Real-Time Indexing over Fixed Finite Alphabets

Amihood Amir[1]and Igor Nor[2]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, ISRAEL, +972-3-531-8770, Fax +972-3-736-0498, email: amir@cs.biu.ac.il; and and College of Computing, Georgia Tech, Atlanta, GA 30332-0280. Partly supported by NSF grant CCR-01-04494 and ISF grant 35/05.

[2] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, ISRAEL, email: norigo@cs.biu.ac.il; Partly supported by ISF grant 35/05.

Abstract. The quest for a real-time indexing algorithm is over three decades old. To date there is no convincing understandable solution to this problem. This paper provides a real-time indexing algorithm over a constant-sized alphabet. Assuming the text is arriving at a constant rate, the algorithm spends $O(1)$ time on every text symbol. Whenever a length m pattern is given, the algorithm decides in time $O(m)$ whether there is an occurrence of the pattern in the text thus far.

Keywords: Pattern matching, suffix tree, real time, quasi real time.

1 Introduction

Indexing is one of the most important paradigms in searching. The idea is to preprocess the text and construct a mechanism that will later provide answer to queries of the the form does a pattern P occur in the text in time proportional to the size of the *pattern* rather than the text. The suffix tree [11, 8, 10, 2] and suffix array [7, 5] have proven to be invaluable data structures for indexing.

One of the intriguing questions that has been plaguing the algorithms community is whether there exists a *real time* indexing algorithm. An algorithm is *on-line* if it accomplishes its task for the ith input without needing the $i+1$st input. It is *real-time* if, in addition, the time it operates between inputs is a constant. While not all suffix trees algorithms are on-line (e.g. McCreight [8], Farach [2]) some certainly are (e.g. Weiner [11], Ukkonen [10]). Nevertheless, the quest for a real-time indexing algorithm is over 30 years old. In addition to the fact that this problem has been stubbornly open for over three decades, it has always had an important practical motivation. Imagine sitting on an input data stream and receiving every once in a while crucial key words that need to be found immediately in the data (e.g. a telephone number of a short call currently made that needs to be located immediately for security reasons). All current suffix tree algorithms are linear in the *amortized* sense. This means that we may find ourselves looking for the key word when the data needs linear time to **adjust** in the tree. This linear time is in order of the **entire** input stream, and not in order of the key word, thus by the time the word is found it may be too late!

The frustrating part of this quest is that the community has been titillated by reports of such algorithms during all that period. Slisenko [9] apparently claimed such a construction. Galil attempted to peruse his voluminous paper but had been stymied [3]. Twenty years later, Kosaraju presented an outline of a *quasi real-time* algorithm [6] for suffix tree construction. In the quasi real-time model, the text is input sequentially and processed on-line. At a certain point the text ends and then the pattern is input sequentially and processed on-line. When the pattern ends, the locations where the pattern occurs in the text are output. Nevertheless, Kosaraju's algorithm **can not handle** a continuous text that keeps arriving while patterns are repeatedly sought.

All above attempts dealt with strings over a fixed finite alphabet. It should be noted that to-date there does not exist a linear-time off-line indexing algorithm over infinite alphabets. In fact, using the known techniques (suffix trees and suffix arrays) such an algorithm is impossible because one can sort using the suffix trees and arrays. There exist algorithms that can process a text stream over an infinite alphabet in time $O(\log n)$ per symbol, where n is the number of text symbols input thus far. The algorithm subsequently decides in time $O(m + \log m)$ whether a length m pattern occurs in the text [4, 1].

In this paper we present a real-time indexing algorithm. Our model is the following. Given a text T over fixed finite alphabet Σ, whose symbols are input sequentially at a constant rate. We process the text spending $O(1)$ time per symbol. During the time the text is oncoming, we answer the following queries: Given a pattern P over Σ, decide, in time $O(m)$, whether the pattern occurs in the part of the text already read or not.

The main idea of the algorithm is a novel method for suffix tree construction. For the sake of clarity, we first present this new idea as an off-line algorithm in section 2. We then show how this construction can be adapted to real-time by fairly standard de-amortization techniques (Section 3). In reality, the tree is not fully constructed in real-time, nevertheless, there is sufficient information to decide in time $O(m)$ whether a length m pattern occurs in the text, whenever such a pattern is input.

2 The New Construction Algorithm

In this section we explain the new suffix tree construction algorithm as an off-line algorithm. We will show that the total time complexity of the new algorithm is $O(n)$ for a fixed finite alphabet. While this in itself is not new, the de-amortization of this algorithm will be fairly straightforward. However, presenting it as an off-line algorithm will enable us to more easily explain some of the basic ideas that drive the real time algorithm.

The algorithm is quite complex and has many details. We will present it in a top-down level. We start with a high level overview of the algorithm.

Since the backbone of the algorithm is a suffix tree construction, we start by a brief definition. A *suffix tree* of string S is a contracted trie of all suffixes of S. We denote the suffix tree by ST. ST keeps changing as S is being input in real time. In the high-level description we will use three atomic operations: *testing*, *candelabra cycle*, and *insertion*. In the *testing* operation, we test whether a given substring can be found in the suffix tree constructed thus far. Details of this operation are found in section 2.3. In the *insertion* operation, we insert new suffixes into the suffix tree. The details of this operation are found in section 2.4.

The candelabra cycle employs a structure very similar to the one defined by Kosaraju [6]. However, the order of constructing the candelabra structure, as well as the insertion are novel. These order schemes are the power that enables the real-time, and the reason that the Kosaraju algorithm did not achieve real-time indexing. It is the insertion order that proved to be the major stumbling block to achieving real-time indexing. The employment of the candelabra procedure as well as the insertion are both subtle and non-trivial.

In the ensuing sections, let S be the string we are processing over fixed finite alphabet Σ. Denote by $S[i,j]$ the substring of S between locations i and j, i.e. $S[i]S[i+1]\cdots S[j-1]S[j]$.

2.1 The Candelabra Cycle

We describe the *candelabra cycle with pivot i*. Suppose we have read all inputs up to and including symbol i. We have also constructed a suffix tree of some prefix of the string input so far, call this partial suffix tree ST. We are now presented with symbol $i+1$. We test whether $S[i,i+1]$ appears in the suffix tree. If it does, we read the next symbol $(i+2)$ and check whether $S[i-1,i+2]$ is in the suffix tree. Again, if it is, we read the $i+3$ symbol and check whether $S[i-2,i+3]$ is in the tree. As long as the answer is positive, we read the next symbol and increase the substring checked by one symbol to the left. Suppose following u_1+1 steps the test fails. A failure is defined as a result of one of two criteria:

1. We encounter a substring which is not in the tree. The situation is that $S[i-u_1, i+u_1+1]$ does not appear in ST while $S[i-u_1+1, i+u_1]$ (for $u_1 > 0$) does appear in the tree.
2. The suffix starting at $S[i-u_1]$ had already been inserted to ST in a previous cycle.

