Fast Algorithm for Partial Covers in Words

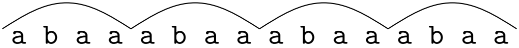
Tomasz Kociumaka[1], Solon P. Pissis[2,3],
Jakub Radoszewski[1], Wojciech Rytter[1],
Tomasz Waleń[4,1]

[1]University of Warsaw
[2]Heidelberg Institute for Theoretical Studies
[3]University of Florida
[4]International Institute of Molecular and Cell Biology in Warsaw

CPM 2013 Bad Herrenalb, June 17, 2013

Periodicity and Quasiperiodicity

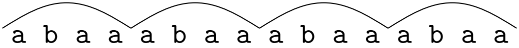
Periodicity: occurrences are aligned.



a b a a a b a a a b a a a b a a

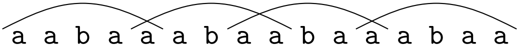
Periodicity and Quasiperiodicity

Periodicity: occurrences are aligned.



a b a a a b a a a b a a a b a a

Quasiperiodicity: occurrences may overlap.

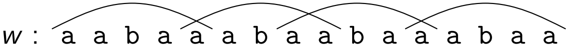


a a b a a a b a a b a a a b a a

Covers

Definition (Apostolico, Farach, Iliopoulos, 1991)

Let u be a factor of w. We say that u is a *cover* of w, if each position (letter) in w lies within some occurrence of u in w.

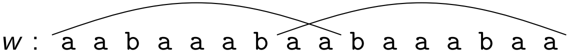

w : a a b a a a b a a b a a a b a a

The covers of w are aabaa

Covers

Definition (Apostolico, Farach, Iliopoulos, 1991)

Let u be a factor of w. We say that u is a *cover* of w, if each position (letter) in w lies within some occurrence of u in w.

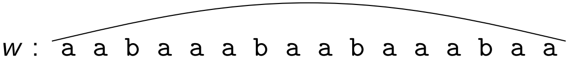


The covers of w are `aabaa`, `aabaaabaa`

Covers

Definition (Apostolico, Farach, Iliopoulos, 1991)

Let u be a factor of w. We say that u is a *cover* of w, if each position (letter) in w lies within some occurrence of u in w.



w : a a b a a a b a a b a a a b a a

The covers of w are aabaa, aabaaabaa and aabaaabaabaaabaa.

Covers

Definition (Apostolico, Farach, Iliopoulos, 1991)

Let u be a factor of w. We say that u is a *cover* of w, if each position (letter) in w lies within some occurrence of u in w.

w : `a a b a a a b a a b a a a b a a`

The covers of w are `aabaa`, `aabaaabaa`
and `aabaaabaabaaabaa`.
The whole word is always a cover of itself,
most words do not have any other cover.

Cover Index, Partial Covers

Definition

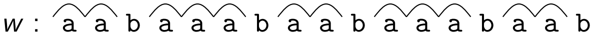
The *cover index* of u in w is the number of positions in w which lie within some occurrence of u in w.

w : a a b a a a b a a b a a a b a a b

Cover Index, Partial Covers

Definition

The *cover index* of u in w is the number of positions in w which lie within some occurrence of u in w.



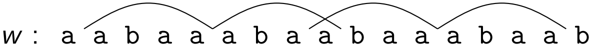
w : a a b a a a b a a b a a a b a a b

The cover index of a is 12

Cover Index, Partial Covers

Definition

The *cover index* of u in w is the number of positions in w which lie within some occurrence of u in w.



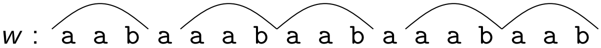
w : a a b a a a b a a b a a a b a a b

The cover index of a is 12, of abaa is 15

Cover Index, Partial Covers

Definition

The *cover index* of u in w is the number of positions in w which lie within some occurrence of u in w.



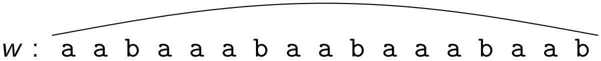
w : a a b a a a b a a b a a a b a a b

The cover index of a is 12, of abaa is 15,
of aab is 15

Cover Index, Partial Covers

Definition

The *cover index* of u in w is the number of positions in w which lie within some occurrence of u in w.



w : `a a b a a a b a a b a a a b a a b`

The cover index of `a` is 12, of `abaa` is 15,
of `aab` is 15, only of `aabaaabaabaaabaab` is 17.

