On Some Algorithmic Problems in the Area of Combinatorics on Partial Words

Florin Manea

Faculty of Mathematics and Computer Science,
University of Bucharest,
Faculty of Computer Science,
Otto-von-Guericke-University Magdeburg

joint work with: Adrian Diaconu, Robert Mercaș, Cătălin Tiseanu,

Theoretical Computer Science - Foundations and Applications
Nis 2009

Outline

1. Preliminaries
2. Definitions
3. Data Structures
4. Queries and Updates on Partial Words
5. Counting Problems

1 Preliminaries

2 Definitions

3 Data Structures

4 Queries and Updates on Partial Words

5 Counting Problems

Combinatorics on Words

- The area of Combinatorics on Words took birth at the beginning of the last century: the works of Axel Thue.

Combinatorics on Words

- The area of Combinatorics on Words took birth at the beginning of the last century: the works of Axel Thue.
- Focus on: several combinatorial problems that arose in the study of the sequences of symbols, which were solved with the usual tools of discrete mathematics.
- Some of the most important results obtained by Thue regarded the repetitions (consecutive occurrences of a factor) inside a word. For example: the infinite Thue word.

Combinatorics on Words

- The area of Combinatorics on Words took birth at the beginning of the last century: the works of Axel Thue.
- Focus on: several combinatorial problems that arose in the study of the sequences of symbols, which were solved with the usual tools of discrete mathematics.
- Some of the most important results obtained by Thue regarded the repetitions (consecutive occurrences of a factor) inside a word. For example: the infinite Thue word.
- The interest in the study of Combinatorics on Words is increasing: computer science (language theoretic properties, algorithms on strings, data compression, data communication, model checking), biology and bio-inspired computing (DNA analysis, bio-inspired computing models, molecular biology), etc., starting from the premise that the data used in these areas can be easily represented as words over some particular alphabet.

Partial and Infinite Words

- Partial words (a canonical extension of the classical words) are sequences that, besides regular symbols, may have a number of unknown symbols, called "holes" or "wild cards".

$$a\diamond aba\diamond babaa\diamond ba$$

Partial and Infinite Words

- Partial words (a canonical extension of the classical words) are sequences that, besides regular symbols, may have a number of unknown symbols, called "holes" or "wild cards".

$$a\diamond aba\diamond babaa\diamond ba$$

- Motivated by molecular biology: the alignment of the DNA sequences is conceived as a construction of two compatible partial words.
- Combinatorial properties studied: periodicity, freeness, primitivity, conjugacy etc.. Part of the studies were focused on finding efficient algorithms for testing such properties.

Partial and Infinite Words

- Partial words (a canonical extension of the classical words) are sequences that, besides regular symbols, may have a number of unknown symbols, called "holes" or "wild cards".

$$a\diamond aba\diamond babaa\diamond ba$$

- Motivated by molecular biology: the alignment of the DNA sequences is conceived as a construction of two compatible partial words.
- Combinatorial properties studied: periodicity, freeness, primitivity, conjugacy etc.. Part of the studies were focused on finding efficient algorithms for testing such properties.
- Typically, the words have finite length (i.e, they have a finite number of symbols).

Partial and Infinite Words

- Partial words (a canonical extension of the classical words) are sequences that, besides regular symbols, may have a number of unknown symbols, called "holes" or "wild cards".

$$a\diamond aba\diamond babaa\diamond ba$$

- Motivated by molecular biology: the alignment of the DNA sequences is conceived as a construction of two compatible partial words.
- Combinatorial properties studied: periodicity, freeness, primitivity, conjugacy etc.. Part of the studies were focused on finding efficient algorithms for testing such properties.
- Typically, the words have finite length (i.e, they have a finite number of symbols).However, in many applications, the length of the words investigated can be arbitrarily large (i.e, they are defined by some iterated algorithmic process, and, at each moment we know only a finite part of the word, but also know how it can be extended).

Focus (first part): Queries and Updates

- We are not interested only in the properties of a given word, but also in the properties of its factors. Thus: given a partial word as input, construct data structures (using efficient algorithms) that enable us to answer in constant time combinatorial queries regarding the factors of that word (is a given factor a repetition?, is a given factor primitive?, etc.).

