
11      for each letter a ∈ A do
12          s ← goto(r, a)
13          if goto(p, a) undefined then
14              add arc (p, a, s)
15          else append (goto(p, a), s) to Q
16  remove all terminal states and their associated arcs
17  set all remaining states as terminal states
18  return transformed automaton
```

The algorithm runs in time $O(|A| \sum\{|w| : w \in F\})$ with an appropriate implementation of the goto function. But instead of setting all possible arcs out of each state, a failure link can be created from state p to state r when (p, r) is in the queue. This is the usual technique to implement this type of automaton, reducing its size to $O(\sum\{|w| : w \in F\})$.

To test if F is unavoidable, it remains to check if the graph formed by nodes of the output automaton contains a cycle. This can be done in linear time with a topological sorting algorithm. The above right automaton contains a cycle, which shows again that F_1 is unavoidable.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Notes

The construction of a dictionary-matching automaton, also called an Aho–Corasick automaton is by Aho and Corasick [3] and described in most textbooks on text algorithms. The automaton is usually implemented with a notion of failure links to save space.

59 Minimal Unique Factors

The topic of this problem has the same flavour as the notion of minimal absent words. It can also be used to identify, to filter or to distinguish files. But the corresponding algorithms and the underlying combinatorial properties are simpler.

A ***minimal unique factor*** of a word x is a factor that occurs once in x and whose proper factors are repeats, that is, have at least two occurrences in x. A minimal unique factor $x[i\,..\,j]$ is stored in the table *MinUniq* by setting $MinUniq[j] = i$.

Example. Minimal unique factors of $x = \texttt{abaabba}$ are $\texttt{aba} = x[0\,..\,2]$, $\texttt{aa} = x[2\,..\,3]$ and $\texttt{bb} = x[4\,..\,5]$, and $MinUniq[2] = 0$, $MinUniq[3] = 2$ and $MinUniq[5] = 4$ (other values are set to -1).

	0	1	2	3	4	5	6
x	a	b	a	a	b	b	a

Algorithm MinUnique applies to a singleton-free word (each letter appear at least twice in it).

```
MinUnique(non-empty singleton-free word x)
 1  (SA, LCP) ← Suffix array of x
 2  for i ← 0 to |x| do
 3      MinUniq[i] ← −1
 4  for r ← 0 to |x| − 1 do
 5      ℓ ← max{LCP[r], LCP[r + 1]}
 6      MinUniq[SA[r] + ℓ] ← max{MinUniq[SA[r] + ℓ], SA[r]}
 7  return MinUniq[0 .. |x| − 1]
```

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Question. Show that the algorithm MinUnique computes the table *MinUniq* associated with its singleton-free input word x.

There is a duality between minimal unique factors and maximal occurrences of repeats in the word x.

Question. Show that a minimal unique factor induces two maximal occurrences of repeats in a singleton-free word.

Question. How many minimal unique factors are there in a de Bruijn word of order k?

Solution

Sketch of the proof of correctness. The notion of minimal unique factor of a word is close to the notion of identifier of a position on the word. The identifier of a position i on $x\#$ (# is an end-marker) is the shortest prefix of $x\#[i\,..\,|x|]$ that occurs exactly once in $x\#$. Then if the factor ua with letter a is the identifier of i, u occurs at least twice in x, corresponding to length ℓ computed at line 5 in MinUnique.

The algorithm implicitly uses identifiers because a minimal unique factor is the shortest identifier among all those ending at a given position, say j. This is done at line 6, where $j = \mathrm{SA}[r] + \ell$ and $MinUniq[j]$ is updated accordingly.

The computation of minimal unique factors of `abaabba` is illustrated below. The value $MinUniq[7] = 6$ is discarded when there is no end-marker. When $r = 3$, $MinUniq[5]$ is set to 3, which is eventually updated to 4 when $r = 6$. The three non-negative values, 0, 2 and 4, correspond to factors given before.

r	0	1	2	3	4	5	6	7
SA[r]	6	2	0	3	5	1	4	
LCP[r]	0	1	1	2	0	2	1	0
j	0	1	2	3	4	5	6	7
$MinUniq[j]$	−1	−1	−1 0	−1 2	−1	−1 3 4	−1	−1 6

Maximal repeats. A minimal unique factor in the singleton-free word x is of the form aub, for a word u and letters a, b, because it cannot be reduced to a

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

single letter. Then au and ub both occur at least twice in x, that is, are repeats. The occurrence of au (determined by the occurrence of aub) can be extended to the left to a maximal occurrence of a repeat. Similarly, the occurrence of ub can be extended to the right to a maximal occurrence of a repeat. This answers the second question.

De Bruijn words. In a de Bruijn word of order k on the alphabet A, each word of length k appears exactly once. Shorter words are repeats. Then there are exactly $|A|^k$ minimal unique factors in the de Bruijn word whose length is $|A|^k + k - 1$.

Notes

The elements in this problem are by Ilie and Smyth [148]. The computation of shortest unique factors is treated in [233]. The computation of minimal unique factors in a sliding window is discussed in [189].

The computation of identifiers is a straight application of Suffix trees (see [98, chapter 5]). Minimal unique factors can be left-extended to produce all identifiers of the word positions.

In genomics, a minimal unique factor, called a marker, has a known location on a chromosome and is used to identify individuals or species.

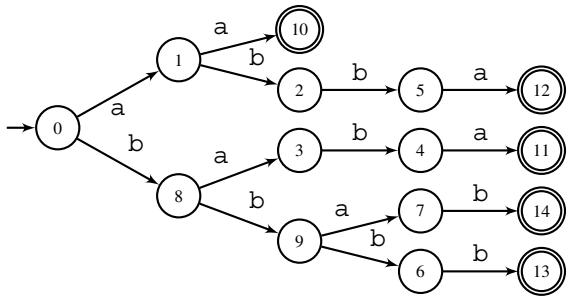
60 Minimal Absent Words

Words that do not occur in files provide a useful technique to discard or discriminate files. They act like the set of factors of a file but admit a more compact trie representation as factor codes.

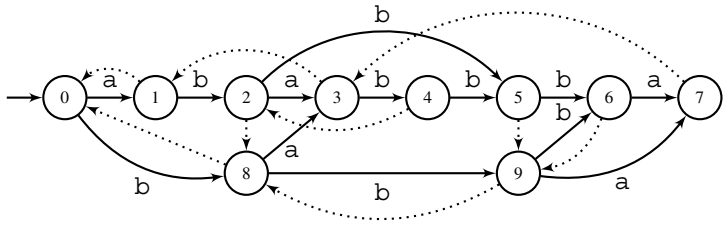
A word w, $|w| > 1$, is said to be absent or forbidden in a word x if it does not occur in x. It is said to be minimal with this property if in addition both $w[0\,..\,|w|-2]$ and $w[1\,..\,|w|-1]$ do occur in x.

Minimal absent words of `ababbba` are `aa`, `abba`, `baba`, `bbab` and `bbbb`. In their trie below, they are each associated with a leaf (double-circled) because no minimal absent word is a factor of another one.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press



A natural method to compute the minimal words absent in x is to start with its Factor automaton $\mathcal{F}(x)$, smallest deterministic automaton accepting all its factors. Each factor is spelt out along a path starting from the initial state and all states are terminal. Below is the Factor automaton of `ababbba` with its failure links (dotted edges).



Question. Design a linear-time algorithm to compute the trie of minimal absent words of a word x from its Factor automaton $\mathcal{F}(x)$.

[**Hint:** Use failure links of the automaton.]

Question. Design a linear-time algorithm to recover the Factor automaton of a word x from the trie of its minimal absent words.

[**Hint:** Use the Aho–Corasick technique.]

Solution

Computing minimal absent words. The algorithm below works on the Factor automaton $\mathcal{F}(x)$ of its input. The automaton on the alphabet A comprises a set of states Q with initial state i and the transition function goto, represented by arcs on the above picture. The algorithm detects absent words by considering undefined arcs.

The algorithm traverses the automaton in a width-first way to ensure the minimality of an absent words. It checks at line 3 if some proper suffix of a candidate is not already absent using the failure link of the automaton

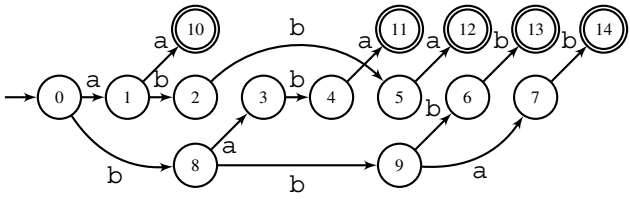
https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

(a by-product of efficient algorithms that build the automaton). The link refers to the longest suffix of the word associated with another state. The algorithm transform the automaton by adding new states and computing the new transition function $goto'$.

MinimalAbsentWords(x non-empty word)