We denote by u_1+1 the *size* of the *candelabra cycle.*

This procedure is illustrated in Figure 1.

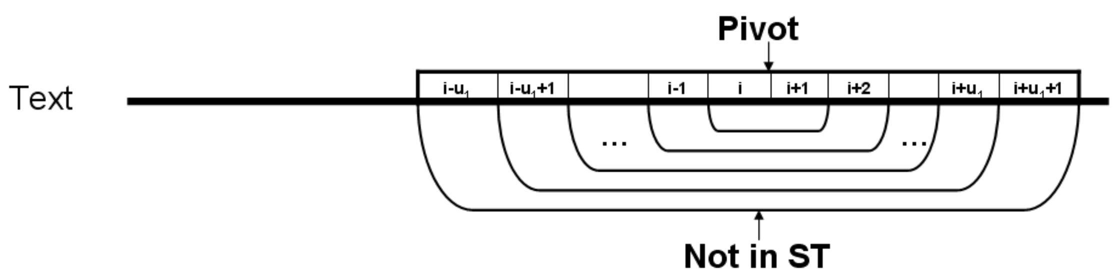

Fig. 1. The input processing (*candelabra* procedure).

Insertion: Once the cycle of the candelabra procedure fails, we insert all suffixes between the rightmost suffix that had already been put in ST and the suffix starting at $S[i-u_1]$. The precise order of insertion of these suffixes, which is the critical contribution of this algorithm, will be elucidated later. Of course, if the candelabra process stopped as a result of criterion 2 (the suffix starting at $S[i-u_1]$ is already in), then there are no suffixes to insert.

2.2 High-Level Description of the New Algorithm

Initialization: We start with an empty tree ST. The first cycle has pivot 0. Input the first two symbols in S $(S[0]S[1])$ and test if this substring is in ST. This test will, obviously fail, since ST is empty, thus the suffix starting at $S[0]$ is inserted in the tree.

Convention 1

1. *We will henceforth say* suffix $S[i]$ *to mean "the suffix starting at* $S[i]$*".*
2. *The suffix insertion follows the line of Ukkonen's algorithm [10], where an edge has a pointer to a starting position and an end position in the string. In case of an edge leading to a leaf the end position is denoted by a "∗", which means "till the end of the string". This "end" dynamically changes as more symbols are added, but the ending position need not change as long as this edge represents the suffix from this starting position till the end of the string.*

New Candelabra Cycle: After the input of $S[i+u_1+1]$, where the test of $S[i-u_1]\cdots S[i+u_1+1]$ failed, and the insertion of the appropriate suffixes, the situation is as shown in Figure 2.

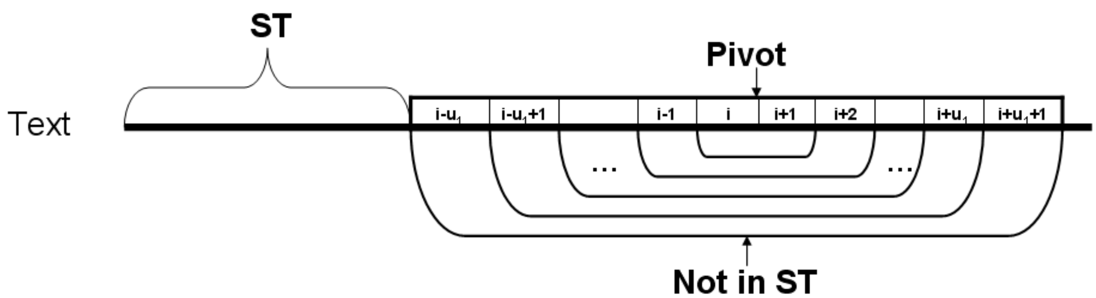

Fig. 2. After candelabra cycle with pivot i.

A new candelabra cycle is started at pivot $i+u_1$ until it fails, i.e. $S[i+u_1-u_2+1]\cdots S[i+u_1+u_2]$ is in the tree but $S[i+u_1-u_2]\cdots S[i+u_1+u_2+1]$ is not. The situation now is illustrated in Figure 3.

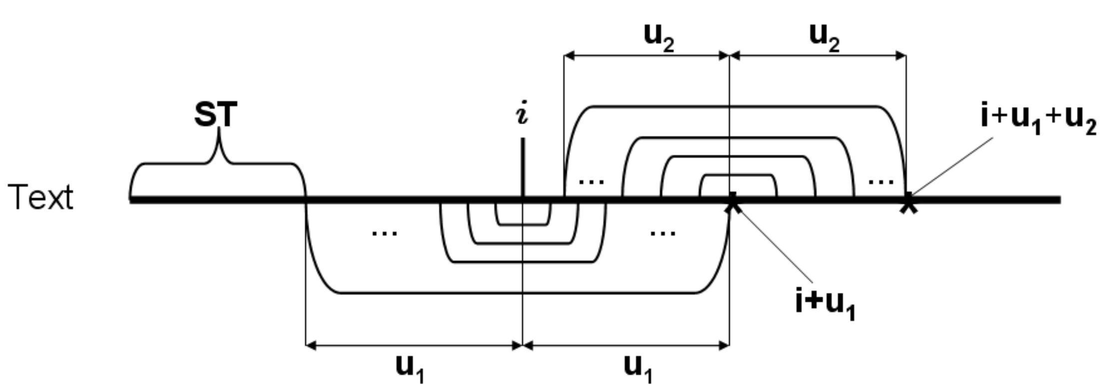

Fig. 3. The candelabra cycle with pivot $i+u_1$.

We insert all suffixes from $S[i-u_1+1]$ till $S[i+u_1+1-u_2]$, inclusive. The situation after the insertion is illustrated in Figure 4.

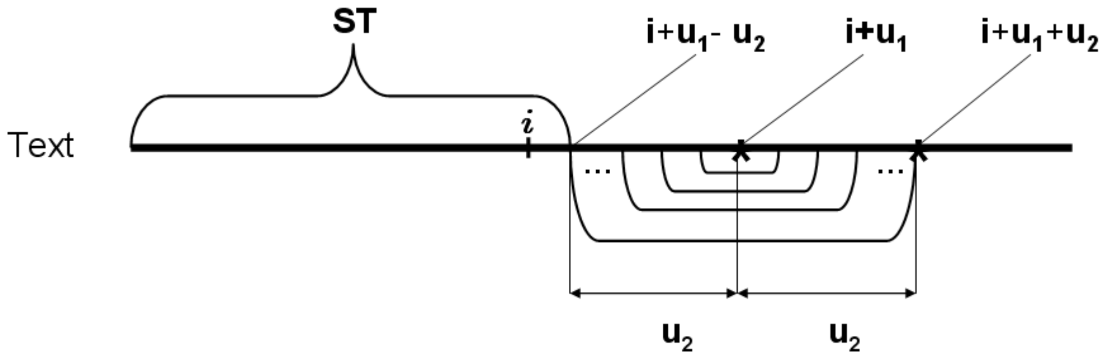

Fig. 4. After candelabra cycle with pivot $i + u_1$.