Cover Index, Partial Covers

Definition

The *cover index* of u in w is the number of positions in w which lie within some occurrence of u in w.

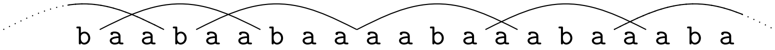
w : a a b a a a b a a b a a a b a a b

The cover index of a is 12, of abaa is 15,
of aab is 15, only of aabaaabaabaaabaab is 17.

Definition

For a positive integer α an α-partial cover of w is a factor of w with cover index at least α.

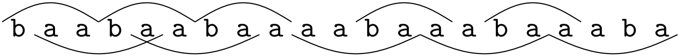
Other variants of covers



b a a b a a b a a a a b a a a b a a a b a

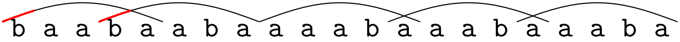
- *seeds* (Iliopoulos, Moore, Park; 1996) – covers of a superstring

Other variants of covers



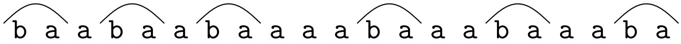
- *seeds* (Iliopoulos, Moore, Park; 1996) – covers of a superstring
- *k-covers* (Iliopoulos, Smyth; 1998) – each position lies within an occurrence of at least one of k factors, together being a k-cover

Other variants of covers



- *seeds* (Iliopoulos, Moore, Park; 1996) – covers of a superstring
- *k-covers* (Iliopoulos, Smyth; 1998) – each position lies within an occurrence of at least one of *k* factors, together being a *k*-cover
- *approximate covers* (Sim, Park, Kim, Lee; 2002) – each position is lies within an occurrence of a factor *similar* to the approximate cover

Other variants of covers



- *seeds* (Iliopoulos, Moore, Park; 1996) – covers of a superstring
- *k-covers* (Iliopoulos, Smyth; 1998) – each position lies within an occurrence of at least one of *k* factors, together being a *k*-cover
- *approximate covers* (Sim, Park, Kim, Lee; 2002) – each position is lies within an occurrence of a factor *similar* to the approximate cover
- *enhanced covers* (Flouri, Iliopoulos, K., Pissis, Puglisi, Smyth, Tyczyński; 2012) – as partial covers with an additional requirement of being simultaneously a border

Our Results

Problem (PartialCovers)

Given a word w and a positive integer α, identify all shortest α-partial covers of w.

Theorem

The PartialCovers *problem can be solved in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, where $n = |w|$.*

Our Results

Problem (PartialCovers)

Given a word w and a positive integer α, identify all shortest α-partial covers of w.

Theorem

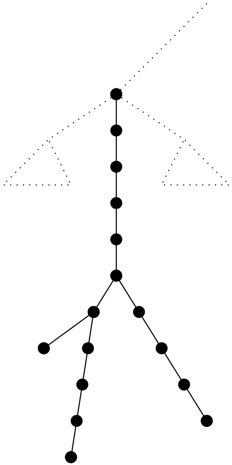
The PartialCovers *problem can be solved in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, where $n = |w|$.*

Theorem

For any word w of length n exists a data structure of size $\mathcal{O}(n)$, which given u can find the cover index of u in $\mathcal{O}(|u|)$ time. It can be built in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. If $u = w[i..j]$ is given as a pair of integers i, j, then $\mathcal{O}(\log \log |u|)$ query time can be achieved.

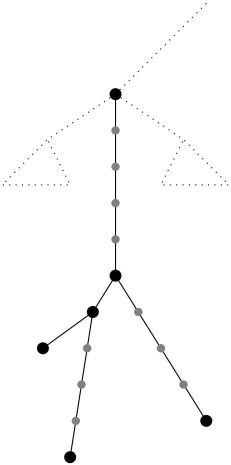
Suffix Trees: Notation

- The *suffix trie* of w for each factor u of w has a node corresponding to u, called the *locus* of u.