```
1  (Q, A, i, Q, goto) ← F(x) Factor automaton of x
       and its failure function fail
2  for each p ∈ Q in width-first traversal from i and each a ∈ A do
3      if goto(p,a) undefined and goto(fail(p),a) defined then
4          goto'(p,a) ← new final state
5      elseif goto(p,a) = q and q not already reached then
6          goto'(p,a) ← q
7  return (Q ∪ {new final states}, A, i, {new final states}, goto')
```

Applied to the above example `ababbba`, the algorithm produces the trie of minimal absent words, drawn differently below to show the similarity with the automaton structure.



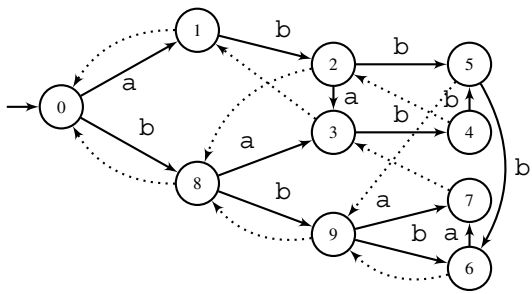
Running times. The construction of the Factor automaton of a word is known to be achievable in linear time, mostly due to the fact that the size of the structure is linear according to the length of the word independently of the alphabet size. The rest of instructions in the algorithm clearly takes $O(|A| \times |x|)$ time if the goto function and the failure links are implemented in arrays.

Computing a Factor automaton. From the trie $(Q, A, i, T', goto')$ of minimal absent words of a single word x, the algorithm below builds its Factor automaton. The construction relies on the failure link *fail* on states of the trie. The process is similar to the construction of a dictionary-matching machine that builds the automaton accepting all words in which appear a word from a finite set.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

```
FactorAutomaton((Q, A, i, T', goto') trie of minimal absent words)
 1  for each a ∈ A do
 2      if goto'(i,a) defined and not in T' then
 3          goto(i,a) ← goto'(i,a)
 4          fail(goto(i,a)) ← i
 5  for each p ∈ Q \ {i} in width-first traversal and each a ∈ A do
 6      if goto'(p,a) defined then
 7          goto(p,a) ← goto'(p,a)
 8          fail(goto(p,a)) ← goto(fail(p),a)
 9      elseif p not in T' then
10          goto(p,a) ← goto(fail(p),a)
11  return (Q \ T', A, i, Q \ T', goto)
```

The next picture displays the Factor automaton of ababbba, drawn differently to show the relation with the first picture of the trie.



Notes

The notion of a minimal absent or forbidden word was introduced by Béal et al. in [28]. The design and analysis of the present algorithms are in [92]. An extension to regular languages appears in [30].

The linear-time construction of a Factor automaton appears in [67]. It can be obtained by minimising the DAWG introduced by Blumer et al. (see [38]) or the Suffix automaton (see [74, 96, 98]).

The second algorithm is similar to the construction of a pattern-matching machine by Aho and Corasick [3]. Applied to the trie of minimal absent words of several words, the method does not always produce a (minimal) Factor automaton.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Antidictionaries, sets of absent words, are used in the data compression method in [93]; see more in [198–200] and references therein. Computation in a sliding window is discussed in [75, 189].

Absent words are also used in bioinformatics to detect avoided patterns in genomic sequences or to help build phylogenies; see, for example, [51, 224].

61 Greedy Superstring

A superstring of a set of words can be used to store the set in a compact way. Formally, a common superstring of a set X of words is a word z in which all elements of X occur as factors, that is, $X \subseteq Fact(z)$. The shortest common superstring of X is denoted by $\mathrm{SCS}(X)$.

The greedy paradigm leads to a simple algorithm that produces a fairly good approximation of $\mathrm{SCS}(X)$. The goal of the problem is to show an efficient implementation of the method.

For two words u and v, $Overlap(u,v)$ is the longest suffix of u that is a prefix of v. If $w = Overlap(u,v)$, $u = u'w$ and $v = wv'$, $u \otimes v$ is defined as the word $u'wv'$. Note that $\mathrm{SCS}(u,v)$ is either $u \otimes v$ or $v \otimes u$. Also note that a word in X factor of another word in X can be discarded without changing $\mathrm{SCS}(X)$. Then X is supposed to be factor free.

GreedySCS(X non-empty factor-free set of words)

```
1  if |X| = 1 then
2      return x ∈ X
3  else let x, y ∈ X, x ≠ y, with |Overlap(x, y)| maximal
4      return GreedySCS(X \ {x, y} ∪ {x ⊗ y})
```

Question. For a set X of words drawn from the alphabet $\{1, 2, \ldots, n\}$ show how to implement the algorithm so that GreedySCS(X) runs in time $O(\Sigma\{|x| : x \in X\})$.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Example. The superstring `fbdiachbgegeakhiacbd` is produced by GreedySCS from the set {`egeakh`, `fbdiac`, `hbgege`, `iacbd`, `bdiach`}.

```
                    i a c b d
              e g e a k h
        h b g e g e
  b d i a c h
f b d i a c
f b d i a c h b g e g e a k h i a c b d
```

Solution

The overlap between two words u and v is the border of the word $v\#u$, where # does not occur in the words. Hence, methods for computing borders in linear time (e.g. in Problem 19) lead to a direct implementation running in time $O(n \cdot |X|)$, where $n = \Sigma\{|x| : x \in X\}$. We show how to design a $O(n)$-time implementation.

If words in the above example are denoted by x_1, x_2, x_3, x_4 and x_5 the superstring produced by the algorithm is $x_2 \otimes x_5 \otimes x_3 \otimes x_1 \otimes x_4$. It is identified by the corresponding permutation $(2, 5, 3, 1, 4)$ of word indices.

Let us first design an iterative version of the algorithm that produces the permutation of word indices associated with the sought superstring. It is implemented as a doubly linked list whose elements are linked by the tables *prev* and *next*, and that starts at some index. During the computation, for a (partial) list starting with index p and ending with q we have $head[q] = p$ and $tail[p] = q$.

Here is the scheme of an iterative version of the algorithm.

IterGreedy($\{x_1, x_2, \ldots, x_m\}$ non-empty factor-free set of words)

```
1  for i ← 1 to m do
2      (prev[i], next[i]) ← (i, i)
3      (head[i], tail[i]) ← (i, i)
4  for m − 1 times do
5      let i, j, next[i] = i, prev[j] = j, head[i] ≠ j
         with |Overlap(x_i, x_j)| maximal
6      (next[i], prev[j]) ← (j, i)
7      head[tail[j]] ← head[i]
8      tail[head[i]] ← tail[j]
9  let i with prev[i] = i
10 return (i, next)
```

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Condition $next[i] = i$ on line 5 ensures that i is the tail of its list, and similarly condition $prev[j] = j$ that j is the head of its list. Condition $head[i] \neq j$ attests that i and j are on different lists, which instructions at line 6 concatenate. The next instructions update heads and tails.

From the output $(i, next)$, the permutation of indices associated with the superstring of $\{x_1, x_2, \ldots, x_m\}$ is i, $next[i]$, $next[next[i]]$, etc.

Algorithm IterGreedy is made efficient by introducing several useful data structures *Last* and *First*: for each $u \in Pref(\{x_1, \ldots, x_m\})$

- $Last(u)$ is the list of indices of words in $\{x_1, \ldots, x_m\}$ having u as a suffix,
- $First(u)$ is the list of indices of words in $\{x_1, \ldots, x_m\}$ having u as a prefix.

In addition, for each index of a word we keep all its locations in the lists to be able to delete it from the lists. Let $n = \Sigma_{i=1}^{m} |x_i|$.

Observation. The total length of all lists is $O(n)$.
Indeed, index i is on the list $First(u)$, for each proper prefix u of w_i. Hence it is on $|w_i|$ lists, which sums up to $O(n)$ globally. The same holds for *Last*.

Algorithm IterGreedy rewrites as Algorithm EffiGreedy that processes all potential overlaps u in order of decreasing length, and merges words having such overlaps if they are eligible for merge. Testing eligibility is done as in Algorithm IterGreedy.