Note: In case the candelabra cycle failed because we have tested all suffixes until $S[i - u_1]$, which is already in ST (criterion 2), then the entire candelabra with pivot i is subsumed, and we are trivially ready for a new candelabra cycle with pivot $i + u_1 + u_2$. Checking for this criterion can be easily done by keeping the index of the rightmost suffix in ST and comparing every tested suffix to this index. Since this case is degenerate and simple to handle, we will not be considering it any longer. All our efforts will henceforth be devoted to the case where we need to insert new suffixes.

Example:

For an example run of the candelabra procedure see Figure 8.

2.3 Testing

We describe the process of testing in time $O(1)$ per test. At this point, some recollection of suffix tree details are necessary.

Useful definitions

1. **Suffix Trees**

   As is common in suffix tree definitions, the edges are represented by two integers which indicate the starting and the end point of the text. The only exception is for the edges leading to leaves. Those edges are represented by one integer as the starting index of the edge and $*$ as the ending index. This representation allows to perform the construction dynamically, without the need to update the end point after every character addition at the end of the text.

2. **Suffix Links**

   The well-known linear-time suffix tree construction algorithms of Weiner, McCreight and Ukkonen use additional links between tree nodes, denoted as *suffix links*. The different algorithms use different types of suffix links. In our algorithm both types will be used:

   Definition 1. Suffix link*: Let v be an internal node with path-label α. If there is another node $s(v)$ with path-label $x\alpha$, then a pointer from v to $s(v)$ is called an x* suffix link*.*

Definition 2. Inverse Suffix Link*: Let v be an internal node with path-label $x\alpha$. If there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called an* inverse suffix link.

The suffix link was defined in Weiner's construction algorithm and must be associated with some alphabet character, while the inverse suffix link, used in McCreight's and Ukkonen's construction algorithms, is unique for any $x\alpha$ and α paths. As mentioned above, our algorithm will use both types of suffix links. However, there is an important issue to discuss which is the explicitness of those links.

Since the insertion order of the suffixes might change during the construction, in some cases both types of the suffix links might not appear explicitly. This can never happen in the Weiner and McCreight algorithms, but we insert suffixes in a different order from the sequential and thus in our algorithm it may happen, for example, that a node v has path-label α, while the path $x\alpha$ appears in the "middle" of an edge. In this case the algorithm will still keep the suffix link for the node but implicitly – it will point to the closest node below the location. In order to enable identifying the exact location of the suffix link, the suffix link will store the offset as an additional information.

Test Details

A candelabra cycle always starts from testing a substring of length 2, the second symbol is the new input symbol and the first is the pivot. The test whether this substring appears in ST can be done in constant time by checking the suffix tree from the top.

Assume that the last substring successfully tested was $S[i-j, i+j+1]$. We have now added a symbol before and a symbol after the substring, and need to test $S[i-j-1, i+j+2]$. We use the suffix link to test if the extension on the left is in the suffix tree, and then move down the path of the extension to check if the symbol extended on the right is in the tree.

Specifically, if $S[i-j, i+j+1]$ ended in a node then we follow the $S[i-j-1]$ suffix link from that node. If it is on an edge, then assume it is ℓ symbols from its parent node. Follow the $S[i-j-1]$ suffix link from the parent node and move down ℓ symbols. If this was successful we now have that $S[i-j-1, i+j+1]$ appears in the suffix tree and we only need to check if the extension by adding $S[i+j+2]$ is in the tree. This is done by following the path of $S[i-j-1, i+j+1]$ down one more symbol and checking if it $S[i+j+2]$. If either there is no suffix link or the path can not be extended, we have failure and the candelabra process stops.

Time complexity: During the input processing of the algorithm, two actions are taken for each new text symbol: $S[i-j, i+j+1] \Longrightarrow S[i-j-1, i+j+1]$ and $S[i-j-1, i+j+1] \Longrightarrow S[i-j-1, i+j+1+2]$.

Both steps should succeed in order for us to declare success.

Obviously, walking upward to the closest node takes constant time. In case of a suffix link failure, which is noted in constant time, the entire extension is fail. Otherwise, the first node must have a suffix link since we assumed suffix tree correctness. The following well-known lemma shows that moving down ℓ symbols also takes constant time.

Lemma 1. *On the path of ℓ characters below the target node of the suffix link there are no suffix tree nodes.*

Proof: Denote $S[i-j, i+j+1]$ as α. Then $S[i-j-1, i+j+1]$ is $S[i-j-1]\alpha$. Assume such a node exists and denote the path to this node be $S[i-j-1]\beta$. This means there are at least two different suffixes with path $S[i-j-1]\beta\gamma$ and $S[i-j-1]\beta\delta$, each of length no greater than ℓ. Because of the suffix tree correctness, this means that $\beta\gamma$ and $\beta\delta$ of length no greater than ℓ exist. But this contradicts the fact that α, whose prefix is β, is of length ℓ. □

We conclude that every extension step takes constant time.

The total number of tests is equal to the number of test symbols, which is n, therefore the total complexity of the candelabra process in the entire algorithm is $O(n)$.

2.4 The Insertion Technique

Recall some more suffix tree definitions.

More Suffix Tree Notations:

Suppose that suffix Suf_i is being inserted and suppose that $S[i, i+t]$ appears in the tree, while the string $S[i, i+t+1]$ doesn't appear in the tree. Call $S[i, i+t]$ the *head* of the string and denote it by $head_i$. In this case $t+1$ will be the *insertion depth* of Suf_i.

Denote a *set* of sequential suffixes with the first suffix at location i and the last suffix at location j as $Set(i, j)$. The algorithm will be also using another term which is a *sequence* that is a subgroup of the set. A sequence of suffixes located between i and j will be denoted as $Seq(i, j)$.

Definition 3. *Let i be a pivot of a candelabra procedure, and $i+u_1$ the pivot of the next candelabra procedure which fails at $S[i+u_1-u_2+1, i+u_1+u_2]$. Then Set_i is the set of suffixes $\{S[i-u_1+1,*], ..., S[i+u_1-u_2+1,*]\}$.*
We define the indices $i+u_1+1, ..., i+u_1+u_2$ to be the end interval *of Set_i. See Figure 5 for an illustration.*

We will prove that the head of any suffix in Set_i ends in the end interval of Set_i.

