An adaptive suffix tree based algorithm for repeats recognition in DNA Sequence

Arya Sharma - 8265911491
Vaibhav Puri - 8529418547
Animesh Anand -9123161695
Vishwajit Kumar Singh - 6203953060

Motivation

A genome is an organism’s complete set of DNA. Genomes vary widely in size: the smallest known genome for a free-living organism (a bacterium) contains about 600,000 DNA base pairs (bp), while mouse and human genomes have about 3 billions base pairs. A genome has a lot of repeats. For example, more than 50% of a human genome is various kinds of repeats. Repeats recognition is a critical part of any analysis of a new sequence genome, because repeats drive genome evolution in various ways and pragmatic needs for thorough repeat mask prior to perform homology searches.

Some human genetic diseases such as the fragile-X chromosome syndrome, Huntington's disease, and Friedreich's ataxia are all related to irregularities of the length of repeats.

An important bioinformatics problem is how to recognize fast and represent efficiently repeats in a genome. There are two major approaches to solve the repeats recognition problems.

Introduction.

- the fast RepSeeker algorithm for repeats identification based on the adaptive Ukkonen algorithm for a suffix tree construction.
- The RepSeeker algorithm uses the lowest frequency limit to maximize the extension of repeats.The information in leaves and branch nodes are different, thus the RepSeeker algorithm can obtain directly needed information from nodes to find out the frequency and locate the positions of a substring.
- A is an elementary repeat of S, if A is a nontrivial substring of S with the maximal length, A occurs in S at least Fm times, and every nontrivial substring of A is a sub repeat of A. Fm is a specified lowest frequency of occurrences of repeats.

- The RepSeeker algorithm first finds all elementary repeats of the input sequence S and then outputs the sorted list of repeats.
- An element of the list of repeats is a pair, which represents the starting position and the ending position of a repeat copy in the input sequence. Hence, the problem of identifying elementary repeats can be converted into the problem of finding boundaries of repeats, the task of the RepSeeker algorithm is to check each position in the input sequence S and determine whether the position is a boundary of a repeat or not.
- It is impractical to use the exhaustive algorithm to find elementary repeats because there are O(n2) substrings in S, and each substring of length m contains O(m2) substrings to be checked. So a suffix tree is constructed from the input sequence S to help count the frequencies

Algorithm steps.

1. Compute the frequency of a substring in a sliding window.
2. Finding the blocks with the same frequency
3. Checking sub repeats
4. Extending repeats
5. Classifying the repeats

Checking Sub-Repeats

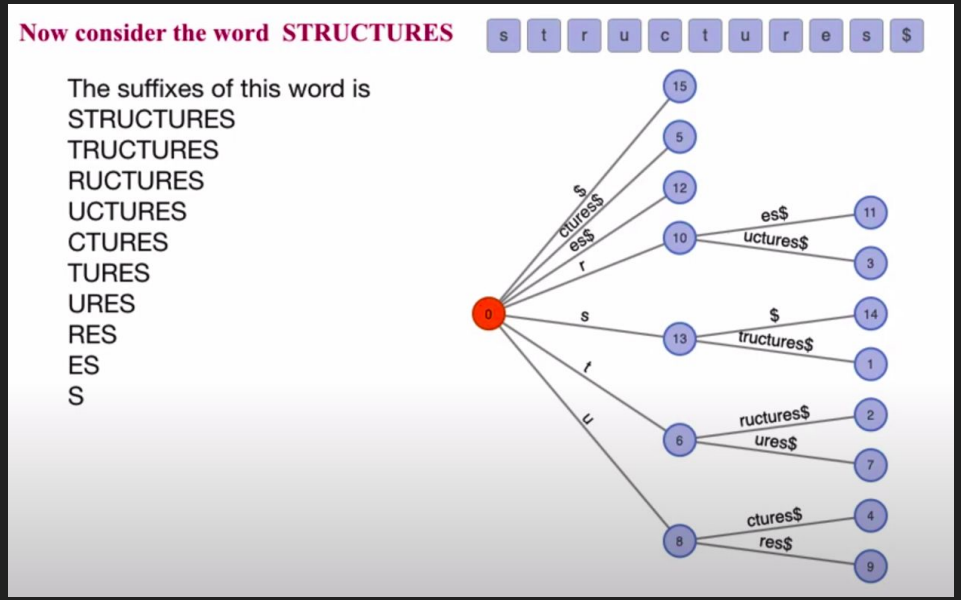
- The goal of this step is to check whether the blocks derived from the first two steps contain non-sub-repeats L-length substrings or not.
- If a block contains a substring which is not a sub-repeat, it means that the block is not a repeat and we need to split it.
- Assume that a block D(i) = S[starting position,ending position] with a frequency k. And all substrings with length L in the block D i have the frequency m.