```
EffiGreedy({x_1, x_2, ..., x_m} non-empty factor-free set of words)
 1  for i ← 1 to m do
 2      (head[i], tail[i]) ← (i, i)
 3  for each u ∈ Pref({x_1, ..., x_m}) in decreasing length order do
 4      for each i ∈ Last(u) do
 5          let j be the first element of First(u) with j ≠ head[i]
 6          ▷ it is the first or second element, or NIL
 7          if j ≠ NIL then   ▷ u is an overlap of x_i and x_j
 8              remove j from all lists First
 9              remove i from all lists Last
10              next[i] ← j
11              head[tail[j]] ← head[i]
12              tail[head[i]] ← tail[j]
13  let i word index for which i ≠ next[j] for all j = 1, ..., m
14  return (i, next)
```

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Algorithm EffiGreedy runs in time $O(n)$ if all lists are preprocessed, since their total size is $O(n)$.

The preprocessing of *Pref* and of other lists is done with the trie of the set $\{x_1, x_2, \ldots, x_m\}$ and with a Suffix tree. The only tricky part is the computation of lists $Last(u)$. To do it, let $\mathcal{T}'$ be the Suffix tree of $x_1\#_1x_2\#_2 \ldots x_m\#_m$, where $\#_i$ are new distinct symbols. Then for each word x_i, $\mathcal{T}'$ is traversed symbol by symbol along the path labelled by x_i and, for each prefix u of x_i, if the corresponding node in $\mathcal{T}'$ has k outgoing edges whose labels start with $\#_{i_1}, \ldots, \#_{i_k}$ respectively then indices $i_1, \ldots, i_k$ are inserted into $Last(u)$. This results in a $O(n)$ preprocessing time if the alphabet is linearly sortable.

Notes

Computing a shortest common superstring is a problem known to be NP-complete. Our version of Algorithm GreedySCS derives from the algorithm by Tarhio and Ukkonen in [230].

One of the most interesting conjectures on the subject is whether GreedySCS produces a 2-approximation of a shortest common superstring of the input. This is true for words of length 3 and is quite possibly always true.

62 Shortest Common Superstring of Short Words

A common superstring of a set X of words is a word in which all elements of X occur as factors. Computing a shortest common superstring (SCS) is an NP-complete problem but there are simple cases that can be solved efficiently, like the special case discussed in the problem.

For example, `1 2 6 3 9 2 3 7 5 7` is a shortest common superstring for the set of seven words {`1 2`, `2 3`, `2 6`, `5 7`, `6 3`, `7 5`, `9 2`}.

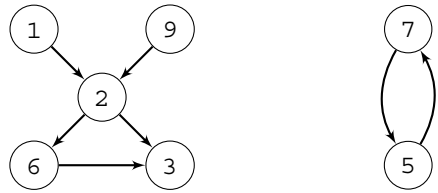
Question. Show how to compute in linear time the length of a shortest common superstring of a set X of words of length 2 over an integer alphabet.

[**Hint:** Transform the question into a problem on graphs.]

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Solution

The problem translates into a question on (directed) graphs as follows. From the set X we consider the graph G whose vertices are letters (integers) and whose edges correspond to words in X. The picture corresponds to the above example.



i	1	2	3	5	6	7	9
outdegree(i)	1	2	0	1	1	1	1
indegree(i)	0	2	2	1	1	1	0

A directed graph is said to be weakly connected if every two nodes are connected by an undirected path, after discarding edge orientations. The observation sketches the road map to build a shortest superstring.

Observation. Let G be a weakly connected graph associated to a set of length-2 words and let Z be the smallest set of (directed) edges to be added to G so that it contains an Eulerian cycle. Then the length of a shortest superstring for X is $|X| + 1$ if Z is empty and $|X| + |Z|$ otherwise.

The problem switches to the question of building an appropriate set Z to get an Eulerian graph. Recall that a directed graph is Eulerian if it is weakly connected and each vertex v is balanced, that is, $indegree(v) = outdegree(v)$. Each weakly connected component of the graph G is processed separately. So to start we can assume that G itself is weakly connected.

To add a minimum number of directed edges to make each vertex v balanced we proceed as follows. If $D(v) = outdegree(v) - indegree(v) > 0$, $D(v)$ incoming edges are added to v; if $D(v) < 0$, $D(v)$ outgoing edges are added from v. The point is that when adding an edge it is always from a vertex that needs an outgoing edge to a vertex that requires an incoming edge. Since each edge contribute to both an incoming edge and an outgoing edge, we cannot be left with only one vertex v with $D(v) \neq 0$ and the process continues until all vertices are balanced. This implies also that the total number of added edges is exactly $|Z| = \frac{1}{2} \sum_v |D(v)|$.

Computing Z is done easily using the tables *outdegree* and *indegree* and runs in linear time. Worst cases are when no two words of X overlap. When the transformed graph has an Eulerian cycle, deleting one of the added edges

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

$v \to w$ provides an Eulerian path from w to v. If the original graph is already Eulerian, a path starting from any vertex and ending to it gives a solution. Paths corresponds to shortest superstrings.

Finally, if the graph G is not weakly connected, each weakly connected component is treated as above and the resulting words are concatenated to get a shortest superstring.

On the above example, only two edges, `3` $\to$ `1` and `3` $\to$ `9`, are added to the left component, giving the new tables:

i	1	2	3	5	6	7	9
$outdegree(i)$	1	2	2	1	1	1	1
$indegree(i)$	1	2	2	1	1	1	1

Removing the first added edge give the word `1 2 6 3 9 2 3` for the first component and `7 5 7` for the second component. The resulting superstring is `1 2 6 3 9 2 3 7 5 7`.

Notes

The method presented in the problem is by Gallant et al. [126]. If the input set consists of words of length 3 the problem becomes NP-complete.

63 Counting Factors by Length

Let $fact_x[\ell]$ denote the number of (distinct) factors of length ℓ occurring in a word x.

Question. Show how to compute in linear time all numbers $fact_x[\ell]$, $\ell = 1, \ldots, |x|$, related to a word x, assuming a constant-size alphabet.

[**Hint:** Exploit the Suffix tree of the word.]

Solution

Let $T = \mathcal{ST}(x)$ be the Suffix tree of x. Recall its internal nodes are factors of x having at least two occurrences in x. They are followed either by at least two different letters or possibly by just one letter when one occurrence is a suffix occurrence.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

With a non-root node v of T whose parent is node u is associated the interval of lengths

$$I_v = [|u| + 1 \,..\, |v|].$$

Observation. $fact_x[\ell]$ is the number of intervals I_v containing the number ℓ.

Indeed, each non-root node v of the Suffix tree corresponds to a set of factors sharing the same set of occurrences. Their lengths are distinct and form the interval I_v. Hence the total number of (distinct) factors of length ℓ is the number of all intervals I_v containing ℓ.

The observation reduces the problem to an *interval covering problem*: given a family $\mathcal{I}(x)$ of subintervals of $[1 \,..\, |x|]$ compute the number of intervals containing ℓ, for each ℓ, $1 \le \ell \le |x|$.