Once the suffixes of Set_i have been inserted, we have a complete $ST(\leq i+u_1)$ and we continue with the input processing stage. We would like to insert the suffixes of Set_i into the suffix tree from right to left. This can not be done in linear time since the exact location of the heads is not known. An alternate possibility is to insert the suffixes from left to right in a way similar to the one used by Ukkonen [10]. while this will give an amortized linear-time algorithm, Ukkonen's algorithm does not seem to de-amortize. We therefore need an additional definition - the *sequence*.

Definition 4. *Let i be a pivot of a candelabra procedure, and $i+u_1$ the pivot of the next candelabra procedure which fails at $S[i+u_1-u_2+1, i+u_1+u_2]$. Then the first sequence, Seq_1 is the set of suffixes $S[i+u_1-u_2+1,*], S[i+u_1-u_2,*], S[i+u_1-u_2-1,*], ..., S[i+u_1-u_2-t,*]$, where $S[i+u_1-u_2+1, i+u_1+1], S[i+u_1-u_2, i+u_1+2], S[i+u_1-u_2-1, i+u_1+3], ..., S[i+u_1-u_2-t+1, i+u_1+t+1]$ are in the suffix tree ST and $S[i+u_1-u_2-t, i+u_1+t+2]$ is not in ST. This process is similar to the candelabra process and is illustrated in Figure 6.*
We define the indices $i+u_1+t+2, ..., i+u_1+u_2$ to be the end interval *of Seq_1. See Figure 7 for an illustration. The heads of all suffixes in Seq_1 end in its end interval.*

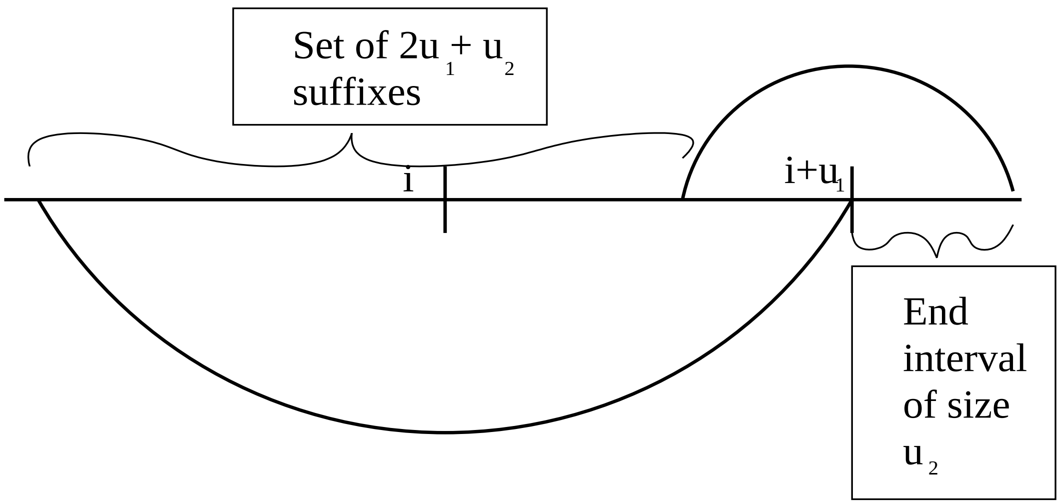

Fig. 5. Set definition.

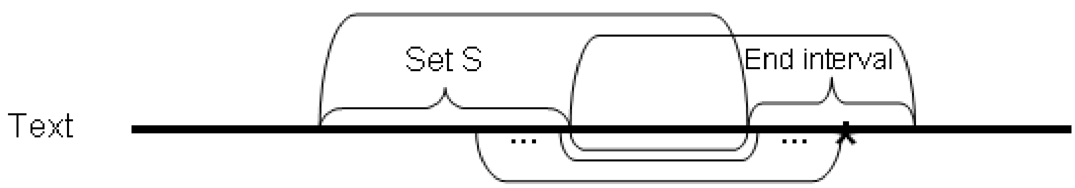

Fig. 6. Sequence definition.

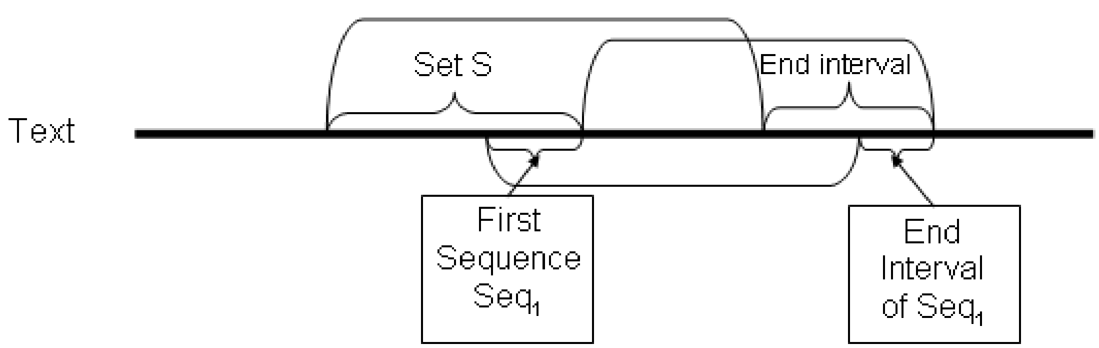

Fig. 7. Sequence definition and end interval.

Implementation Note: Testing the first sequence can be done while the candelabra cycle with pivot $i + u_1$ is taking place. It is easy to find the exact head location of the first sequence suffix, since we already know the location of $S[i + u_1 - u_2 - t + 1, i + u_1 + t + 1]$ and its possible head location is at least at index $i + u_1 + t + 1$. Following the insertion of the first suffix of the sequence, use the inverse suffix link in order to insert the remaining t suffixes in left to right order.

After these suffixes are inserted we define the second sequence in a manner similar to the first. We test whether $S[i+u_1-u_2-t-1, i+u_1+1]$ is in the suffix tree. We continue enlarging the substring on both sides and testing until we fail, say after t_1 steps. This defines the second sequence whose range is $(i + u_1 - u_2 - t - t_1, i + u_1 - u_2 - t)$ and its end interval is $(i + u_1 + t_1 - 1, i + u_1 + t - 2)$. We insert the suffixes of the second sequence from left to right as described above. Similarly we define all sequences from right to left, and insert their suffixes (from left to right), until the set is exhausted.

An important property of the suffixes insertion is the fact that after inserting the suffix and creating a suffix tree leaf, we can be sure that there will be no need to come back and fix it. The justification for this property are the lemmas in the Ukkonen construction algorithm. These lemmas proved that a leaf remains a leaf throughout the entire suffix tree construction.

During the insertion process, the inverse suffix links are used in order to locate the next sequence suffix. Following the creation of the new nodes it might be necessary to update some links and add the correct links to the newly created suffix tree nodes, so that the suffix tree will remain correct. We briefly describe this process.