Extending Repeats

In order to maximize the length of a repeat, we merge the overlapped repeats if the resulting repeats have a frequency at least Fm , the threshold of frequency. The repeats with the higher frequency are still left and the repeats with the lower frequency are extended.

What is a Suffix Tree

Suffix Tree is compressed trie of all the suffixes of a given string,
Example



General Rules

A suffix tree T for a m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Ukkonen Algorithm :

The naive implementation for generating a suffix tree going forward requires
O(n*n) or even O(n*n*n) time complexity , with the help of ukkonen algorithm we can generate suffix tree in O(n) time.

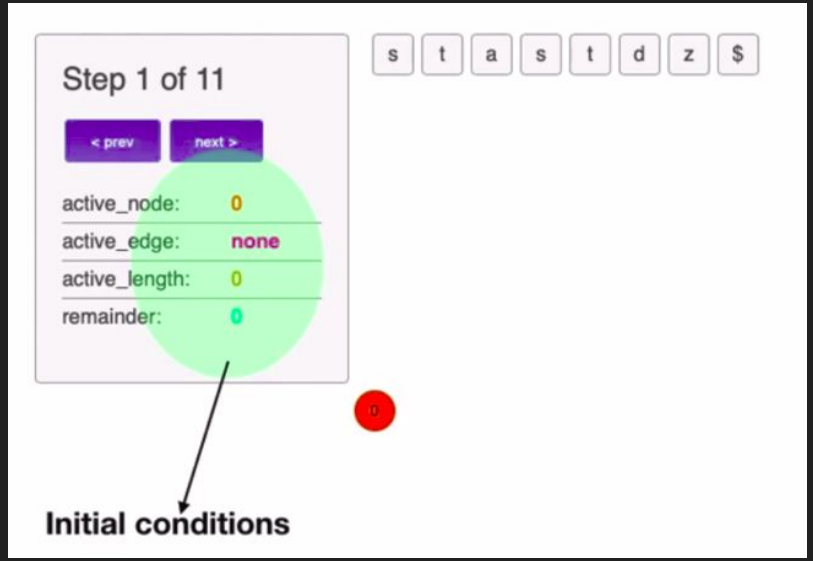
continued….

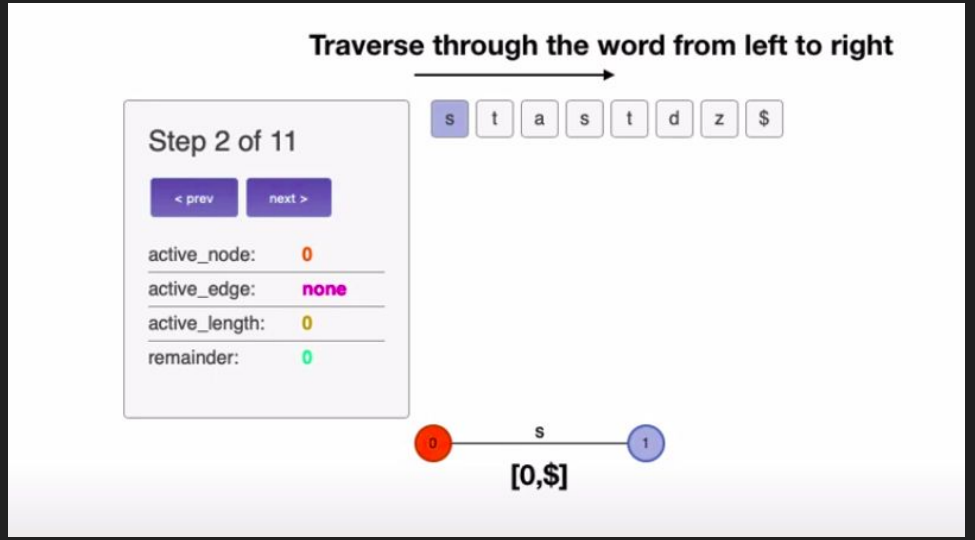
The algorithm begins with an implicit suffix tree containing the first character of the string. Then it steps through the string, adding successive characters until the tree is complete. This order addition of characters gives Ukkonen's algorithm its "online" property.