```
NumbersOfFactors(I(x) for a non-empty word x)
 1  Count[1..|x|+1] ← [0,0,...,0]
 2  for each [i..j] ∈ I(x) do
 3      Count[i] ← Count[i] + 1
 4      Count[j+1] ← Count[j+1] − 1
 5  prefix_sum ← 0
 6  for ℓ ← 1 to n do
 7      prefix_sum ← prefix_sum + Count[ℓ]
 8      fact_x[ℓ] ← prefix_sum
 9  return fact_x
```

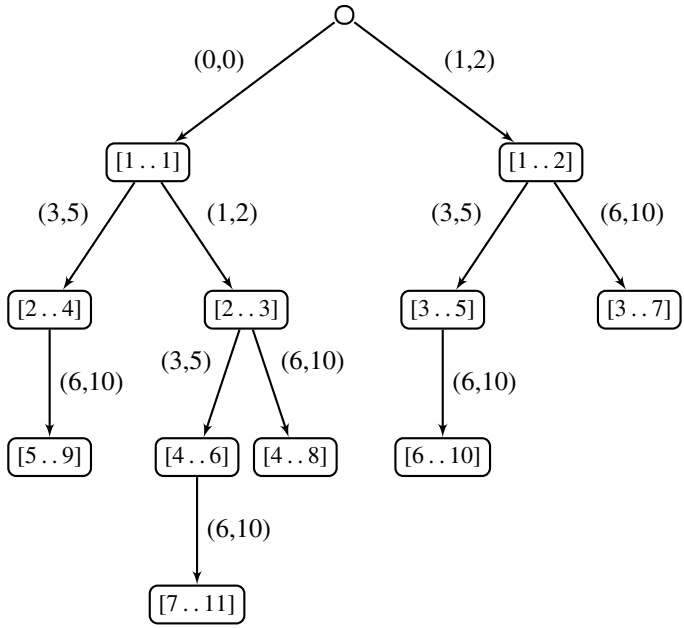
Algorithm NumbersOfFactors computes $fact_x$ from the family $\mathcal{I}(x)$ of intervals defined from the Suffix tree of x. To do it, an auxiliary array $Count[1 \,..\, n+1]$ that initially contains null values is used.

Example. Let $x =$ `abaababaaba`. The intervals in $\mathcal{I}(x)$ are labels of nodes in the picture below showing the Suffix tree of x. The algorithm computes the table $Count = [2, 1, 1, 1, 0, 0, 0, -1, -1, -1, -1]$ (discarding the value at position $|x| + 1$) and outputs the sequence of prefix sums of the table $Count$:

$$fact_x = [2, 3, 4, 5, 5, 5, 5, 4, 3, 2, 1].$$

For instance, the value $fact_x[3] = 4$ corresponds to the four factors of length 3 occurring in x: `aab`, `aba`, `baa` and `bab`.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press



It is clear Algorithm NumbersOfFactors runs in linear time mainly because the number of nodes in the Suffix tree of x is $O(|x|)$.

Notes

An alternative algorithm can be achieved using the Factor automaton of x. In the automaton each non-initial state v is labelled by the interval $[s(v)\,..\,l(v)]$, where $s(v)$ and $l(v)$ are respectively the length of the shortest and of the longest path from the root to v.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

64 Counting Factors Covering a Position

A factor of a word x covers a position k on x if it has an occurrence $x[i\,..\,j]$ that satisfies $i \leq k \leq j$.

Let $\mathcal{C}(x,k)$ denote the number of (distinct) factors of x covering the position k and let $\mathcal{N}(x,k)$ denote the number of factors having an occurrence that does not cover k.

Question. Show how to compute in linear time $\mathcal{C}(x,k)$ and $\mathcal{N}(x,k)$ for a given position k on x, assuming a constant-size alphabet.

Solution

Nodes of the Suffix tree $\mathcal{ST}(w)$ of a word w are factors of w. For an edge (u,v) of the tree let $weight(v) = |v| - |u|$, the length of its label.

Computing $\mathcal{N}(x,k)$. Let # be a letter that does not occur in x and let x' be the word obtained from x by changing its letter $x[k]$ to #. Then, using the Suffix tree $\mathcal{ST}(x')$ the number N of distinct factors of x' is computed as the sum of weights of all non-root nodes.

Observation. $\mathcal{N}(x,k) = N - M$, where M is the number of (distinct) factors of x' containing the letter #.

This leads to an evaluation of $\mathcal{N}(x,k)$ since $M = (k+1) \times (n-k)$.

Computing $\mathcal{C}(x,k)$. Assume x ends with special end-marker and each leaf of $\mathcal{ST}(x)$ is labelled with the starting position of the corresponding suffix of x. For each node v let $LeftLeaves(v,k)$ be the set of leaves i in the subtree rooted at v satisfying both $i \leq k$ and $k - i < |v|$.

Let V be the set of nodes v with a non-empty set $LeftLeaves(v,k)$. In other words, V is the set of nodes corresponding to factors covering the position k. For $v \in V$ let $Dist(v,k) = \min\{k - i : i \in LeftLeaves(v,k)\}$.

Observation. $\mathcal{C}(x,k) = \sum_{v \in V} \min\{|v| - Dist(v,k), weight(v)\}$.

Computing $\mathcal{C}(x,k)$ reduces to the computation of all $Dist(v,k)$, which is done during a bottom-up traversal of the Suffix tree.

On constant-size alphabets all computations run in linear time.

Notes

Interesting versions of the problem are when factors are to cover all positions of a set of positions. An attractor, a related notion introduced by Prezza [202] (see also [157, 183]), is a set K of positions on x whose factors have at least one occurrence covering an element of K. Attractors provide a framework to analyse dictionary text compressors and are used in [193] to develop universal compressed self-indexes.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

65 Longest Common-Parity Factors

For a word $v \in \{0,1\}^+$ we denote by $parity(v)$ the sum modulo 2 of letter 1 occurring in v. For two words $x, y \in \{0,1\}^+$ we denote by $lcpf(x,y)$, the ***longest common-parity factor***, the maximal common length of two factors u and v of x and y respectively with $parity(u) = parity(v)$. Surprisingly this problem essentially amounts to computing all periods of words.

Question. Show how to compute in linear time $lcpf(x,y)$ for two binary words x and y.

Solution

The solution uses a data structure called the *parity table*. For a word x, $Parity[\ell, x]$ is the set of distinct parities of factors of length ℓ of x. If two factors of length ℓ have different parities $Parity[\ell, x] = \{0,1\}$.

Observation. The length $lcpf(x,y)$ derives from the parity tables of x and of y: $lcpf(x,y) = \max\{\ell : Parity[\ell,x] \cap Parity[\ell,y] \neq \emptyset\}$.

Fast computation of the table *Parity*. The problem reduces to the computation of *Parity*, which relies on the following simple fact.

Observation. $Parity[\ell, x] \neq \{0,1\}$ if and only if ℓ is a period of x.

Indeed, if ℓ is a period of x then obviously the parity of all factors of length ℓ is the same. Conversely, assume all factors of length ℓ have the same parity. Then whenever the next sum is defined we get the equality $\sum_{j=i}^{i+\ell-1} x[j] \pmod 2 = \sum_{j=i+1}^{i+\ell} x[j] \pmod 2$, which implies $x[i+\ell] = x[i]$ and completes the proof.

All the periods of x are computed, for example, as by-products of the border table computation (see Problem 19). Next, when ℓ is a period of x, $Parity[\ell, x]$ is the parity of the prefix of length ℓ of x, which results from a prefix-sum computation for all ℓ's. If ℓ is not a period of x, by the observation, $Parity[\ell, x] = \{0,1\}$. In this way the entire parity tables for x and y are computed in linear time, which gives a linear-time solution to compute $lcpf(x,y)$.

Notes

The problem extends to words on a larger alphabet $\{0,1,\ldots,k-1\}$ considering sums modulo k. A similar algorithm gives a solution.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

66 Word Square-Freeness with DBF

The *dictionary of Basic Factors* (*DBF* in short) is a useful elementary data structure to produce efficient algorithmic solutions to many problems on words. It is used here to test the square-freeness of a word.

The DBF of a word w consists of a logarithmic number of tables $Name_k$, $0 \leq k \leq \log |w|$. Tables are indexed by positions on w and $Name_k[j]$ intends to identify $w[j\,..\,j+2^k-1]$, factor of length 2^k starting at position j on w. Identifiers have the following property, for $i \neq j$: $Name_k[i] = Name_k[j]$ if and only if

$$i+2^k-1, j+2^k-1 < |w| \text{ and } w[i\,..\,i+2^k-1] = w[j\,..\,j+2^k-1].$$

It is known that the DBF of w can be computed in time $O(|w| \log |w|)$.

To test the square-freeness of w, let $Pred_k$, $0 \leq k < \log |w|$, denote the table indexed by positions on w and defined by

$$Pred_k[j] = \max\{i < j : Name_k[i] = Name_k[j]\} \cup \{-1\}$$

and let $Cand_w$ denote the set of pairs of positions $(i, 2j-i)$ on w, candidates for a square occurrence $w[i\,..\,2j-i-1]$ in w:

$$Cand_w = \{(i, 2j-i) : 2j-i \leq |w| \text{ and } i = Pred_k[j] \neq -1 \text{ for some } k\}.$$

Question. Show that a word w contains a square if $w[p\,..\,q-1]$ is a square for some $(p,q) \in Cand_w$. Deduce an algorithm checking if the word w is square-free in time $O(|w| \log |w|)$.

Example. Here are the tables *Pred* for the word $w =$ `abacbaca`:

j	0	1	2	3	4	5	6	7
$x[j]$	a	b	a	c	b	a	c	a
$Pred_0[j]$	−1	−1	0	−1	1	2	3	5
$Pred_1[j]$	−1	−1	−1	−1	1	2	−1	
$Pred_2[j]$	−1	−1	−1	−1	−1			

The associated set $Cand_w$ is $\{(0,4),(1,7),(2,8)\}$. Only the pair $(1,7)$ corresponds to a square, namely $w[1\,..\,6] =$ `bacbac`.

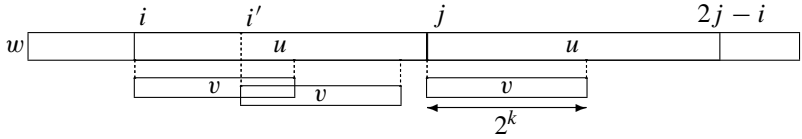
Solution

To answer the first part of the question, let i be the starting position of an occurrence of a shortest square occurring in w. Let 2ℓ be its length and

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

$j = i + \ell$. The square factor is $w[i\,..\,i + 2\ell - 1]$ and $u = w[i\,..\,i + \ell - 1] = w[j\,..\,j + \ell - 1]$. Let us show that $(i, i + 2\ell)$ belongs to $Cand_w$, that is, $i = Pred_k[j]$ for some integer k.

Let k be the largest integer for which $2^k \leq \ell$. As prefix of u we have $v = w[i\,..\,i + 2^k - 1] = w[j\,..\,j + 2^k - 1]$, that is, $Name_k[i] = Name_k[j]$.



By contradiction, assume $i < Pred_k[j]$, that is, there is an occurrence of v starting at position $Pred_k[j]$ (i' on picture). This occurrence is distinct from the occurrences of v prefixes of u (see picture) and overlaps at least one of them due to its length. But this yields a shorter square, a contradiction. Thus $Pred_k[j] = i$, which means $(i, i + 2\ell) \in Cand_w$ as expected.

Algorithm SQUAREFREE applies the above property by searching $Cand_w$ for a pair of positions corresponding to a square.