As was explained above, the insertion processing is a sequential insertion of sequences which contains the following basic processes:

1. Discover a set sequence.
2. Insert the sequence suffixes from left to right in the following way:
   (a) Find the location of the head of the first suffix in the sequence.
   (b) Use inverse suffix links in order to move from the previous suffix to the next.
   (c) Explicitly check characters in order to find the exact location of the head.
   (d) Update the relevant suffix links in case of new suffix tree node creation.

We now explain the suffix insertion details.

Lemma 2. *For the entire set of suffixes with end interval of size u, the number of performed extensions is $O(u)$.*

Proof: Every set of suffixes is inserted sequence by sequence, where all sequences end locations have no intersection, so the total number of the possible end locations is u. Every sequence is inserted left to right, so using the proved fact that $S[i, j]$'s existence in suffix tree causes $S[i + 1, j]$'s suffix tree existence, one can claim that any symbol that extended the current suffix will extend the next suffix and thus need not be tested. Consequently, there are at most u total number of extensions. □

However, by finishing the current suffix insertion and using the inverse suffix link, there is no certainty that the correct location will be achieved in constant time. The problem is that given the head location, it is possible that the node will not have a correct suffix link. Thus it might be necessary to move upward to the closest suffix tree node which has an inverse suffix link. Despite the fact that this suffix link must exist, by using it we might jump to a point from which it will take more than constant time to get to the destination. Formally speaking, denote the head location as $S[i,j]$, the closest node might appear k characters upward from this location, at $S[i,j-k]$. Thus, using the inverse suffix link, the new location $S[i+1,j-k]$ will also be k characters down from the node to which the suffix link pointed.

The first step toward the solution of this problem is to use an idea of the known suffix tree construction algorithms, which Ukkonen calls the skip/count solution. Since we know that $S[i+1,j]$ appears in the tree, this location can be reached by skipping between nodes without checking characters explicitly. The only thing needed to be checked is the number of characters on the skipped edge, and then subtract this number from k. Continue until the suffix tree node that has the sought location on its outgoing edge, is reached.

However, this process might take $O(k)$ steps, which is not necessarily constant.

The next lemma, similar to one proved in [10] shows that this process takes linear amortized time.

Lemma 3. *For the entire set of $v-u$ suffixes with end interval of size u, the number of nodes traversed by the algorithm is $O(v+u)$.*

It remains to show how to keep the suffix tree correct from the suffix links point of view. In other words, we need to show that after a new node creation, it is possible to fix suffix links of all types, including creating the correct suffix links for the new node. The solution is similar to the suffix links updates of each of the known algorithms. It is necessary to update all types of suffix links. Generally, there are four types of suffix links that might change as a result of a new node insertion:

1. Suffix links outgoing from the new node N. In this case one can use suffix links information of the closest node above N. Denote that node by N_1. Suppose that N_1, associated with character $c \in \Sigma$, points to some node SN_1. SN_1 must be located above the location of the link of the new created node, otherwise its suffix tree length will overreach the length of N. By one of the previous lemmas, SN_1 must be the closest node above, otherwise there should be a closer node above N than N_1. As a result, the exact location of the correct suffix link will be on the suffix tree edge, outgoing of SN_1, which can be found in $O(1)$ time.
2. Inverse suffix link incoming to the new node. This incoming link can be found using the fact that the node that should point to the new node was used in the previous iteration of the insertion. Thus, the only necessary change is to keep a pointer to the previous insertion step node and update its inverse suffix link to the new created node.
3. Suffix link incoming to the new node. This kind of changes might be the most problematic, since there can be a lot of implicit suffix links, that pointed to the node below the new created node and now they have to change their links. However, the next step of the algorithm will be to use the inverse suffix link and fix the suffix links in the opposite direction. The algorithm's order of processing allows to fix the suffix links, since nodes that might need the suffix link changing are exactly the same nodes that are being skipped during the skip/count processing. Thus, while skipping nodes one can also change their outgoing suffix links to the new created auxiliary node that will be created with the new created suffix tree node.
4. Inverse suffix link outgoing of the new node. This case will be solved with a similar idea as the previous one. While traversing the skipping nodes we reach the required location or possibly

create a new node. The correct outgoing inverse suffix link should point to this node, so by the end of the next suffix insertion, this outgoing inverse suffix link will be updated.

2.5 Time Complexity

We showed that the time of all described processes is linear in the input size. It includes reading the input, processing it by performing string tests, discovering set and sequences, finding the rightmost problematic sequence, inserting the entire set and finally fixing suffix links of all types after a new node creation. Thus, the entire construction algorithm is also a linear time.

3 Real-Time Construction

In this section we will show how to de-amortize the linear time construction algorithm, creating a real-time indexing algorithm.

The algorithm is based on the idea of the construction algorithm technique of inserting suffixes into suffix tree. However, since this algorithm should support real-time indexing, some changes must be performed, most of them for de-amortizing the processes. First, as was explained above, we insert a set of suffixes by dividing it into sequences. In the linear construction algorithm those sequences were inserted right to left, while the suffixes within the sequence were inserted left to right.

For the real-time algorithm we need first to define the global step of the algorithm. Any global step of the algorithm will primarily contain reading a new character of the text and performing some constant time operations of the input algorithm. In parallel inserting operations will be performed. The purpose is to make those additions be constant time, so that the entire complexity of the global step will still be $O(1)$.

In order to allow this constant time processing, the insertion technique must be improved in two aspects. It was proven in the previous sections that the time complexity of inserting a set of suffixes requires a number of operations linear in the size of the set and the end interval (in our notation, $O(v+u)$). The same is true for any sequence of the set. This means, that there is some constant c, so that the number of operations that should be performed, in order to complete the set insertion, will be $c(v+u)$. We will perform $3c$ steps on the current sequence at any global step. This improvement allows to claim the next important lemma:

Lemma 4. *Assume there is a sequence $suf_\ell, suf_{\ell+1}, \cdots, suf_r$ of suffixes, located within some set S, and assume w is the number of the set suffixes located to the right of this sequence. Then following k global steps and $3ck$ operation on the sequence, each suffix with head of length $< u' + k$ will be inserted into the suffix tree, when $u' = u + w$.*

Proof: See Appendix, Subsection 6.3.

In the offline algorithm, every set is inserted sequence by sequence and the algorithm starts discovering and inserting the next sequence only once the previous sequence had been completed. The previous lemma proved that the size of the heads of suffixes that are not inserted after k global steps on a sequence, each of $3c = c'$ operation, is at least some minimal size. In order to make it real-time we need to add another improvement, and instead of performing c' operations on the current sequence, we will perform $3c'$ operations at any global step: c' of them on the current sequence, and additional $2c'$ operations on the next sequence. Certainly, it is impossible to perform operations on the next sequence before it was defined, so we will start to perform operations immediately after the current sequence was defined. The next theorem will prove that this processing leads to real-time indexing.