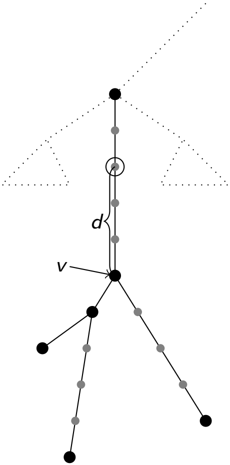
Suffix Trees: Notation

- The *suffix trie* of w for each factor u of w has a node corresponding to u, called the *locus* of u.
- In the *suffix tree* only $\mathcal{O}(|w|)$ nodes are stored *explicitely* (*explicit nodes*).



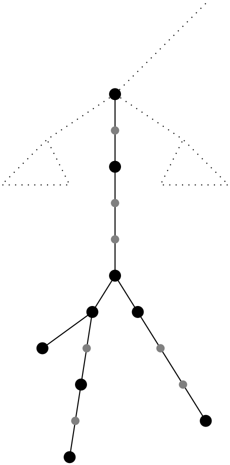
Suffix Trees: Notation

- The *suffix trie* of w for each factor u of w has a node corresponding to u, called the *locus* of u.
- In the *suffix tree* only $\mathcal{O}(|w|)$ nodes are stored *explicitely* (*explicit nodes*).
- The remaining nodes (*implicit nodes*) are represented by the highest explicit descendant and the distance to it.



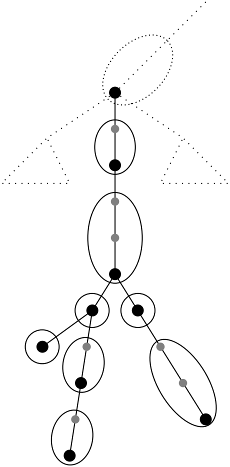
Suffix Trees: Notation

- The *suffix trie* of w for each factor u of w has a node corresponding to u, called the *locus* of u.
- In the *suffix tree* only $\mathcal{O}(|w|)$ nodes are stored *explicitely* (*explicit nodes*).
- The remaining nodes (*implicit nodes*) are represented by the highest explicit descendant and the distance to it.
- We *augment* the suffix tree: some implicit nodes are back explicit (called *extra nodes*).



Suffix Trees: Notation

- The *suffix trie* of w for each factor u of w has a node corresponding to u, called the *locus* of u.
- In the *suffix tree* only $\mathcal{O}(|w|)$ nodes are stored *explicitely* (*explicit nodes*).
- The remaining nodes (*implicit nodes*) are represented by the highest explicit descendant and the distance to it.
- We *augment* the suffix tree: some implicit nodes are back explicit (called *extra nodes*).
- An *edge* of a tree contains all implicit nodes and the lower explicit end.



Auxiliary definitions

Definition

A factor u is a *primitive square* if $u = v^2$ for some v, but $u \neq v^{2k}$ for any v and $k \geq 2$.

Examples: `aa`, `abaaba`. Non-examples: `ababa`, `abababab`.

Auxiliary definitions

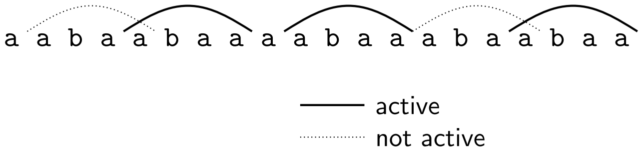
Definition

A factor u is a *primitive square* if $u = v^2$ for some v, but $u \neq v^{2k}$ for any v and $k \geq 2$.

Examples: `aa`, `abaaba`. Non-examples: `ababa`, `abababab`.

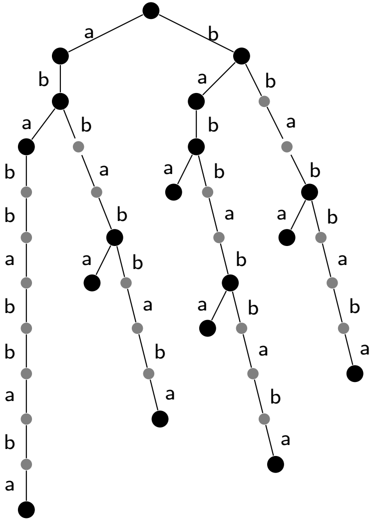
Definition

An occurrence of u in w is *active* if no other occurrence of u in w starts within it.