```
SQUAREFREE(w non-empty word of length n, DBF of w)
1  for k ← 0 to ⌊log n⌋ do
2      compute Pred_k from DBF of w
3  compute Cand_w from Pred_k tables
4  for each pair (p,q) ∈ Cand_w do
5      k ← ⌊log(q − p)/2⌋
6      if Name_k[p] = Name_k[(p + q)/2] and
           Name_k[(p + q)/2 − 2^k] = Name_k[q − 2^k] then
7          return FALSE
8  return TRUE
```

Correctness of SQUAREFREE. The correctness of the algorithm is an extension of the previous proof that justifies the choice of k at line 5. Testing if a pair (p, q) corresponds to a square, that is, checking if the two factors $w[p\,..\,(p + q)/2 - 1]$ and $w[(p + q)/2\,..\,q - 1]$ are equal, then amounts to checking both that their prefixes of length 2^k match and that their suffixes of length 2^k also match. This is exactly what is done at line 6 in the algorithm.

Running time of SQUAREFREE. For a given k the table $Pred_k$ can be computed in linear time, scanning the table $Name_k$ from left to right. Computing the set $Cand_w$ by traversing the $\lfloor \log |x| \rfloor$ tables $Pred$ takes $O(|x| \log |x|)$ time.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

The same bound holds for lines 5-7 thanks to *Name* from the DBF structure. Thus SQUAREFREE runs in time $O(|x| \log |x|)$.

Notes

There are many algorithms testing the square-freeness of a word with similar running time. But this one is especially simple when the DBF structure is available. It is a version of an algorithm published in [84].

67 Generic Words of Factor Equations

The problem deals with an algorithm to build words from factor equations. A factor equation is of the form $w[p\,..\,q] = w[p'\,..\,q']$ and has length $q-p+1$. In short, the equation is written as a triple $(p, p', q - p + 1)$.

We say that a word w of length n is a solution to a system of factor equations E if it satisfies each equation of the system. We are interested in generic solutions containing the largest number of different letters. Such a solution of length n can be used to describe all other solutions of the system. It is denoted by $\Psi(E, n)$ and defined up to a renaming of letters.

Example. For the system of equations

$$E = \{(2,3,1),\ (0,3,3),\ (3,5,3)\},$$

the generic solution $\Psi(E, 8)$ is $w = \texttt{abaababa}$. Indeed, $w[2] = w[3] = \texttt{a}$, $w[0\,..\,2] = w[3\,..\,5] = \texttt{aba}$ and $w[3\,..\,5] = w[5\,..\,7] = \texttt{aba}$. In other words, we have an equivalence on the set of positions on w comprising two equivalence classes $\{0, 2, 3, 5, 7\}$ and $\{1, 4, 6\}$. It is the finest equivalence satisfying the equations in E. Note that $\Psi(E, 11) = \texttt{abaababacde}$.

Question. Describe how to build a generic solution $\Psi(E, n)$ for a given system of factor equations E in time $O((n + m) \log n)$, where $m = |E|$.

[**Hint:** Use spanning forests to represent sets of equivalent positions.]

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

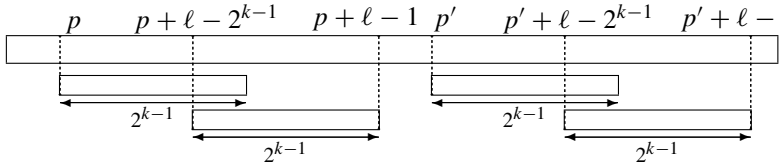
Solution

For $k \geq 0$, let E_k be the subset of equations of length ℓ, $2^{k-1} < \ell \leq 2^k$, in E. In particular, E_0 is its subset of equations of length 1.

Operation REDUCE**.** Let $k > 0$. For a set X of equations of length ℓ, $2^{k-1} < \ell \leq 2^k$, the operation REDUCE(X) produces an equivalent set of equations of shorter length as follows.

- **Split**. Each equation (p, p', ℓ) in X is replaced by two equations

$$(p, p', 2^{k-1}) \text{ and } (p + \ell - 2^{k-1}, p' + \ell - 2^{k-1}, 2^{k-1}).$$



  After the operation, X is transformed into an equivalent system of size $O(n + m)$ of equations of the same length.
- Then we create the graph G whose vertices are starting positions of equations of the same length 2^{k-1} and whose edges correspond to equations. If there is a cycle in G then we can remove one of its edges together with the corresponding equation without changing equivalence classes.
- **Shrink**. A spanning tree is built for each connected component of G. Trees form a spanning forest of the whole graph. Eventually, REDUCE(X) is the set of equations corresponding to edges of the spanning forest.

Key observation. Since there are $O(n)$ edges in the spanning forest, the size of the set of equations $|\text{REDUCE}(X)|$ is $O(n)$.

Main algorithm. The whole algorithm consists in applying a logarithmic number of iterations executing operation REDUCE. After each iteration the obtained equivalent system contains equations of much smaller length.

Eventually we get a system E_0 with equations of length 1, from which $\Psi(E_0, n) = \Psi(E, n)$ is easily computed in linear time.

```
PSI(E set of equations, n positive length)
1  ▷ E = ⋃_{i=0}^{⌈log n⌉} E_i
2  for k ← ⌈log n⌉ downto 1 do
3      E_{k-1} ← E_{k-1} ∪ REDUCE(E_k)
4      ▷ invariant: system ⋃_{i=0}^{k-1} E_i is equivalent to E
5  return Ψ(E_0, n)
```

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

The last system E_0 concerns only single positions and gives equivalence classes of positions. All positions in the same equivalence class are assigned the same letter, unique for the class. The resulting word is the required word $\Psi(E,n)$. Since the operation REDUCE runs in time $O(n+m)$, the whole algorithm runs in time $O((n+m)\log n)$, as expected.

Notes

The present algorithm is a version of the algorithm by Gawrychowski et al. presented in [127]. In fact, the algorithm is transformed in [127] into a linear-time algorithm using intricate data structures. It can be used to construct a word having a given set of runs, if there are any.

68 Searching an Infinite Word

The goal is to design an algorithm for matching a pattern in an infinite word. Since there is no answer for general infinite words, we restrict the question to a pure morphic word. It is an infinite word obtained by iterating a morphism θ from A^+ to itself, where $A = \{\mathtt{a},\mathtt{b},\ldots\}$ is a finite alphabet. To do so, we assume that θ is prolongable over the letter a, that is, $\theta(\mathtt{a}) = \mathtt{a}u$ for $u \in A^+$. Then $\Theta = \theta^\infty(\mathtt{a})$ exists and is $\mathtt{a}u\theta(u)\theta^2(u)\ldots$. The infinite word Θ is a fixed point of θ, that is, $\theta(\Theta) = \Theta$.

To avoid trivial cases, like that of the morphism η defined by $\eta(\mathtt{a}) = \mathtt{ab}$, $\eta(\mathtt{b}) = \mathtt{c}$, $\eta(\mathtt{c}) = \mathtt{b}$ and $\eta(\mathtt{d}) = \mathtt{d}$ where the letter $\mathtt{d}$ is useless and the letter $\mathtt{a}$ appear only once in Θ, we assume that θ is irreducible. It means that any letter is accessible from any letter: for any distinct letters $c,d \in A$ the letter d appears in $\theta^k(c)$ for some integer k.

Thue–Morse morphism μ and Fibonacci morphism ϕ (see Chapter 1) are both irreducible morphisms.

Question. Show how to test if a morphism is irreducible.

Question. Design an algorithm to compute the set of factors of length m occurring in the infinite word $\Theta = \theta^\infty(\mathtt{a})$, where θ is an irreducible morphism.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

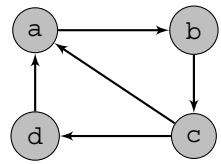
When the set of length-m factors of Θ is represented by a (deterministic) trie, testing if a pattern of length m appears in Θ becomes a mere top-down traversal of the trie.

Solution

To test the irreducibility of the morphism θ we build its accessibility graph on letters. Vertices of the graph are letters and, for any two different letters c and d, there is an arc from c to d if d appears in $\theta(c)$. Irreducibility holds if the graph contains a cycle going through all alphabet letters, which can be tested in polynomial time.

For example, the graph of the morphism ζ satisfies the property

$$\begin{cases} \zeta(\mathtt{a}) = \mathtt{ab} \\ \zeta(\mathtt{b}) = \mathtt{c} \\ \zeta(\mathtt{c}) = \mathtt{cad} \\ \zeta(\mathtt{d}) = \mathtt{a} \end{cases}$$



To solve the second question, one can extract length-m factors from words $\theta^k(\mathtt{a})$ by iterating the morphism from a. Indeed it is rather clear that after a finite number of iterations all length-m factors are captured.

Instead, the algorithm below handles only words that are images by θ of factors of Θ having length at most m. Its correctness is a consequence of the irreducibility of the morphism because it implies that any factor of $\theta^k(\mathtt{a})$ is a factor of $\theta^\ell(b)$ for any letter b and some integer ℓ.

The sought set of length-m factors of Θ is the set of length-m words stored in the trie T produced by the algorithm.

FACTORS(irreducible morphism $\theta, \mathtt{a} \in A$, positive integer m)

