Factorizing a string into squares in linear time

Yoshiaki Matsuoka, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda (Kyushu U.)
Florin Manea (Kiel U.)
From string to squares?

- In this presentation, I talk about decomposition of a string into squares.
Squares (as strings!)

“Our square” is a string of form $xx$.

- aabaab
- abababab
- ababaababa
- ababaababa
Primitively rooted squares

A square $xx$ is called a *primitively rooted square* if its root $x$ is primitive (i.e., $x \neq y^k$ for any string $y$ and integer $k$).

- *aabaab* : primitively rooted square
- *ababababab* : not primitively rooted square
- *ababaababa* : primitively rooted square
Our problem

- Determine whether a given string can be factorized into a sequence of squares. If the answer is yes, then compute one of such factorizations.

E.g.)
- aabaabaaaaaa → Yes
  ◦ (aabaab, aaaaaa),
  ◦ (aabaab, aaaa, aa),
  ◦ (aa, baabaa, aa, aa), and so on.
- aabaabbbbab → No
Previous work

Times for computing square factorization

<table>
<thead>
<tr>
<th></th>
<th>[Dumitran et al., 2015]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sq. factor.</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

- $n$ is the length of the input string.
## Previous work

### Times for computing square factorization

<table>
<thead>
<tr>
<th></th>
<th>[Dumitran et al., 2015]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sq. factor.</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Largest sq. factor.</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

$n$ is the length of the input string.
Our contribution

### Times for computing square factorization

<table>
<thead>
<tr>
<th></th>
<th>[Dumitran et al., 2015]</th>
<th>Our solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sq. factor.</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Largest sq. factor.</td>
<td>$O(n \log n)$</td>
<td>$O(n + (n \log^2 n) / \omega)$</td>
</tr>
<tr>
<td>Smallest sq. factor.</td>
<td>$-$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

- $n$ is the length of the input string.
- Our results for arbitrary/largest square factorizations are valid on word RAM with word size $\omega = \Omega(\log n)$. 


Our contribution

Times for computing square factorization

<table>
<thead>
<tr>
<th></th>
<th>[Dumitran et al., 2015]</th>
<th>Our solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sq. factor.</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Largest sq. factor.</td>
<td>$O(n \log n)$</td>
<td>$O(n + (n \log^2 n) / \omega)$</td>
</tr>
<tr>
<td>Smallest sq. factor.</td>
<td>–</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

- $n$ is the length of the input string.
- Our results for arbitrary/largest square factorizations are valid on word RAM with word size $\omega = \Omega(\log n)$. 
Simple observation

- Every square is of even length.
- Thus, if string $w$ has a square factorization, then $w$ also has a square factorization which consists only of primitively rooted squares.

E.g.)
- $aaaaaa|abababab$
- $aa|aa|aa|abab|abab$
Any string of length $n$ contains $O(n \log n)$ primitively rooted squares [Crochemore & Rytter, 1995].

The simple observation + the above lemma lead to a natural DP approach which computes a square factorization in $O(n \log n)$ time.
Dumitran et al.’s algorithm

Consider the following DAG $G$ for string $w$:
- There are $n+1$ nodes.
- There is a directed edge $(e+1, b)$ in $G$. ⇔ Substring $w[b..e]$ is a primitively rooted square.
Consider the following DAG $G$ for string $w$:
- There are $n+1$ nodes.
- There is a directed edge $(e+1, b)$ in $G$. $\iff$ Substring $w[\text{b..e}]$ is a primitively rooted square.
Dumitran et al.’s algorithm

- DAG $G$ has a path from the rightmost node to the leftmost node.
  $\iff$ There is a square factorization of $w$. 
Dumitran et al.'s algorithm

- The rightmost node is associated with a 1.
- Initially, all the other nodes are associated with 0’s.
We process each node from right to left.

Each node $v$ gets a 1 iff there is an incoming edge to $v$ from a node that is associated with a 1.
Dumitran et al.’s algorithm

- We process each node from right to left.
- Each node $v$ gets a 1 iff there is an incoming edge to $v$ from a node that is associated with a 1.
Dumitran et al.'s algorithm

- We process each node from right to left.
- Each node \( v \) gets a 1 iff there is an incoming edge to \( v \) from a node that is associated with a 1.
Dumitran et al.’s algorithm

- We process each node from right to left.
- Each node $v$ gets a 1 iff there is an incoming edge to $v$ from a node that is associated with a 1.
Dumitran et al.’s algorithm

- We process each node from right to left.
- Each node $v$ gets a 1 iff there is an incoming edge to $v$ from a node that is associated with a 1.
Finally, there is a square factorization of the string iff the leftmost node is associated with a 1.
Dumitran et al.’s algorithm

A path from the rightmost node to the leftmost node corresponds to a square factorization.
Dumitran et al.’s algorithm

- Another path from the rightmost node to the leftmost node corresponds to another square factorization.
Dumitran et al.’s algorithm

- Clearly, the number of edges in this DAG is equal to the number of primitively rooted squares in the string, which is $O(n \log n)$.
- Hence, their algorithm takes $O(n \log n)$ time.
Ideas of our $O(n)$-time algorithm

- We accelerate Dumitran et al.’s algorithm by a mixed use of
  - *runs* (maximal repetitions in the string);
  - *bit parallelism* (performing some DP computation in a batch).
Runs

- A triple $(p, b, e)$ of integers is said to be a *run* of a string $w$ if
  - The substring $w[b..e]$ is a repetition with the smallest period $p$ (i.e., $2p \leq e-s+1$), and
  - The repetition is non-extensible to left nor right with the same period $p$.