Cover Suffix Tree

The cover suffix tree of w $CST(w)$ is the suffix tree of w:

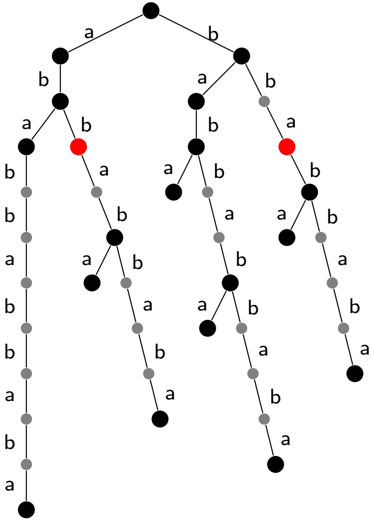


$CST($ababbabbaba$)$

Cover Suffix Tree

The cover suffix tree of w $CST(w)$ is the suffix tree of w:

- *augmented* with nodes corresponding to halves of primitive squares,

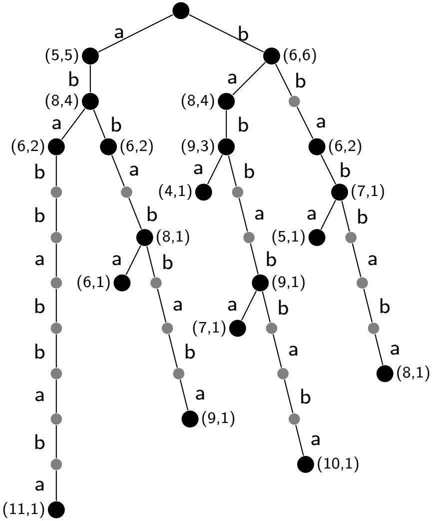


$CST(\mathtt{ababbabbaba})$

Cover Suffix Tree

The cover suffix tree of w $CST(w)$ is the suffix tree of w:

- *augmented* with nodes corresponding to halves of primitive squares,
- with each explicit node *annotated* with a pair $(cv(v), \Delta(v))$, where $cv(v)$ is the cover index of v and $\Delta(v)$ is the number of active occurrences of v.



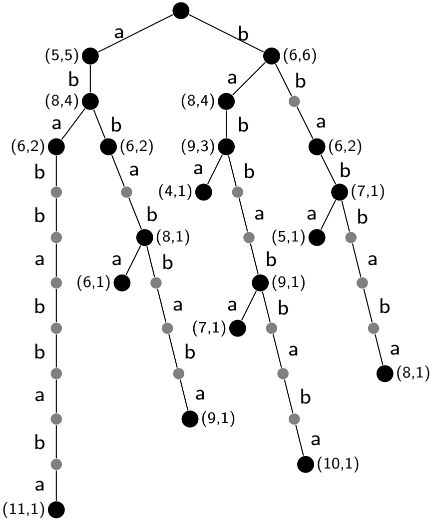
$CST(\texttt{ababbabbaba})$

Cover Suffix Tree

The cover suffix tree of w $CST(w)$ is the suffix tree of w:

- *augmented* with nodes corresponding to halves of primitive squares,
- with each explicit node *annotated* with a pair $(cv(v), \Delta(v))$, where $cv(v)$ is the cover index of v and $\Delta(v)$ is the number of active occurrences of v.

The number of square factors is linear (Fraenkel, Simpson, 1998), so the size of $CST(w)$ is $O(|w|)$.

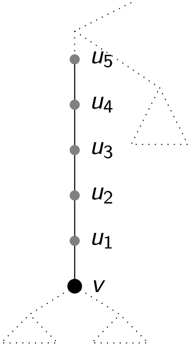


$CST(\mathtt{ababbabbaba})$

Crucial Lemma

Lemma

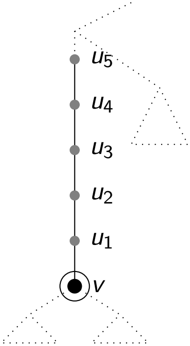
Let $v = u_0, u_1, \ldots, u_k$ *be the nodes of an edge of* $CST(w)$ *with* v *being the lowest node. Then* $cv(u_i) = cv(v) - i\Delta(v)$*.*



Crucial Lemma

Lemma

Let $v = u_0, u_1, \ldots, u_k$ be the nodes of an edge of $CST(w)$ with v being the lowest node. Then $cv(u_i) = cv(v) - i\Delta(v)$.