```
1  initialise T to the empty trie
2  Queue ← A
3  while Queue not empty do
4      v ← extract the first word in Queue
5      w ← θ(v)
6      for each length-m factor z of w do
7          if z not in T then
8              insert z into T and append z to Queue
9      if |w| < m and w not in T then
10         insert w into T and append w to Queue
11 return T
```

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Depending on the properties of the morphism, the algorithm can be tuned to get a faster execution. This is the case if, for example, the morphism is k-uniform: $|\theta(c)| = k$ for any letter c. Then only factors of length $\lfloor m/k \rfloor + 1$ need to be appended to the queue, which reduces dramatically the number of words put in the queue.

The insertion into the trie at line 8 can be implemented carefully to avoid useless operations. In fact, after inserting the factor $z = cy$ (for some letter c) it is natural to continue with the next factor of the form yd (for some letter d). If the trie is equipped with suffix links (same links as in a Suffix tree) the operation takes constant time (or at most $\log |A|$). Then the insertion of all factors z of w takes $O(|w|)$ time (or $O(|w| \log |A|)$).

Notes

A stronger hypothesis on the morphism is to be primitive, which means that there is an integer k for which the letter d appears in $\theta^k(c)$ for any $c, d \in A$ (k is independent of the pair of letters). For primitive morphisms there is another solution to the problem. It consists in considering return words in the infinite word x: a return word to a factor w of x is a shortest (non-empty) word r for which rw has border w and is a factor of x. Durand and Leroy [104] prove, for a primitive morphism θ, that there is a constant K for which both $|r| \leq K|w|$ and all length-m factors of Θ appear in factors of length $(K+1)m$. Moreover they are able to bound the constant K by $\max\{\theta(c) : c \in A\}^{4|A|^2}$. This leads to another algorithm for finding the set of length-m factors of Θ.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

69 Perfect Words

A word of length is called *dense* if it has the largest number of (distinct) factors among words of the same length on the same alphabet. A word is said to be *perfect* if all its prefixes are dense. Note that each prefix of a perfect word is also perfect.

Example. The word `0110` is dense but `0101` is not. The longest binary perfect words are `011001010` and its complement `100110101`, they have length 9. However, on the ternary alphabet the word `0120022110` of length 10 is perfect.

There are only finitely many binary perfect words, but the situation changes dramatically for larger alphabets.

Question. Show how to construct in linear time a ternary perfect word of any given length. Prove also the existence of an infinite perfect ternary word.

[**Hint:** Consider Hamiltonian and Eulerian cycles in de Bruijn automata.]

Solution

Let $A = \{0,1,2\}$ be the alphabet and consider the length $\Delta_n = 3^n + n - 1$ of a de Bruijn word of order n over A. It is enough to show how to construct perfect words having these particular lengths, since their prefixes are perfect.

Any perfect ternary word of length Δ_n is a de Bruijn word. Hence the problem reduces to the construction of perfect de Bruijn words.

Our basic data structure is the de Bruijn graph G_n of order n (graph structure of de Bruijn automaton) over the alphabet A. Vertices of G_n are ternary words of length $n-1$. The label of an Eulerian cycle is a circular de Bruijn word of order n, which produces a (linear) de Bruijn word of the same order when its prefix of length $n-1$ is appended to it.

Our first goal is to extend such a de Bruijn word w of order n to a de Bruijn word of order $n+1$. Let u be the border of length $n-1$ of w and ua its prefix of length n. Let $\widehat{w} = wa$.

Observation. In the graph G_{n+1} whose vertices are the words of A^n, $\widehat{w}$ is the label of a Hamiltonian cycle, denoted $Cycle_n(\widehat{w})$, starting and ending at vertex ua, prefix of length n of w.

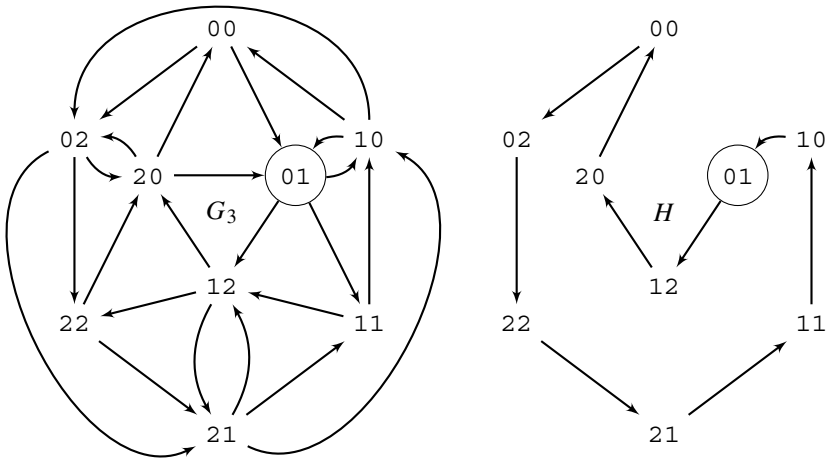
Example. The word $w = \texttt{0122002110}$ is a de Bruijn word of order 2 on A. It is associated with the Eulerian cycle in G_2:

$Cycle_1(w) = 0 \to 1 \to 2 \to 2 \to 0 \to 0 \to 2 \to 1 \to 1 \to 0.$

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Hence the word $\widehat{w} = \texttt{01220021101}$ corresponds to the Hamiltonian cycle H in G_3 (see picture where loops at nodes `00`, `11` and `22` are omitted for clarity):

$Cycle_2(\widehat{w}) = 01 \to 12 \to 22 \to 20 \to 00 \to 02 \to 21 \to 11 \to 10 \to 01.$



Drawing on the observation, the cycle is concatenated to a disjoint cycle to create an Eulerian cycle in G_{n+1} yielding a de Bruijn word of order $n+1$ prefixed by w.

The next goal is to extend a perfect de Bruijn word of order n to a perfect de Bruijn word of order $n+1$. To do so, we construct a sequence of perfect de Bruijn words $w_1, w_2, \ldots$ that satisfies: w_i is a prefix of w_{i+1}. The limit is then a perfect infinite word, as expected.

Let $EulerExt_n(h)$ be an Eulerian cycle in G_n extending a Hamiltonian cycle h in G_n, if this is possible. Let also $Word_n(e)$ be the word associated with an Eulerian cycle e in G_n.

PERFECTWORD(N positive length, $\{0,1,2\}$ alphabet)