Theorem 1. *Suppose we perform the insertion algorithm on some set of suffixes. Following k insertion steps, with a constant operations number for each, any suffix i with $|head_i| \leq u + k$ will be inserted into the suffix tree.*

Proof: See Appendix subsection 6.3.

The purpose now is to use this theorem for a real-time algorithm by changing the construction algorithm. The new algorithm will process the input using the same technique of input processing. The difference will be at the insertion algorithm where instead of insertion of a new discovered set at one fling, the algorithm will start processing those suffixes one by one in constant time, while at the same time it will continue reading the input. Since both processes are real-time, as it was shown in the previous theorem, the entire processing will also be constant time for each step.

However, there is a concern regarding this order changing which is to insure that operations being performed on the new input symbols will not be affected by the different insertion order. More precisely, since the linear time construction technique is based on the fact that all left located suffixes have been already inserted, we need to insure that insertion order changing will be transparent for both input and insertion processes of other suffixes.

First we need to define the number of operations per step. In the previous theorem we performed $3c' = c''$ operations on each global step. Using the linear algorithm we can also claim that we need $< c''(v+u) = O(v+u)$ operations for inserting any set of size $v-u$ and end interval of size u. Thus, for every step we have to perform at least c'' operations. The improvement here is to multiply by a constant the number of insertion suffix steps. More precisely, for each new input symbol we will perform $3c''+2$ steps of inserting suffixes on the rightmost unfinished set of suffixes.

In order to prove the correctness we need some lemmas.

Lemma 5. *Consider m sets of suffixes, denoted $Set_1, Set_2, \cdots, Set_m$ and suppose that the first set is located at index ℓ and the last set ends at location r. Then during the input processing of the new input symbols and testing string of size t, all suffixes with head length $\leq t$ will already be inserted into the suffix tree.*

Proof: See Appendix subsection 6.3.

Lemma 6. *Assume suffix s, located in Set_j, is being inserted. Then the head length of every suffix located at Set_i, $i < j$, which was not inserted, is $> head_s$.*

Proof: See Appendix subsection 6.3.

As a conclusion of the real time algorithm, one can claim that the new construction algorithm is a real-time algorithm, since it requires $3c''+2$ operations for each global step. The algorithm keeps its linear time total complexity, since the real-time solution performs the same operations in a different order, thus it doesn't spoil the total complexity.

4 Pattern Matching Problem

In this section we will explain the use of the suffix tree for real-time indexing. Assume there is a Pattern P of size m and we want to process the pattern matching of this pattern with the real-time processed text.

The ability to perform this pattern matching follows directly from the previous lemmas. Since following t steps any suffix of length t will be inserted into the suffix tree, following m steps of searching in the suffix tree, all relevant suffixes will be already inserted. Thus the result whether given Pattern appears in the suffix tree will be correct.

5 Conclusion and Open Problems

We presented the first fully proven and coherent real-time indexing algorithm over a fixed finite alphabet. The most important remaining open problem is whether one can process a text in real-time and produce **all** occurrences of pattern string P in the text in time $|P| + tocc$, where $tocc$ is the number of occurrences.

It is also open whether indexing over infinite alphabets can be done in real-time. It is clearly impossible using suffix trees or arrays, but nothing is known if an entirely different idea is employed. We conjecture that alphabet independent linear time indexing is impossible.

References

1. A. Amir, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th Symposium on String Processing and Information Retrieval (SPIRE)*, volume 3772 of *LNCS*, pages 67–78. Springer, 2005.
2. M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.
3. Z. Galil. On converting on-line algorithms into real-time and on real-time algorithms for string matching and palindrome recognition. *SIGACT News*, pages 26–30, 1975.
4. R. Grossi and G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. In *Proc. 26th Intl. Col. on Automata, Languages and Programming (ICALP)*, pages 372–381, 1999.
5. Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pages 943–955, 2003. LNCS 2719.
6. S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees. *Proc. 26th STOC*, pages 310–316, 1994.
7. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 319–327, 1990.
8. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.
9. A.O. Slisenko. Recognition of palindromes by multihead turing machines. In *Proc. of the Steklov Math. Inst.*, volume 129, pages 30–202. Acad. of Sciences of the USSR, 1973.
10. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
11. P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

6 Appendix

6.1 Example Run of Candelabra Procedure

6.2 Extending Head Length

Suppose there are two suffixes starting at indices i and j while their longest common prefix $lcp(Suf_i, Suf_j)$ is bigger than the head of both suffixes in the original tree. We claim that in the original suffix tree $head_i = h = head_j$. Otherwise, their lcp would be at most the shorter head of the two suffixes, and smaller than the head length of the other suffix.

Observation 1 *The distance between the locations of $head_i$ and $head_j$ is at most u.*

Proof: Suppose the distance is bigger than u. The heads of both suffixes end in range of u possible locations. On the other hand, both suffixes have the same insertion depth in the original suffix tree, which contradicts the assumption. □

Denote such a pair (i, j) of suffixes with the same head as *head pair*. Consider a head pair (i, j) so that j is the leftmost index over all head pairs. Since $head(j)$ is equal to $head(i)$, any suffix Suf_{j+k} in the set has a common prefix with the suffix Suf_{i+k} of length at least $|head(j)| - k$. Which means that if the head of the leftmost j of all head pairs (i, j) ends outside the interval, the heads of all suffixes to the right of it also end outside the interval.

The idea for solving the extending head length problem is to insert suffixes that end in the appropriate interval and not to insert suffixes whose head will be extended and end out of the range. We will show that these two sets are contiguous, and the set of suffixes whose head ends outside the interval is to the right. Now we can attach the suffixes whose head ends outside the interval to the next set. This is a natural union since the only property of suffixes in the next set is that they end in an interval located to the right of the current set interval.

The purpose, therefore, is to find the first suffix with head out of the interval. By solving those suffixes problem it is possible to continue the original algorithm. More precisely, by finding the t in the previous process, we can claim the next observation:

Observation 2 *Suppose the suffix j that failed in the test following t steps is the right index of some head pair (i, j). Then the left index of the head pair is at most t indexes to the left of j.*

Proof: Since Suf_j failed following t steps, the length of its head is at most $u + 2t$, which causes $head_i = head_j \leq u + 2t$ by previous lemmas. But any suffix located to the left of Suf_{j-t} is at distance at least $u + 2t$ from the beginning of the set end interval. Thus, Suf_i is located at most t indexes to the left of Suf_j. □

As a result, one can claim that the leftmost possible suffix that ends out of the end interval is Suf_{j-t+1}. Using the previous explanations we can claim that the leftmost location of the left component of the head pair of which $j - t + 1$ is the right element, is located at most t indices to the left of index $j - t + 1$.