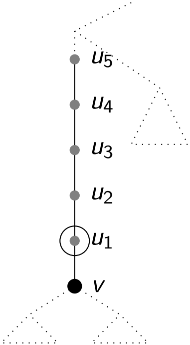
Proof.

Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐

Crucial Lemma

Lemma

Let $v = u_0, u_1, \ldots, u_k$ be the nodes of an edge of $CST(w)$ with v being the lowest node. Then $cv(u_i) = cv(v) - i\Delta(v)$.



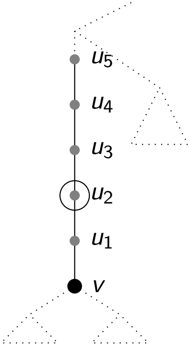
Proof.

Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐

Crucial Lemma

Lemma

Let $v = u_0, u_1, \ldots, u_k$ *be the nodes of an edge of* $CST(w)$ *with* v *being the lowest node. Then* $cv(u_i) = cv(v) - i\Delta(v)$*.*



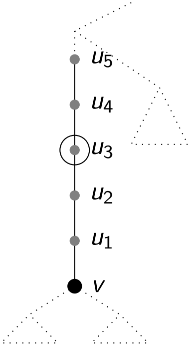
Proof.

Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐

Crucial Lemma

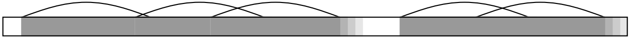
Lemma

Let $v = u_0, u_1, \ldots, u_k$ be the nodes of an edge of $CST(w)$ with v being the lowest node. Then $cv(u_i) = cv(v) - i\Delta(v)$.



Proof.

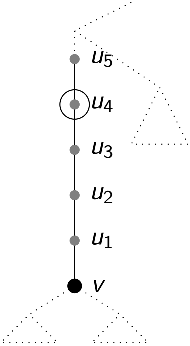
Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐



Crucial Lemma

Lemma

Let $v = u_0, u_1, \ldots, u_k$ be the nodes of an edge of $CST(w)$ with v being the lowest node. Then $cv(u_i) = cv(v) - i\Delta(v)$.



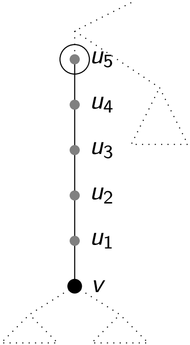
Proof.

Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐

Crucial Lemma

Lemma

Let $v = u_0, u_1, \ldots, u_k$ be the nodes of an edge of $CST(w)$ with v being the lowest node. Then $cv(u_i) = cv(v) - i\Delta(v)$.



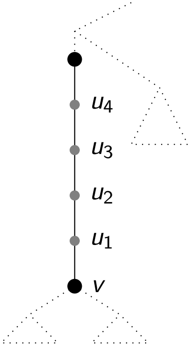
Proof.

Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐

Crucial Lemma

Lemma

Let $v = u_0, u_1, \ldots, u_k$ be the nodes of an edge of $CST(w)$ with v being the lowest node. Then $cv(u_i) = cv(v) - i\Delta(v)$.



Proof.

Recall that if u, u' are on the same edge of the suffix tree, then occurrences of u and u' start at the same positions. In $CST(w)$ also the active occurrences agree. Thus, u_{i+1} covers $\Delta(u_i) = \Delta(v)$ positions less than u_i. ☐

Answering Queries

- Recall that we have defined the locus of u as a pair (v, d), where v is the highest explicit descendant of u.
- The Lemma proves that $cv(u) = cv(v) - d\Delta(v)$, so computing the cover index of u given its locus in $CST(w)$ is trivial.