```
1  (w,n) ← (012,1)
2  while |w| < N do
3      n ← n + 1
4      h ← Cycle_n(ŵ)
5      e ← EulerExt_n(h)
6      w ← Word_n(e)
7  return prefix of length N of w
```

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Informal explanation of the construction. The word w_n after extending it by one letter to $\widehat{w_n}$ corresponds to a Hamiltonian cycle $h = Cycle_n(\widehat{w_n})$ in G_{n+1}. We extend it to an Eulerian cycle e in G_{n+1}, and finally we define w_{n+1} as the word representation of e. The interesting point in this construction is that we treat cycles as words and words as cycles, and, in the main step, for computing an Eulerian extension we use graph-theoretic tools rather than stringologic arguments.

Example. For the perfect word $w_2 = \texttt{0120022110}$ of length 10 we have $\widehat{w}_2 = \texttt{01200221101}$, which corresponds in G_3 to the cycle H (see above picture):

$$\texttt{01} \to \texttt{12} \to \texttt{20} \to \texttt{00} \to \texttt{02} \to \texttt{22} \to \texttt{21} \to \texttt{11} \to \texttt{10} \to \texttt{01}.$$

H is extended to an Eulerian cycle E by concatenating it with the following Eulerian cycle in $G_3 - H$:

$$\texttt{01} \to \texttt{11} \to \texttt{11} \to \texttt{12} \to \texttt{21} \to \texttt{12} \to \texttt{22} \to \texttt{22} \to \texttt{20} \to \texttt{02}$$
$$\to \texttt{21} \to \texttt{10} \to \texttt{02} \to \texttt{20} \to \texttt{01} \to \texttt{10} \to \texttt{00} \to \texttt{00} \to \texttt{01}.$$

Finally we get from E:

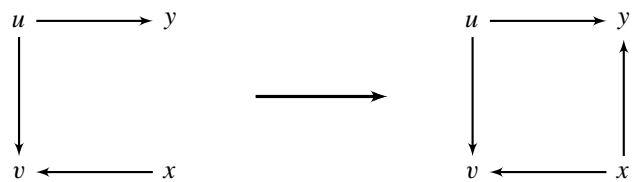
$$w_3 = Word(E) = \texttt{01200221101 11212220210 2010001}.$$

Before showing the word produced by the algorithm is perfect, we need to be sure it is possible to get an Eulerian cycle in $G_n - H$.

***Lemma* 5** *If H is a Hamiltonian cycle in G_n then after removing the edges of H the graph G_n remains Eulerian.*

Proof We use the following obvious but useful property of de Bruijn graphs shown schematically in the figure below: a special configuration of 3 edges implies the existence of the 4th edge. More formally:

(∗) If $u \to v, u \to y, x \to v$ are edges of G_n then $x \to y$ is an edge.



We are to show that $G_n - H$ is Eulerian. Clearly each node of $G_n - H$ has the same in-degree and out-degree. Hence it is enough to show it is strongly connected. However, weak connectivity (disregarding directions of edges) is sufficient due to well-known property (a graph is regular when its vertices have the same degree).

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

Property. A regular weakly connected directed graph is also strongly connected.

Hence it is enough to show that, for any edge $u \to v \in H$, nodes u and v are weakly connected in $G_n - H$ (there is a path between them not using edges of H and disregarding directions of edges). Indeed, since each node has in-degree and out-degree 3, there are nodes x, x', y for which the edges

$$u \to y,\ x \to v,\ x' \to v$$

are in $G_n - H$. Now property $(*)$ implies the existence in G_n of two additional edges $x \to y$ and $x' \to y$ (see the figure), and at least one of them is not in H. Removing directions of these edges, we deduce there is an undirected path from u to v, not using edges of H.

Consequently, $G_n - H$ is weakly connected and is Eulerian (as a directed graph), which completes the proof. ∎

Correctness of PerfectWord. Let w_n be the value of w immediately before instruction at line 3. By induction, all the prefixes of length at most $|w_{n-1}|$ of w_n are dense since w_{n-1} is perfect. A longer prefix of w_n contains all words of length $n-1$ and no repeat of words of length n, since it is a prefix of a de Bruijn word. Consequently it is also dense. Hence each prefix of w_n is dense, so w_n is perfect.

Complexity. The algorithm runs in linear time since Eulerian cycles can be found in linear time, and in de Bruijn graphs finding a Hamiltonian cycle reduces to the computation of an Eulerian cycle.

Notes

Perfect words are also called *super complex* and their construction is presented in [237]. In case of binary words the notion of perfect words is weakened to semi-perfect words whose existence is shown in [206].

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

70 Dense Binary Words

A word is called *dense* if it has the largest number of (distinct) factors among words of the same length on the same alphabet.

Over an alphabet with at least three letters, generating dense words for any given length is solved by the generation of perfect words (see Problem 69). But the solution does not apply to binary words and the present problem shows how to deal efficiently with this case.

Question. Show how to construct in $O(N)$ time a dense binary word of any given length N.

[**Hint:** Consider Hamiltonian and Eulerian cycles in de Bruijn automata.]

Solution

Let $A = \{0,1\}$ be the alphabet. Let us fix N and let n be such that $\Delta_{n-1} < N \leq \Delta_n$, where $\Delta_n = 2^n + n - 1$. Our basic data structure is the de Bruijn graph G_n of order n (graph structure of de Bruijn automaton) over the alphabet A. Vertices of G_n are binary words of length $n-1$.

We say that a path π in G_n is an *Eulerian chain* if it contains all nodes of G_n, possibly many times, and no repeating edge. Let $Word_n(\pi)$ be the word associated with the Eulerian cycle π in G_n.

Property 1. When π is an Eulerian chain of length $N-(n-1)$ in G_n, $Word_n(\pi)$ is a dense word of length N.

Proof Any binary word of length N, where $\Delta_{n-1} < N \leq \Delta_n$, contains at most 2^{n-1} (distinct) factors of length $n-1$ and at most $N-n+1$ factors of length n. Hence a word achieving these bounds is dense. In particular, if π is an Eulerian chain, $Word_n(\pi)$ contains all words of length $n-1$ and all its factors of length n are distinct since they correspond to distinct edges of the Eulerian chain in G_n. Consequently $Word_n(\pi)$ is dense. ∎

Following property 1, the answer to the question lies in the next property.

Property 2. An Eulerian chain of a length $N-(n-1)$ in G_n can be computed in linear time.

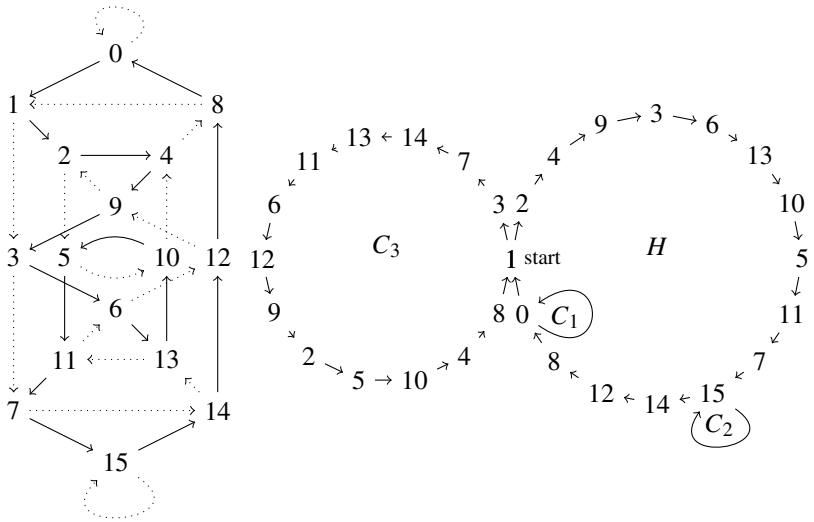
Proof To do it we first compute a Hamiltonian cycle H of size 2^{n-1} in G_n, given by an Eulerian cycle of G_{n-1}. The graph $G_n - H$ consists of disjoint simple cycles $C_1, C_2, \ldots, C_r$, called *ear-cycles*. Then we choose a

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

subset $C'_1, C'_2, \ldots, C'_t$ of ear-cycles for which $\sum_{i=1}^{t-1} |C'_i| < M \leq \sum_{i=1}^{t} |C'_i|$. Then we add a prefix subpath c'_t of C'_t to get

$$\sum_{i=1}^{t-1} |C'_i| + |c'_t| = M.$$

It is clear that $H \cup C'_1 \cup C'_2 \cup \cdots \cup C'_{t-1} \cup c'_t$ can be sequenced into an Eulerian chain of length M. It starts at any node of c'_t, then goes around H and around each encountered ear-cycle C'_i. After coming back it traverses the path c'_t. ∎



Example. The above picture displays G_5 (left) whose vertices are binary words of length 4, shown in decimal to shorten the display. The picture (right) shows the decomposition of G_5 into the edge-disjoint ear-cycles H, C_1, C_2 and C_3. Cycle H is the Hamiltonian cycle of length 16, C_1 and C_2 are loops and C_3 is the big ear-cycle of length 14. The three last ear-cycles cover the dotted edges (left) in G_5, those that are not in H.

To compute a dense binary word of length $N = 33$, we first construct an Eulerian chain π of length $21 = 25 - 4$. We can start at node 1, go around the Hamiltonian cycle additionally traversing the two loops, then come back to 1, and follow a path of 4 edges on the big ear-cycle C_3. In this case $t = 3$, C'_1, C'_2 are loops and $c'_3 = 1 \to 3 \to 7 \to 14$. We get the path

$$\pi = (1, 2, 4, 9, 3, 6, 13, 10, 5, 11, 7, 15, 15, 14, 12, 8, 0, 0, 1, 3, 7, 14)$$

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

whose label is the binary word

$$0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,1\,1\,0\,0\,0\,0\,0\,1\,1\,1\,0.$$

The final dense word of length 25 results by prepending to it the binary representation 0 0 0 1 of the first node 1:

$$Word_5(\pi) = 0\,0\,0\,1\,0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,1\,1\,0\,0\,0\,0\,0\,1\,1\,1\,0.$$

Notes

The first efficient and quite different algorithm for constructing dense words was by Shallit [222]. Observe that in our example, for $n = 5$, the graph G_5 decomposes into four edge disjoint simple cycles: a Hamiltonian cycle H, two loops and one big ear-cycle of length $2^{n-1} - 2$. If we disregard the loops then G_5 is decomposed into two edge-disjoint simple cycles. In fact, such a special decomposition of any binary graph G_n, for $n > 3$, can be found using so-called *complementary* Hamiltonian cycles, see [206]. Nevertheless any other decomposition is sufficient to compute dense words.

71 Factor Oracle

The Factor oracle is an indexing structure similar to the Factor or Suffix automaton (or DAWG) of a word x. It is a deterministic automaton with $|x|+1$ states, the minimum number of states the Suffix automaton of x can have. This makes it a well-suited data structure in many applications that require a simple indexing structure and leads both to a space-economical data structure and to an efficient online construction. The drawback is that the oracle of x accepts slightly more words than the factors of x.

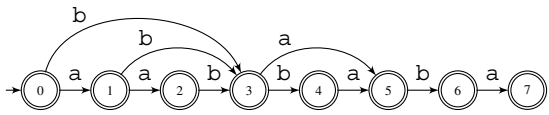
For a factor v of y, let $pocc(v, y)$ be the position on y following the first occurrence of v in y, that is, $pocc(v, y) = \min\{|z| : z = wv \text{ prefix of } y\}$. The following algorithm may be viewed as a definition of the Factor oracle $\mathcal{O}(x)$ of a word x. It computes the automaton in which Q is the set of states and E the set of labelled edges.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