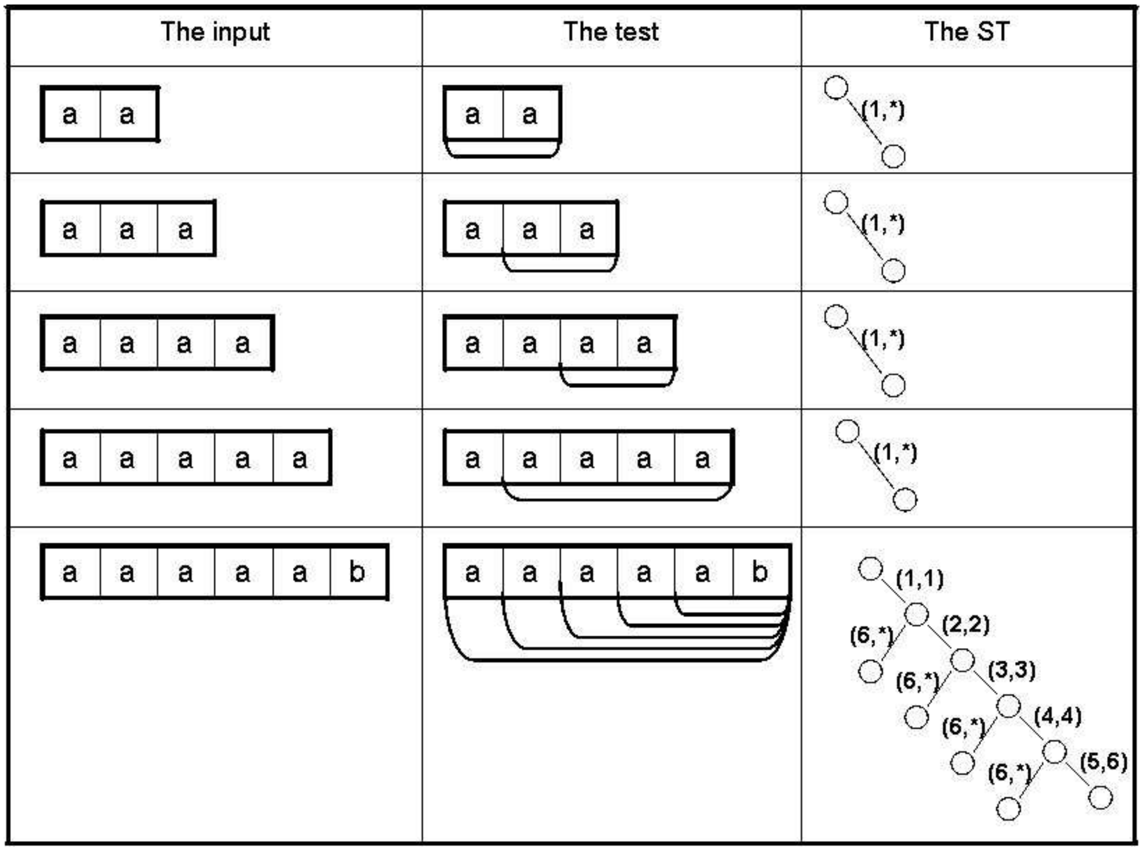

Fig. 8. an example run of the candelabra procedure.

6.3 Proofs of the Real-Time Algorithm

Lemma 7. *Assume there is a sequence $suf_\ell, suf_{\ell+1}, \cdots, suf_r$ of suffixes, located within some set S, and assume w is the number of the set suffixes located to the right of this sequence. Then following k global steps and $3ck$ operation on the sequence, each suffix with head of length $< u' + k$ will be inserted into the suffix tree, when $u' = u + w$.*

Proof: For any given sequence, there are two possible cases:

1. Case $r - \ell > k$. In this case all heads of the sequence suffixes will be of length $> u+k$. The reason is that as a part of sequence discovering, the leftmost suffix of the sequence had been successfully tested on a string $S[\ell, 2r-\ell+u']$. By one of the previous lemmas this means that the head of any suffix in this sequence ends at location $> 2r+u'-\ell$ which is $> r+u'+k$ by the case assumption. As a result, even the rightmost sequence suffix head is of length $> r + u' + k - r = u' + k$.
2. Case $r - \ell \leq k$. Assume, to the contrary, that there are suffixes of head length $< u' + k$ in the sequence that were not inserted. Denote j as the leftmost index of those suffixes.
   Based on the case assumption, the number of sequence suffixes located to the left of j is at most k. But since we assumed that the head size of Suf_j is $< u' + k$, it ends to the left of index $r + u' + k$. Based on the previous lemmas we have that the head of all suffixes located to the left of j, cannot end to the right of this location. This means that the end interval for all those suffixes is of size at most k, since they also cannot end to the left of the set end interval, which starts at location $r + u'$. Using previous lemmas, in order to insert k suffixes with end interval of size k $ck + ck = 2ck$ operations must be performed . However, after k global steps, $3ck$ operations were complete, which means that Suf_j had to be already inserted. This contradicts the assumption that Suf_j was not inserted during k global steps.

□

Theorem 2. *Suppose we perform the insertion algorithm on some set of suffixes. Following k insertion steps, with a constant operations number for each, any suffix i with $|head_i| \leq u + k$ will be inserted into the suffix tree.*

Proof: At each step c' operations are being performed on the current sequence, and $2c'$ on the next one. Note, that the algorithm processing works in such a way that the size of the end interval of any sequence is bound. More precisely, after performing k_1 tests on the first sequence, its end interval will be of size $u - k_1$, where u is the end interval for the entire set. The second sequence will be defined following k_2 tests and its end interval will be of size $k_1 - k_2$. In general, Seq_i will be defined after k_i tests and its end interval will be of size $k_{i-1} - k_i$. But the number of the suffixes in Seq_i is exactly k_i. Thus, based on the above lemmas, the number of operations that must be performed in order to insert all suffixes of Seq_i is $< c'(k_{i-1} + k_i)$.

However, the way the processing is defined, the algorithm performs exactly the number of operations that is needed in order to complete sequences. For example, after finishing the first sequence definition, the new sequence began to be processed. Thus, following k_1 global steps, $c'k_1$ operations will be performed on the first sequence, and $2c'(k_1 - k_1/c') = 2c'k_1 - 2k_1 > c'k_1$ operations will be already performed on the second sequence.

Afterwards, during the next k_2 operations, $c'k_2$ operations will be performed on Seq_2 and after first k_2/c' global steps, $> c'k_2$ operations will be performed on Seq_3. Note that this processing guarantees one important property:

Observation 3 *After k_2 global steps, the insertion processing will complete for this sequence.*

Proof: The property is based on the fact that in parallel to Seq_1 processing, $> c'k_1$ operations will be performed on Seq_2. Now, while Seq_2 is being processed, another $c'k_2$ operations will be performed on it. However, as it was shown before, in order to complete this set insertion, $c'k_1 + c'k_2$ operations must be performed. But this is precisely the number of operations that have been actually performed. $\square$

This property holds on any subsequent sequence, which means that by finishing k_i operations on Seq_i, it is being completed and there is no unfinished sequence left behind, except possibly the first one.