Focus (first part): Queries and Updates

- We are not interested only in the properties of a given word, but also in the properties of its factors. Thus: given a partial word as input, construct data structures (using efficient algorithms) that enable us to answer in constant time combinatorial queries regarding the factors of that word (is a given factor a repetition?, is a given factor primitive?, etc.).
- We define an update operation for partial words: add a new symbol at a word's rightmost end. After such an operation is performed, we investigate how the data structures constructed before can be brought up to date, in order to still be able to answer in constant time combinatorial queries.

Focus (first part): Queries and Updates

- We are not interested only in the properties of a given word, but also in the properties of its factors. Thus: given a partial word as input, construct data structures (using efficient algorithms) that enable us to answer in constant time combinatorial queries regarding the factors of that word (is a given factor a repetition?, is a given factor primitive?, etc.).
- We define an update operation for partial words: add a new symbol at a word's rightmost end. After such an operation is performed, we investigate how the data structures constructed before can be brought up to date, in order to still be able to answer in constant time combinatorial queries.
- Differently from previous approaches: we are interested in constructing data structures allowing us to answer, in constant time, various combinatorial queries regarding every factor of a partial word, and that can be easily updated when the word is updated; on the other hand, the problems approached so far in this line of research consisted in testing combinatorial properties of the entire input word.

Focus (second part): Counting Problems

- Once we are able to identify fast the factors verify a combinatorial property, we are interested in computing the number of full words, over a given alphabet, compatible with such factors. For example: the number of distinct squares (i.e., full words of the form xx compatible with a factor of the given partial word).

Focus (second part): Counting Problems

- Once we are able to identify fast the factors verify a combinatorial property, we are interested in computing the number of full words, over a given alphabet, compatible with such factors. For example: the number of distinct squares (i.e., full words of the form xx compatible with a factor of the given partial word).
- Moreover, we can focus on even more natural problems, related to subword complexity: what is the number of distinct full factors of the same length of a partial word (for a given alphabet), what is the number of distinct full factors of a partial word (for a given alphabet)? The distinct full factors of a partial word are the full words compatible with at least one factor of that partial word.

Focus (second part): Counting Problems

- Once we are able to identify fast the factors verify a combinatorial property, we are interested in computing the number of full words, over a given alphabet, compatible with such factors. For example: the number of distinct squares (i.e., full words of the form xx compatible with a factor of the given partial word).
- Moreover, we can focus on even more natural problems, related to subword complexity: what is the number of distinct full factors of the same length of a partial word (for a given alphabet), what is the number of distinct full factors of a partial word (for a given alphabet)? The distinct full factors of a partial word are the full words compatible with at least one factor of that partial word.
- We show that several such problems are complete for the class $\#P$ (the class of function problems of the form "compute $f(x)$" where f is the number of accepting paths of an **NP** Turing machine).

Focus (second part): Counting Problems

- Once we are able to identify fast the factors verify a combinatorial property, we are interested in computing the number of full words, over a given alphabet, compatible with such factors. For example: the number of distinct squares (i.e., full words of the form xx compatible with a factor of the given partial word).
- Moreover, we can focus on even more natural problems, related to subword complexity: what is the number of distinct full factors of the same length of a partial word (for a given alphabet), what is the number of distinct full factors of a partial word (for a given alphabet)? The distinct full factors of a partial word are the full words compatible with at least one factor of that partial word.
- We show that several such problems are complete for the class $\#P$ (the class of function problems of the form "compute $f(x)$" where f is the number of accepting paths of an **NP** Turing machine).
- We were not able to solve the main problem (what is the number of distinct full factors of a partial word): is it $\#P$-complete? can it be solved efficiently? However, we were able to show that the usual

Basic References:

- "Freeness of partial words", F. Manea, R. Mercas, Theoretical Computer Science 389(1-2), pg. 265-277, Elsevier, 2007.
- "Combinatorial Queries and Updates on Partial Words", A. Diaconu, F. Manea, C. Tiseanu, Proceedings of FCT 2009, Lecture Notes in Computer Science 5699, pg. 96 - 108, Springer, 2009.
- "A Series of Hardness Results on Counting Distinct Full Factors of Partial Words", F. Manea, C. Tiseanu, work in progress.