Answering Queries

- Recall that we have defined the locus of u as a pair (v, d), where v is the highest explicit descendant of u.
- The Lemma proves that $cv(u) = cv(v) - d\Delta(v)$, so computing the cover index of u given its locus in $CST(w)$ is trivial.
- If u is given explicitly, simply traverse $CST(w)$ to find the locus ($\mathcal{O}(|u|)$ time)
- If $u = w[i..j]$ is given as a pair of indices, use the weighted ancestors data structure ($\mathcal{O}(\log\log|u|)$ time).

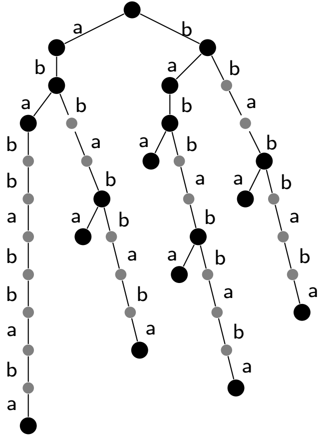
Answering Queries

- Recall that we have defined the locus of u as a pair (v, d), where v is the highest explicit descendant of u.
- The Lemma proves that $cv(u) = cv(v) - d\Delta(v)$, so computing the cover index of u given its locus in $CST(w)$ is trivial.
- If u is given explicitly, simply traverse $CST(w)$ to find the locus ($\mathcal{O}(|u|)$ time)
- If $u = w[i..j]$ is given as a pair of indices, use the weighted ancestors data structure ($\mathcal{O}(\log\log|u|)$ time).
- Finding the shortest α-partial covers reduces to solving one linear inequality per edge ($cv(v) - d\Delta(v) \geq \alpha$), this takes linear time once $CST(w)$ is given.

Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

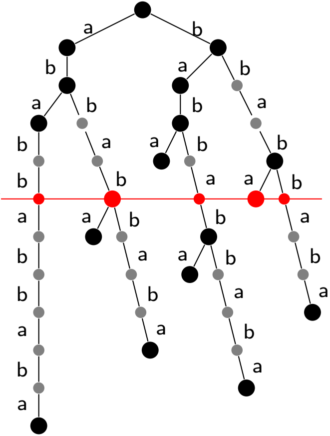
- We start with the suffix tree.



Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

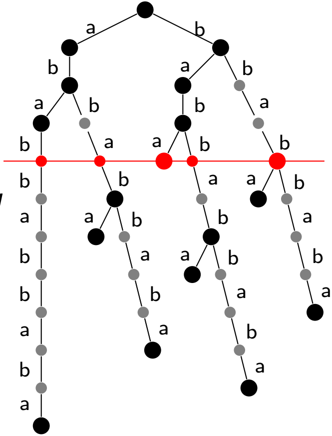
- We start with the suffix tree.
- We process the nodes in the decreasing order of the corresponding factors' lengths.
- While at level d, for each factor of length d we implicitly keep a *sorted* linked list of its occurrences.



Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

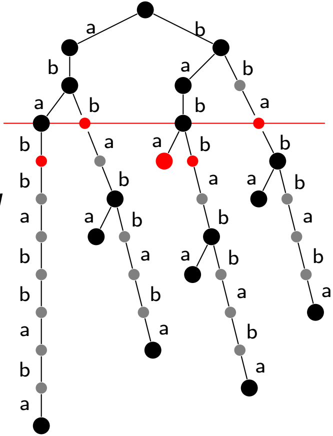
- We start with the suffix tree.
- We process the nodes in the decreasing order of the corresponding factors' lengths.
- While at level d, for each factor of length d we implicitly keep a *sorted* linked list of its occurrences.



Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

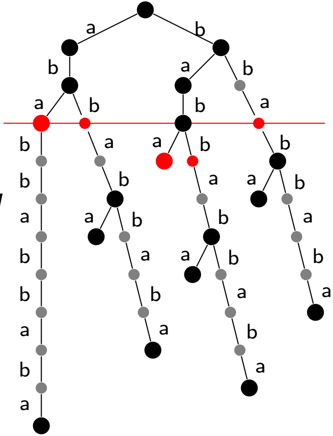
- We start with the suffix tree.
- We process the nodes in the decreasing order of the corresponding factors' lengths.
- While at level d, for each factor of length d we implicitly keep a *sorted* linked list of its occurrences.
- At implicit nodes, these lists do not need to be update.



Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

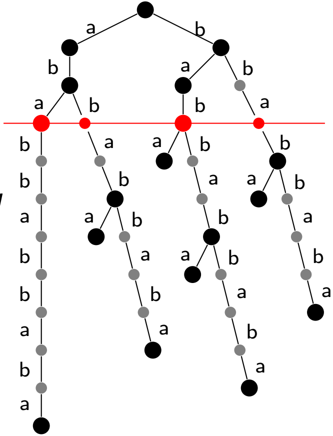
- We start with the suffix tree.
- We process the nodes in the decreasing order of the corresponding factors' lengths.
- While at level d, for each factor of length d we implicitly keep a *sorted* linked list of its occurrences.
- At implicit nodes, these lists do not need to be update.
- We need manually to take care of explicit nodes.



Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

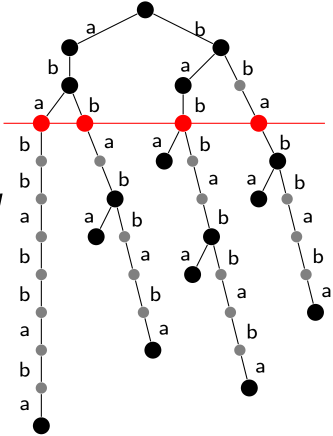
- We start with the suffix tree.
- We process the nodes in the decreasing order of the corresponding factors' lengths.
- While at level d, for each factor of length d we implicitly keep a *sorted* linked list of its occurrences.
- At implicit nodes, these lists do not need to be update.
- We need manually to take care of explicit nodes.



Construction Algorithm

The structure resembles an $\mathcal{O}(n \log n)$-time construction of a similar data structure, MAST (Brodal et al.; 2002).

- We start with the suffix tree.
- We process the nodes in the decreasing order of the corresponding factors' lengths.
- While at level d, for each factor of length d we implicitly keep a *sorted* linked list of its occurrences.
- At implicit nodes, these lists do not need to be update.
- We need manually to take care of explicit nodes.
- We also need to add extra nodes.



Change Sets

Definition

Let $\mathcal{P}$ be a partition of $[n] = \{1, \ldots, n\}$. For any $a \in [n]$ we define the *successor* of a in $\mathcal{P}$ as $\min\{b \in P : b > a\}$ where $P \in \mathcal{P}$ is the partition class containing a.
We assume $\min \emptyset = \infty$.

Change Sets

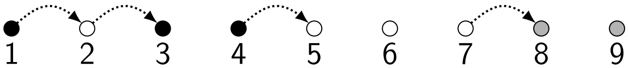
Definition

Let $\mathcal{P}$ be a partition of $[n] = \{1, \ldots, n\}$. For any $a \in [n]$ we define the *successor* of a in $\mathcal{P}$ as $\min\{b \in P : b > a\}$ where $P \in \mathcal{P}$ is the partition class containing a.
We assume $\min \emptyset = \infty$.

Definition

Consider two partitions $\mathcal{P}$, $\mathcal{P}'$ of $[n]$. The *change set* of $\mathcal{P}$ and $\mathcal{P}'$ is the family of pairs (i, j) such that j is the successor of i in $\mathcal{P}'$, but not in $\mathcal{P}$.

Let $\mathcal{P} = \{\{1, 3, 4\}, \{2, 5, 6, 7\}, \{8, 9\}\}$ and $\mathcal{P}' = \{\{1, \ldots, 9\}\}$.
The change set is $\{(1, 2), (2, 3), (4, 5), (7, 8)\}$.



Ordered Disjoint Sets

Problem (Ordered Disjoint Sets)

Maintain a partition of $[n]$*, support the following operations:*

- *given* i ***find*** *the partition class of* i*,*
- *given* $I \subseteq [n]$ ***merge*** *all the partition classes of elements contained in* I*, return the change set of the underlying modification of the partition.*

Ordered Disjoint Sets

Problem (Ordered Disjoint Sets)

Maintain a partition of $[n]$*, support the following operations:*

- *given* i ***find*** *the partition class of* i*,*
- *given* $I \subseteq [n]$ ***merge*** *all the partition classes of elements contained in* I*, return the change set of the underlying modification of the partition.*