![Diagram of runs](image)
Long and short period runs

- Let $\omega$ be the machine word size.
- A run $(p, b, e)$ in a string is called
  - a *long period run* (LPR) if $2p \geq \omega$;
  - a *short period run* (SPR) if $2p < \omega$.

E.g.) For $\omega = 4$

- $a\ a\ a\ b$ is an LPR $(3, 1, 8)$
- $a\ a\ a\ b$ is a SPR $(1, 4, 5)$
- $a\ a\ a\ a\ a$ is a SPR $(1, 7, 10)$
Edges that correspond to long period runs are called *long edges*.
Short edges

- Edges that correspond to short period runs are called *short edges*.

SPR (1, 1, 2)  SPR (1, 4, 5)  SPR (1, 7, 10)
How to process long edges

- We partition the nodes into blocks of length $\omega$ each.
How to process long edges

- Since the long edges that correspond to the same LPR have the same length and are consecutive, we can process $\omega$ of them in a batch, by performing a bit-wise OR.

※ Our algorithm does NOT create edges explicitly.
How to process long edges

- Since the long edges that correspond to the same LPR have the same length and are consecutive, we can process $\omega$ of them in a batch, by performing a bit-wise OR.

※ Our algorithm does NOT create edges explicitly.
Time cost for long edges

- We can process at most $\omega$ long edges in a batch in $O(1)$ time, hence we can process all long edges in $O((n \log n)/\omega)$ time.

- An $O(n + \#\text{LPR})$-time preprocessing allows us to perform these operations without constructing long edges explicitly.

- Thus we need $O(n + \#\text{LPR} + (n \log n)/\omega)$ total time for long edges.
How to process short edges

- Every short edge is shorter than $\omega$.
- Hence, for each node $i$, it is enough to consider at most $\omega$ in-coming short edges.

※ Our algorithm does NOT create edges explicitly.
How to process short edges

- To process these short edges in a batch, we use a bit mask $B_i$, indicating if each node has a short edge to node $i$.

\[
\begin{array}{cccccccc}
\cdots & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \cdots \\
\end{array}
\]

\[
B_i = \begin{array}{ccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1
\end{array}
\]

※ Our algorithm does NOT create edges explicitly.
How to process short edges

- To process these short edges in a batch, we use a bit mask $B_i$ indicating if each node has a short edge to node $i$.

```
\[ B_i = \begin{bmatrix}
    0 & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}
\]
```

$\text{bitwise AND } B_i = \begin{bmatrix}
    0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}$

※ Our algorithm does NOT create edges explicitly.
How to process short edges

- If there is a 1 in the resulting bit string, then node $i$ gets a 1.

\[ i\]

\[ B_i = \begin{array}{cccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1
\end{array} \]

bitwise AND

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array} \]

※ Our algorithm does NOT create edges explicitly.
How to process short edges

- If there is a 1 in the resulting bit string, then node $i$ gets a 1.

$B_i = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$

bitwise AND

$\Box$ Our algorithm does NOT create edges explicitly.
**Time cost for short edges**

- Given bit mask $B_i$, we can process all incoming short edges of node $i$ in $O(1)$ time.

- An $O(n + \#SPR)$-time preprocessing allows us to compute the bit mask $B_i$ for all nodes $i$.

- Overall, we need $O(n + \#SPR)$ total time for short edges.
Main result

**Theorem**

Given a string of length $n$, we can compute a square factorization of the string in $O(n)$ time.

- $O(n + \#\text{LPR} + \#\text{SPR} + (n \log n)/\omega)$ time.
  - $\#\text{LPR} + \#\text{SPR} < n$ [Bannai et al., 2015]
  - $(n \log n)/\omega = O(n)$ because $\omega = \Omega(\log n)$.

- Hence, it takes $O(n)$ total time.
Open questions

- Is it possible to compute a square factorization in $O(n)$ time without bit parallelism?
- Is it possible to compute largest/smallest square factorizations in $O(n)$ time?

It is possible to compute largest/smallest repetition factorization in $O(n \log n)$ time [PSC 2016, accepted].

- Here each factor is a repetition of form $x^k x'$ with $k \geq 2$ and $x'$ being a prefix of $x$.
- $O(n)$-time algorithm exists for this?