1 Preliminaries

2 Definitions

3 Data Structures

4 Queries and Updates on Partial Words

5 Counting Problems

Partial Words

- A *partial word* of length n over the alphabet A is a partial function $u\colon \{1,\ldots,n\} \xrightarrow{\circ} A$. For $i \in \{1,\ldots,n\}$, if $u(i)$ is defined we say that i *belongs to the domain of* u (denoted by $i \in D(u)$), otherwise we say that i *belongs to the set of holes of* u (denoted by $i \in H(u)$).

Partial Words

- A *partial word* of length n over the alphabet A is a partial function $u\colon \{1,\ldots,n\} \xrightarrow{\circ} A$. For $i \in \{1,\ldots,n\}$, if $u(i)$ is defined we say that i *belongs to the domain of* u (denoted by $i \in D(u)$), otherwise we say that i *belongs to the set of holes of* u (denoted by $i \in H(u)$).
- Finite partial words are seen as words over the extended alphabet $A \cup \{\diamond\}$: a partial word u of length n is depicted as $u = a_1 \ldots a_n$, where $a_i = u(i)$, for $i \in D(u)$, and $a_i = \diamond$, otherwise. In this way, we define the catenation, respectively the equality, of partial words, as the catenation, respectively the equality, of the corresponding words over $A \cup \{\diamond\}$.

Partial Words

- A *partial word* of length n over the alphabet A is a partial function $u\colon \{1,\ldots,n\} \xrightarrow{\circ} A$. For $i \in \{1,\ldots,n\}$, if $u(i)$ is defined we say that i *belongs to the domain of* u (denoted by $i \in D(u)$), otherwise we say that i *belongs to the set of holes of* u (denoted by $i \in H(u)$).
- Finite partial words are seen as words over the extended alphabet $A \cup \{\diamond\}$: a partial word u of length n is depicted as $u = a_1 \ldots a_n$, where $a_i = u(i)$, for $i \in D(u)$, and $a_i = \diamond$, otherwise. In this way, we define the catenation, respectively the equality, of partial words, as the catenation, respectively the equality, of the corresponding words over $A \cup \{\diamond\}$.
- If u and v are two partial words of equal length, then u *is said to be contained in* v, $u \subset v$, if all the elements of $D(u)$ are contained in $D(v)$ and $u(i) = v(i)$ for all $i \in D(u)$.

Partial Words

- A *partial word* of length n over the alphabet A is a partial function $u\colon \{1,\ldots,n\} \xrightarrow{\circ} A$. For $i \in \{1,\ldots,n\}$, if $u(i)$ is defined we say that i *belongs to the domain of* u (denoted by $i \in D(u)$), otherwise we say that i *belongs to the set of holes of* u (denoted by $i \in H(u)$).
- Finite partial words are seen as words over the extended alphabet $A \cup \{\diamond\}$: a partial word u of length n is depicted as $u = a_1 \ldots a_n$, where $a_i = u(i)$, for $i \in D(u)$, and $a_i = \diamond$, otherwise. In this way, we define the catenation, respectively the equality, of partial words, as the catenation, respectively the equality, of the corresponding words over $A \cup \{\diamond\}$.
- If u and v are two partial words of equal length, then u *is said to be contained in* v, $u \subset v$, if all the elements of $D(u)$ are contained in $D(v)$ and $u(i) = v(i)$ for all $i \in D(u)$.
- Two partial words u *and* v *are compatible*, $u \uparrow v$, if there exists a partial word w such that $u \subset w$ and $v \subset w$.