```
ORACLE(x non-empty word)
1  (Q, E) ← ({0, 1, ..., |x|}, ∅)
2  for i ← 0 to |x| − 1 do
3      u ← shortest word recognised in state i
4      for a ∈ A do
5          if ua ∈ Fact(x[i − |u| .. |x| − 1]) then
6              E ← E ∪ {(i, a, pocc(ua, x[i − |u| .. |x| − 1]))}
7  return (Q, E)
```

Actually the structure has several interesting properties. Its $|x|+1$ states are all terminal states. Every edge whose target is $i+1$ is labelled by $x[i]$. There are $|x|$ edges of the form $(i, x[i], i+1)$, called internal edges. Other edges, of the form $(j, x[i], i+1)$ with $j < i$, are called external edges. The oracle can thus be represented by x and its set of external edges without their labels.

Example. The oracle $\mathcal{O}(\texttt{aabbaba})$ accepts all the factors of `aabbaba` but also `abab`, which is not. It is determined by its external unlabelled edges $(0,3)$, $(1,3)$ and $(3,5)$.



Question. Show that the Factor oracle of a word x has between $|x|$ and $2|x|-1$ edges.

Solution

First note the bounds are met. Indeed, $\mathcal{O}(a^n)$ has n edges for any letter a, and $\mathcal{O}(x)$ has $2|x|-1$ edges when the letters of x are pairwise distinct, that is, $|alph(x)| = |x|$.

Fact. Let u be a shortest word among the words recognised in state i of $\mathcal{O}(x)$. Then $i = pocc(u, x)$ and u is unique. Let $sh(i)$ denote it.

To answer the question, since there are $|x|$ internal edges, we have to show there are less than $|x|$ external edges. To do so, let us map each external edge of the form (i, a, j) with $i < j-1$ to the proper non-empty suffix $sh(i)ax[j+1\,..\,|x|-1]$ of x. We show the mapping is injective.

Assume there are edges (i_1, a_1, j_1) and (i_2, a_2, j_2) with

$$sh(i_1)a_1x[j_1+1\,..\,|x|-1] = sh(i_2)a_2x[j_2+1\,..\,|x|-1]$$

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

and w.l.o.g. that $i_1 \leq i_2$.

- If $j_1 < j_2$ then $sh(i_1)a_1$ is a proper prefix of $sh(i_2)$. Setting $d = |sh(i_2)| - |sh(i_1)a_1|$ we get $j_1 = j_2 - d - 1$. An occurrence of $sh(i_2)$ ends in i_2 then an occurrence of $sh(i_1)a_1$ ends in $i_2 - d < j_2 - d - 1 = j_1$. But this is a contradiction with the construction of the Factor oracle of x.
- If $j_1 > j_2$ the word $sh(i_2)$ is a proper prefix of $sh(i_1)$. Consequently there is an occurrence of $sh(i_2)$ ending before $i_1 \leq i_2$, a contradiction again with the construction of the Factor oracle.

Therefore $j_1 = j_2$, which implies $a_1 = a_2$, $sh(i_1) = sh(i_2)$, $i_1 = i_2$ and eventually $(i_1, a_1, j_1) = (i_2, a_2, j_2)$.

Since the mapping is injective and since there are $|x| - 1$ proper non-empty suffixes of x, adding internal and external edges gives the maximum of $2|x| - 1$ edges in the Factor oracle, as expected.

Question. Design an online construction of the Factor oracle of a word x running in linear time on a fixed alphabet with linear space.

[**Hint:** Use suffix links.]

Solution

Since the oracle is deterministic, let δ denote its transition function, that is, $\delta(i, a) = j \Leftrightarrow (i, a, j) \in E$. Let S be the suffix link defined on states as follows: $S[0] = -1$ and, for $1 \leq i \leq |x|$, $S[i] = \delta(0, u)$ where u is the longest (proper) suffix of $x[0\,..\,i]$ for which $\delta(0, u) < i$. For the above example we get

i	0	1	2	3	4	5	6	7
$x[i]$	a	a	b	b	a	b	a	
$S[i]$	−1	0	1	0	3	1	3	5

Fact. Let $k < i$ be a state on the suffix path of state i of the Factor oracle of $x[0\,..\,i]$. If $\delta(k, x[i+1])$ is defined then the same holds for all the states on the suffix path of k.

Following the fact, step i, for $0 \leq i \leq |x| - 1$, of the online construction of the Factor oracle makes a standard use of the suffix link and consists of

- adding state $i + 1$ and setting $\delta(i, x[i]) = i + 1$;
- following the suffix path of i to set $\delta(S^k[i], x[i]) = i + 1$ whenever necessary; and
- setting $S[i + 1]$.

The following algorithm implements this strategy.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

```
OracleOnline(x non-empty word)
 1  (Q, δ, S[0]) ← ({0}, undefined, −1)
 2  for i ← 0 to |x| − 1 do
 3      Q ← Q ∪ {i + 1}
 4      δ(i, x[i]) ← i + 1
 5      j ← S[i]
 6      while j > −1 and δ(j, x[i]) undefined do
 7          δ(j, x[i]) ← i + 1
 8          j ← S[j]
 9      if j = −1 then
10          S[i + 1] ← 0
11      else S[i + 1] ← δ(j, x[i])
12  return (Q, δ)
```

The correctness of OracleOnline comes mainly from the equality $(S[i], x[i], i+1) = (S[i], x[i], S[i] + pocc(sh(S[i]), x[i - S[i]\,..\,|x|-1]))$.

The time linearity comes from the fact that at each iteration of the while loop of lines 6–8 an external transition is created and there can be only $|x|-1$ such transitions in $\mathcal{O}(x)$. The loop of lines 2–11 runs exactly $|x|-1$ times and all the other instructions take constant time.

The space linearity comes from the fact that the Factor oracle needs linear space, so does the array S.

Question. Show the Factor oracle $\mathcal{O}(x)$ can be used for locating all the occurrences of x in a text, despite the oracle may accept words that are not factors of x.

[**Hint:** The only word of length $|x|$ recognised by $\mathcal{O}(x)$ is x itself.]

Solution

A solution mimicking KMP algorithm is possible but a more time-efficient solution use the Boyer–Moore strategy. To do so, we use the Factor oracle of x^{R}, the reverse of x. A window of length $|x|$ slides along the text and when the whole window is accepted by the oracle a match is detected, since the window contains x as said in the hint.

When a mismatch occurs, that is, when a factor au of the text is not accepted by the oracle, au is not either a factor of x. Then a shift of length $|x-u|$ can be safely performed. The following algorithm implements this strategy. It outputs the starting positions of all the occurrences of x in y.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press

```
BackwardOracleMatching(x, y non-empty words)
 1  (Q, δ) ← OracleOnline(x^R)
 2  j ← 0
 3  while j ≤ |y| − |x| do
 4      (q, i) ← (0, |x| − 1)
 5      while δ(q, y[i + j]) is defined do
 6          (q, i) ← (δ(q, y[i + j]), i − 1)
 7      if i < 0 then
 8          report an occurrence of x at position j on y
 9          j ← j + 1
10      else j ← j + i + 1
```

Notes

The notion of Factor oracle and its use for text searching is by Allauzen et al. [5] (see also [79]). Improvements given in [109, 111] lead to the fastest string-matching algorithms in most common applications.

The exact characterisation of the language of words accepted by the Factor oracle is studied in [182] and its statistical properties are presented in [40].

The oracle is used to efficiently find repeats in words for designing data compression methods in [173].

The data structure is well suited for computer-assisted jazz improvisation in which states stand for notes as it has been adapted by Assayag and Dubnov [17]. See further developments of the associated OMax project at `recherche.ircam.fr/equipes/repmus/OMax/`.

https://doi.org/10.1017/9781108835831.005 Published online by Cambridge University Press