General structure of a suffix tree

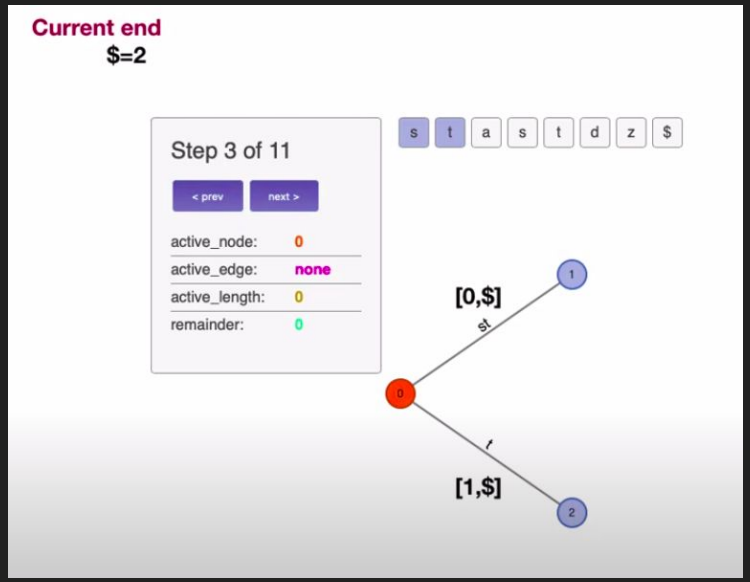
```
def __init__(self, data):
    self._string = data
    self.lastNewNode = None
    self.activeNode = None
    self.activeEdge = -1
    self.activeLength = 0
    self.remainingSuffixCount = 0
    self.rootEnd = None
    self.splitEnd = None
    self.size = -1
    self.root = None
```

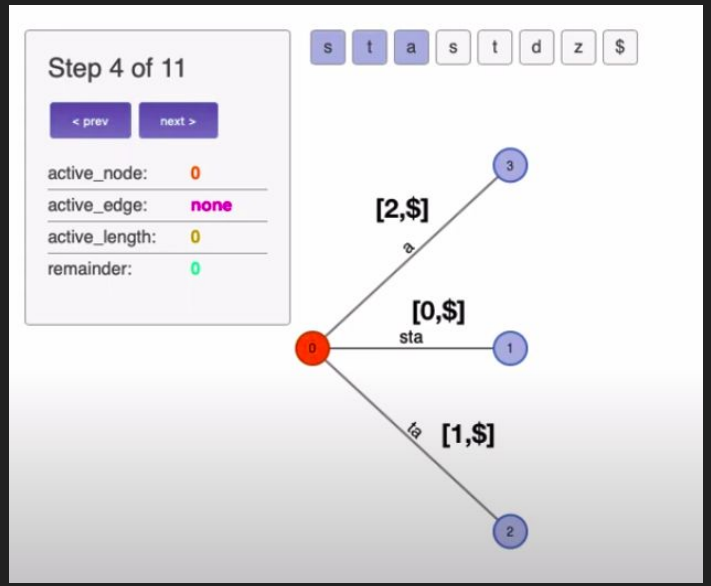
Example : stanstdz



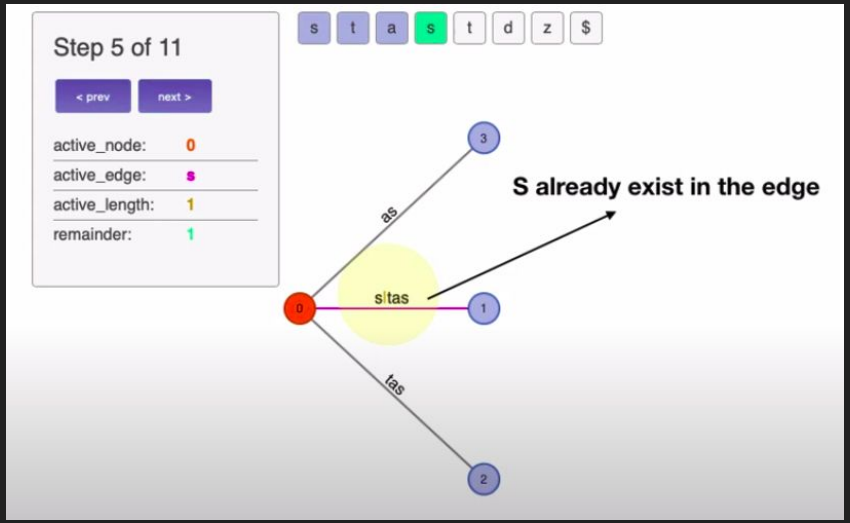


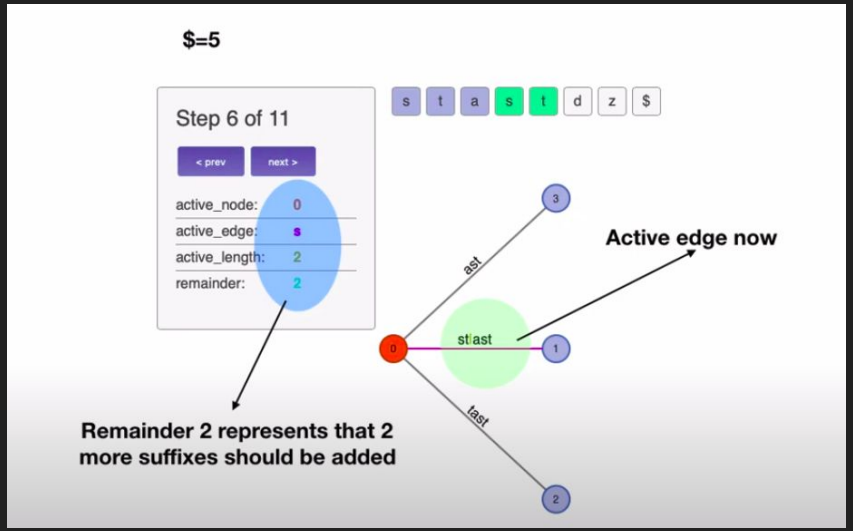
Further simple steps :