Partial Words

- A *partial word* of length n over the alphabet A is a partial function $u\colon \{1,\ldots,n\} \xrightarrow{\circ} A$. For $i \in \{1,\ldots,n\}$, if $u(i)$ is defined we say that i *belongs to the domain of* u (denoted by $i \in D(u)$), otherwise we say that i *belongs to the set of holes of* u (denoted by $i \in H(u)$).
- Finite partial words are seen as words over the extended alphabet $A \cup \{\diamond\}$: a partial word u of length n is depicted as $u = a_1 \ldots a_n$, where $a_i = u(i)$, for $i \in D(u)$, and $a_i = \diamond$, otherwise. In this way, we define the catenation, respectively the equality, of partial words, as the catenation, respectively the equality, of the corresponding words over $A \cup \{\diamond\}$.
- If u and v are two partial words of equal length, then u *is said to be contained in* v, $u \subset v$, if all the elements of $D(u)$ are contained in $D(v)$ and $u(i) = v(i)$ for all $i \in D(u)$.
- Two partial words u *and* v *are compatible*, $u \uparrow v$, if there exists a partial word w such that $u \subset w$ and $v \subset w$.
- We say that *the partial word* u *is a factor of the partial word* w if there exist partial words x and y such that $w = xuy$. If $w = a_1 \ldots a_n$, we denote by $w[i..j]$ the factor $a_i \ldots a_j$ of w, and by $w[i]$ the symbol a_i.

Combinatorial Properties of Partial Words

Let $w \in (A \cup \{\diamond\})^*$ be a partial word.

- *w is said to be a k-repetition* if $w = x_1 \dots x_k$ and there exists a non-empty partial word u such that $x_i \subset u$ for all $i \in \{1, \dots, k\}$.
- *w is said to be primitive* if it is not a k-repetition, for any $k > 1$.
- *w is said to be k-free* if it does not contain a k-repetition.
- A word of the form $x_1 y_1 x_2 y_2 x_3$ for which there exist two partial words x and y, with $|x| > 0$, such that $x_i \subset x$, for $i \in \{1, 2, 3\}$, and $y_j \subset y$, for $j \in \{1, 2\}$, is called overlap; *w is said to be overlap-free* if it does not contain an overlap.

Combinatorial Queries on Partial Words

Problem

Given a partial word w, with n symbols (labeled $w[1], \ldots, w[n]$), over the alphabet V, preprocess this partial word in order to answer the queries:

- *"is w[i..j] a k-repetition?", denoted* $\mathbf{rep}(i, j, k)$, *where* $i, j, k \in \{1, \ldots, n\}, i < j$.
- *"is w[i..j] k-free?", denoted* $\mathbf{free}(i, j, k)$, *where* $i, j, k \in \{1, \ldots, n\}, i < j$.
- *"is w[i..j] overlap-free?", denoted* $\mathbf{o\text{-}free}(i, j)$, *where* $i, j \in \{1, \ldots, n\}, i < j$.
- *"is w[i..j] primitive?", denoted* $\mathbf{prim}(i, j)$, *where* $i, j \in \{1, \ldots, n\}, i < j$.

Updates and Queries on Partial Words

Problem

Consider the following update operation for a partial word w: add a symbol $a \in V \cup \{\diamond\}$ at the rightmost end of w, to obtain wa. Preprocess w and define a method to update the data structures constructed during the preprocessing in order to answer in constant time **rep**, **free**, **o-free** *and* **prim** *queries, for a word obtained after several update operations were applied to w.*

1 Preliminaries

2 Definitions

3 Data Structures

4 Queries and Updates on Partial Words

5 Counting Problems

Constant Range Minimum Queries

Problem (Static)

Given an array T with n elements from a totally ordered set, and a natural constant L, preprocess this array in order to answer queries "find $\min pos_T(i, i+L-1)$*", where* $\min pos_T(i, i+L-1) = arg\min_{k \in \{i,\ldots,i+L-1\}} T[k]$ *(i.e.* $\min pos_T(i, i+L-1)$ *returns the position of the smallest value in the interval of T starting on position i and having length L: $T[i], T[i+1], \ldots, T[i+L-1]$); in case of multiple possible answers,* $\min pos$ *returns the rightmost (greatest) position where the smallest value in the interval is found.*

Constant Range Minimum Queries

Problem (Static)