Using this property, consider the number of performed operations on Seq_i. It began to be processed after sequence Seq_{i-1} was defined. As was shown, the sequence definition of Seq_{i-1} requires $< c'k_{i-1}$ operations, thus after less than $c'k_{i-1}$ operations on Seq_{i-1} we will start to perform operations on Seq_i.

Assume a general case with m sequences and k operations that have been performed on the set, while $k = k_1 + k_2 + \cdots + k_{i-1} + \alpha$, where $\alpha < k_i$. Using the previous lemma and observation, one can conclude that the theorem holds for any possible sequence:

1. Seq_1. Each global step c' operations will be performed on this sequence, until all suffixes will be inserted. Thus, using the previous lemma, after k global steps, any suffix with head length of size $\leq u + k$ will be inserted into the suffix tree.
2. Seq_2 to Seq_{i-1}. As was proven in the observation, all those sequences will be completely inserted, thus there is no suffix in there that can contradict the theorem conditions.
3. Seq_i. After α global steps, $> c'\alpha$ operations will be performed on Seq_i, thus using the previous lemma, after k global steps, any suffix with head length of size $\leq u' + \alpha = u + k$ will be inserted into the suffix tree.
4. Seq_{i+1} to Seq_m. The minimal head size for any suffix in those sequences is $u + k_1 + k_2 + \cdots + k_{i-1} + k_i > u + k_1 + k_2 + \cdots + k_{i-1} + \alpha + 1 = u + k + 1$. Thus those sequences cannot contradict the theorem conditions.

$\square$

Lemma 8. *Consider m sets of suffixes, denoted $Set_1, Set_2, \cdots, Set_m$ and suppose that the first set is located at index ℓ and the last set ends at location r. Then during the input processing of the new input symbols and testing string of size t, all suffixes with head length $\leq t$ will already be inserted into the suffix tree.*

Proof: Assume some string of length t is being tested and assume there is a suffix with head length $\leq t$ that is not inserted. Consider the rightmost head of the same size, let it be located at Set_i. Denote the length of two cycles that define the Set_i as u_i and u_{i+1}. As was analyzed in the linear time algorithm, in order to insert any set i of suffixes, $c'' sizeof(cycle_i) + c'' \cdot sizeof(cycle_{i+1})$ operations should be performed, which means in order to insert all suffixes of Set_i, we need to perform $c''u_i + c''u_{i+1}$ operations. Thus, since all sets are located sequentially one by one and each suffix belongs to some set, the time complexity we need in order to insert all suffixes of all sets to the right of Set_i is the sum of operations we need to perform over all sets in the range, which is $c''u_{i+1} + 2c''u_{i+2} + \cdots + 2c''u_{m-1} + 2c''u_m + c''u_{m+1}$. On the other hand, the number of steps we made over all sets $S_{i+1}, \cdots, S_m$ during this processing is $(3c''+2)u_{i+3} + (3c''+2)u_{i+4} + (3c''+2)u_{i+5} + \cdots (3c''+2)u_m + (3c''+2)u_{m+1} + (3c''+2)t/2$, since for any step we perform during discovering the cycle u_{j+2} we are making $(3c''+2)$ operations on the set j and in order to test a string of size t, we need to perform $t/2$ tests. Taking the subtraction between those two expressions, we get that it is equal to $(c''+2)u_{i+3} + (c''+2)u_{i+4} + \cdots + (c''+2)u_m + (2c''+2)u_{m+1} + (3c''+2)t/2 - c''u_{i+1} - 2c''u_{i+2} = \Delta$.

However, based on the previous theorem one can assume that in case of the assumption which is having a string of size t and there is some suffix in Set_i of length $\leq t$ which is not inserted, then $t \geq u_{i+1} + (3c''+2)u_{i+2}$, since $(3c''+2)u_{i+2}$ steps have been already performed on Set_i. This causes $c''t \geq c''u_{i+1} + c''(3c''+2)u_{i+2}$. Using this inequality for the previous expression we have that $\Delta \geq (c''+2)u_{i+3} + (c''+2)u_{i+4} + \cdots + (c''+2)u_m + (2c''+2)u_{m+1} + (c''/2+1)t + c''u_{i+1} + c''(3c''+2)u_{i+2} - c''u_{i+1} - 2c''u_{i+2} \geq t$.

The result guarantees that if there is a set i with some suffix with head size $\leq t$ which was not inserted, we have more than t steps to perform insertions on the Set_i which contradicts the possibility of some head of size t not being inserted. $\square$

Lemma 9. *Assume suffix s, located in Set_j, is being inserted. Then the head length of every suffix located at Set_i, $i < j$, which was not inserted, is $> head_s$.*

Proof: Assume there is such a suffix, located at Set_i, $i < j$. Using the technique of the last lemma, the number of operations that need to be performed on sets in range (Set_{i+1}, Set_{j-1}) is equal to $c''u_{i+1} + 2c''u_{i+2} + 2c''u_{i+3} + \cdots + 2c''u_{j-1} + c''u_j$. On the other hand, the number of performed operations over all sets in this range is $(3c''+2)u_{i+3} + (3c''+2)u_{i+4} + (3c''+2)u_{i+5} + \cdots + (3c''+2)u_j + (3c''+2)u_{j+1}$. Subtracting those two expressions, we get $(c''+2)u_{i+3} + (c''+2)u_{i+4} + \cdots + (c''+2)u_{j-1} + (2c''+2)u_j + (3c''+2)u_{j+1} - c''u_{i+1} - 2c''u_{i+2} = \Delta$.

Since we assumed that there is a suffix in Suf_j with head length larger than some head length in Suf_i, we can assume that $MaxSize_j = 2u_j + u_{j+1} > u_{i+1} + (3c''+2)u_{i+2} = MinSize_i$, since the left expression is the biggest size of some possible head length, and the right expression is the minimal head size of any suffix in the set followed by $(3c''+2)u_{i+2}$ performed operations on it. This causes $2c''u_j + c''u_{j+1} > c''u_{i+1} + c''(3c''+2)u_{i+2}$. Using this inequality for the previous expression we get that $\Delta = (c''+2)u_{i+3} + (c''+2)u_{i+4} + \cdots + (c''+2)u_{j-1} + (2c''+2)u_j + (3c''+2)u_{j+1} - c''u_{i+1} - 2c''u_{i+2} \geq 2u_j + (2c''+2)u_{j+1} + c''u_{i+1} - c''u_{i+1} + c''(3c''+2)u_{i+2} - 2c''u_{i+2} \geq 2u_j + u_{j+1}$.

As a result, in such a case we will have more than $MaxSize_j$ extra steps on Set_i which cause contradiction to the existence of suffix with head size less than $head_s$ in set i. $\square$