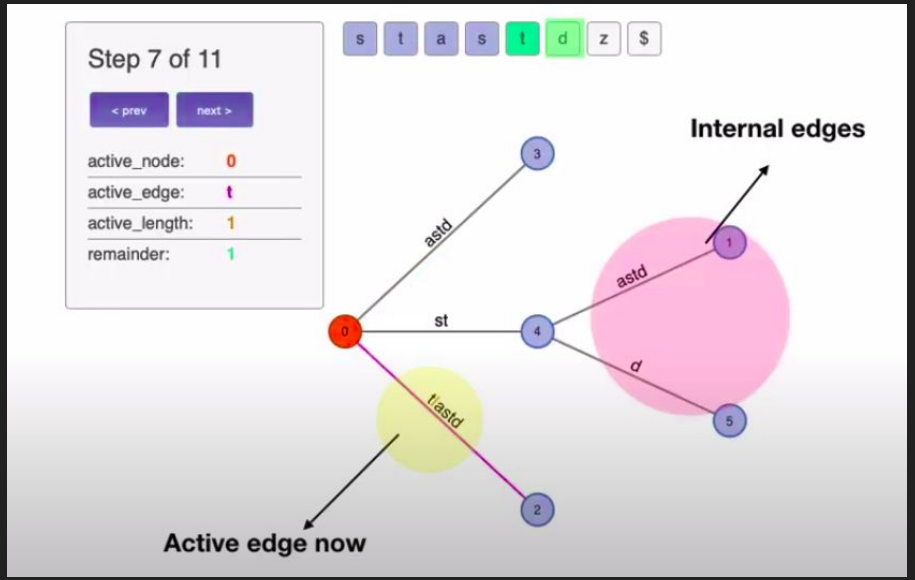


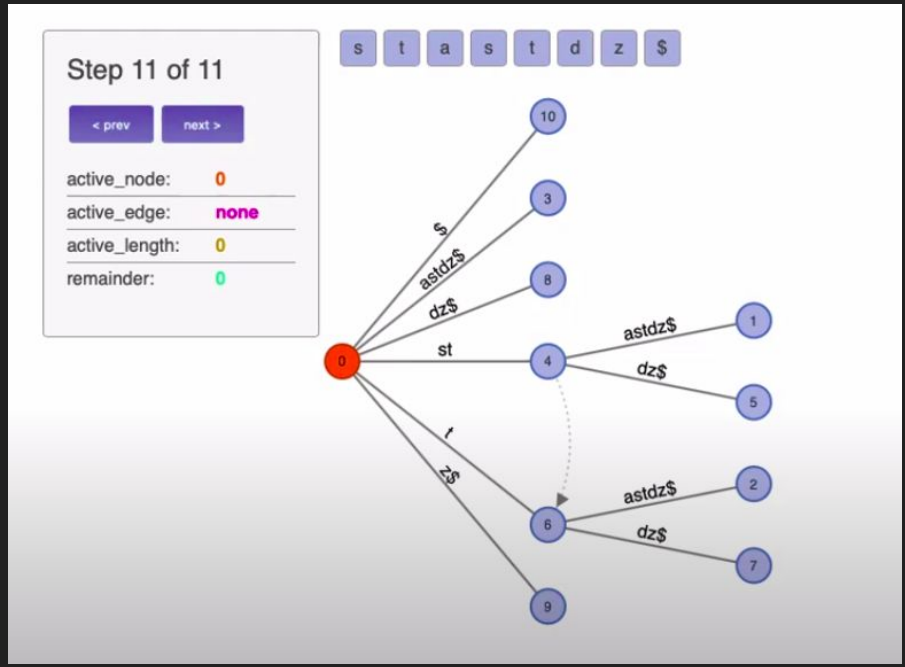
When we have common suffix





Splitting :


Step 7 of 11
< prev
next >
active_node: 0
active_edge: t
active_length: 1
remainder: 1
s t a s t d z $
Internal edges
astd
st
astd
d
t.astd
Active edge now


Step 11 of 11
< prev
next >
active_node: 0
active_edge: none
active_length: 0
remainder: 0
s t a s t d z $
$
astdz$
dz$
st
astdz$
dz$
t
z$
astdz$
dz$

Thank You