Lemma

There is a data structure of size $\mathcal{O}(n)$*, which handles a sequence of* q *operations in* $\mathcal{O}(q + n \log n)$ *total time. Moreover, the total size of change sets is* $\mathcal{O}(n \log n)$*.*

Ordered Disjoint Sets

Problem (Ordered Disjoint Sets)

Maintain a partition of $[n]$, support the following operations:

- *given i **find** the partition class of i,*
- *given $I \subseteq [n]$ **merge** all the partition classes of elements contained in I, return the change set of the underlying modification of the partition.*

Lemma

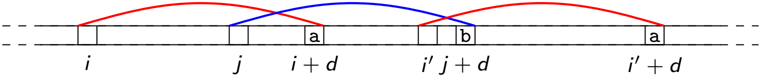
There is a data structure of size $\mathcal{O}(n)$, which handles a sequence of q operations in $\mathcal{O}(q + n \log n)$ total time. Moreover, the total size of change sets is $\mathcal{O}(n \log n)$.

We use this data structure to store a partition into equivalence classes of the $w[i..i+d-1] = w[j..j+d-1]$ relation.

Construction Algorithm

We handle two types of events:

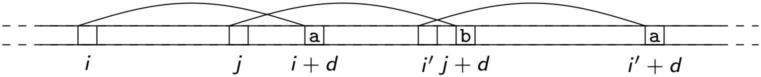
- A branching node, the lists need to be merged.



Construction Algorithm

We handle two types of events:

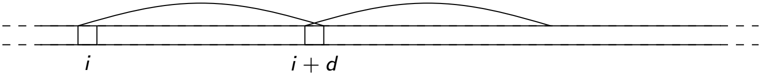
- A branching node, the lists need to be merged.



Construction Algorithm

We handle two types of events:

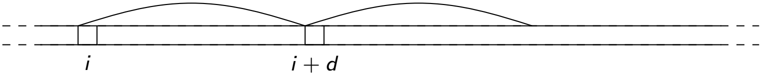
- A branching node, the lists need to be merged.
- An occurrence becomes active. The corresponding node must be made explicit.



Construction Algorithm

We handle two types of events:

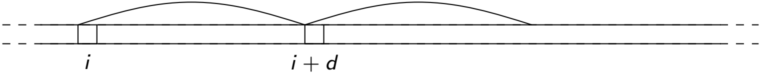
- A branching node, the lists need to be merged.
- An occurrence becomes active. The corresponding node must be made explicit.



Construction Algorithm

We handle two types of events:

- A branching node, the lists need to be merged.
- An occurrence becomes active. The corresponding node must be made explicit.

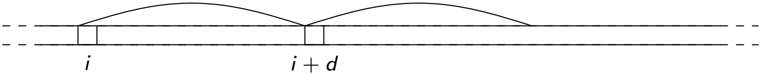


- With each list we keep a few aggregation values, which let us determine $cv(v)$ and $\Delta(v)$.

Construction Algorithm

We handle two types of events:

- A branching node, the lists need to be merged.
- An occurrence becomes active. The corresponding node must be made explicit.



- With each list we keep a few aggregation values, which let us determine $cv(v)$ and $\Delta(v)$.

The complexity of the construction algorithm is amortized by the total size of change sets, which gives

Theorem

Given a word w of length n, its Cover Suffix Tree $CST(w)$ can be constructed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

Summary

- We have shown that with CST it is very easy compute the cover index of any factor u.

Summary

- We have shown that with CST it is very easy compute the cover index of any factor u.
- We can also find the shortest factor covering at least α positions of w.

Summary

- We have shown that with CST it is very easy compute the cover index of any factor u.
- We can also find the shortest factor covering at least α positions of w.
- These are just sample queries on partial covers that CST can handle, e.g. a different criterion instead of length.

Summary

- We have shown that with *CST* it is very easy compute the cover index of any factor u.
- We can also find the shortest factor covering at least α positions of w.
- These are just sample queries on partial covers that *CST* can handle, e.g. a different criterion instead of length.

Can *CST* be constructed faster?
A linear time algorithm might be difficult, all seeds can be easily found in linear time using *CST*.

Thank you for your attention!