Given an array T with n elements from a totally ordered set, and a natural constant L, preprocess this array in order to answer queries "find $\min pos_T(i, i+L-1)$*", where* $\min pos_T(i, i+L-1) = arg\min_{k \in \{i,\ldots,i+L-1\}} T[k]$ *(i.e.* $\min pos_T(i, i+L-1)$ *returns the position of the smallest value in the interval of T starting on position i and having length L: $T[i], T[i+1], \ldots, T[i+L-1]$); in case of multiple possible answers,* $\min pos$ *returns the rightmost (greatest) position where the smallest value in the interval is found.*

Problem (Dynamic)

We consider the following update operation for an array T: insert a number M at the end of the array T. Preprocess T and define an algorithmic method to update the data structures constructed during the preprocessing (if necessary, construct additional data structures), such that we can still answer $\min pos$ *queries, defined in the problem above, for an array obtained from T after an arbitrary sequence of update operations was applied to it.*

First Approach: The static problem

Problem (Static Range Minimum Query)

Given an array T with n elements from a totally ordered set, preprocess this array in order to be able to answer queries "find $\min pos_T(i,j)$*", where* $\min pos_T(i,j) = arg \min_{k \in \{i,\ldots,j\}} T[k]$*; in case of multiple possible answers, we assume that* $\min pos$ *returns the rightmost position where the smallest value in the interval is found.*

The most efficient solution of this problem requires $\mathcal{O}(n)$ preprocessing time, $\mathcal{O}(n)$ space to store the data structures constructed, and $\mathcal{O}(1)$ time to answer each query.

• H. N. Gabow, J. L. Bentley, R. E. Tarjan, *Scaling and related techniques for geometry problems*, Proc. 16th ACM STOC, 135–143, 1984.

• D. Harel, R.E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13, 338–355, 1984.

• E. Demaine, O. Weimann, *Advanced Data Structures, Lecture Notes from MIT*, Lecture 15,

`http://courses.csail.mit.edu/6.851/spring07/lec.html`, 2007.

First Approach: The dynamic problem

Problem (Dynamic RMQ)

We consider the following update operation for an array T: insert a number M at the end of the array T. Preprocess T and define an algorithmic method to update the data structures constructed during the preprocessing (if necessary, construct additional data structures), such that we can still answer minpos *queries for an array obtained from T after an arbitrary sequence of update operations was applied to it.*

The problem can be solved with $\mathcal{O}(\log_2 n)$ update time, in the worst case, and constant query time.
The solution can be particularized to solve the Dynamic Constant Range Minimum Query problem with $\mathcal{O}(\log_2 L)$ update time, in the worst case, and constant query time.

Second Approach: static and dynamic problems

• We use the linear data structure called *deque (double ended queue)*: a linear list where both the insertion (push) and deletion (pop) operation can be executed at any of the list's ends (*back* and *front*).

Second Approach: static and dynamic problems

• We use the linear data structure called *deque (double ended queue)*: a linear list where both the insertion (push) and deletion (pop) operation can be executed at any of the list's ends (*back* and *front*).
• The idea is to insert one by one, in increasing order, the numbers $\{1, 2, \ldots, n\}$ in the back end of a deque D_T, using the following algorithm:
– When i is inserted in the back end of the deque D_T, we pop from the front of the deque all the numbers k such that $k \leq i - L$; then, we pop from the back end of the deque all the numbers k that verify $T[k] \geq T[i]$;

Second Approach: static and dynamic problems

• We use the linear data structure called *deque (double ended queue)*: a linear list where both the insertion (push) and deletion (pop) operation can be executed at any of the list's ends (*back* and *front*).
• The idea is to insert one by one, in increasing order, the numbers $\{1, 2, \ldots, n\}$ in the back end of a deque D_T, using the following algorithm:
– When i is inserted in the back end of the deque D_T, we pop from the front of the deque all the numbers k such that $k \leq i - L$; then, we pop from the back end of the deque all the numbers k that verify $T[k] \geq T[i]$;
• What we get:
– Thus, D_T contains only numbers in the interval $[i - L + 1..i]$. If i and k are elements of D_T, such that k is closer to the front than i, then $k < i$ and $T[k] < T[i]$; thus, the element k at the front of D_T verifies $T[k] < T[j]$, for all j in D_T.
– After i was inserted D_T contains exactly all the elements k from the interval $[i - L + 1..i]$ that verify $T[k] < T[i]$, and is ordered increasingly.

Second Approach: static and dynamic problems

- To conclude: after i was inserted, the element in front of the deque is $\min pos_T(i - L + 1, i)$. Memorizing these values in an array enables us to answer $\min pos$ queries in constant time.

Second Approach: static and dynamic problems

- To conclude: after i was inserted, the element in front of the deque is $\min pos_T(i - L + 1, i)$. Memorizing these values in an array enables us to answer $\min pos$ queries in constant time.
- Every i, $1 \leq i \leq n$, is inserted once in the deque and popped at most once, thus the overall complexity of the preprocessing of the array T is $\mathcal{O}(n)$. This is, also, the total time needed to perform n insertions in the deque, therefore the amortized complexity of an insertion operation is $\mathcal{O}(1)$ (although a single insertion may require, in the worst case, L pops from the deque, thus $\mathcal{O}(L)$ time).

Second Approach: static and dynamic problems

- To conclude: after i was inserted, the element in front of the deque is $\min pos_T(i - L + 1, i)$. Memorizing these values in an array enables us to answer $\min pos$ queries in constant time.
- Every i, $1 \leq i \leq n$, is inserted once in the deque and popped at most once, thus the overall complexity of the preprocessing of the array T is $\mathcal{O}(n)$. This is, also, the total time needed to perform n insertions in the deque, therefore the amortized complexity of an insertion operation is $\mathcal{O}(1)$ (although a single insertion may require, in the worst case, L pops from the deque, thus $\mathcal{O}(L)$ time).
- An update of the array T leads to the insertion of a new element in the deque. This insertion is handled in the same way as in the preprocessing phase, since, basically, the algorithm presented above treats every element of the array T as an element that was newly inserted in that array, and updates the deque according to the value of this element.

Second Approach: static and dynamic problems

- To conclude: after i was inserted, the element in front of the deque is $\min pos_T(i - L + 1, i)$. Memorizing these values in an array enables us to answer $\min pos$ queries in constant time.
- Every i, $1 \leq i \leq n$, is inserted once in the deque and popped at most once, thus the overall complexity of the preprocessing of the array T is $\mathcal{O}(n)$. This is, also, the total time needed to perform n insertions in the deque, therefore the amortized complexity of an insertion operation is $\mathcal{O}(1)$ (although a single insertion may require, in the worst case, L pops from the deque, thus $\mathcal{O}(L)$ time).
- An update of the array T leads to the insertion of a new element in the deque. This insertion is handled in the same way as in the preprocessing phase, since, basically, the algorithm presented above treats every element of the array T as an element that was newly inserted in that array, and updates the deque according to the value of this element.
- The amortized cost of an insertion in the deque is $\mathcal{O}(1)$, thus an update requires $\mathcal{O}(1)$ amortized time (but up to $\mathcal{O}(L)$ time in the worst case). After this update was done, we can still answer queries

1 Preliminaries

2 Definitions

3 Data Structures

4 Queries and Updates on Partial Words

5 Counting Problems

rep queries

We define the matrix T, with n rows and $\lfloor n/2 \rfloor$ columns:

$$T[i][l] = \max\{m \mid m \leq \lfloor i/l \rfloor + 1, \text{ for which there exists } a \in V \text{ such that } w[i - jl] \subset a, \forall j \in \{0, \ldots, m-1\}\},$$

T is computed in time $\mathcal{O}(n^2)$ by dynamic programming.

Algorithm REP

1. *Construct the matrix T for the word w, as described above.*
2. *Solve the Static Constant RMQ Problem for each of the arrays $T[][l]$ and intervals of length l, with $l \in \{1, \ldots, \lfloor n/2 \rfloor\}$.*

q. *The answer to a query* **rep**(i, j, k) *is: if $j - i + 1$ is divisible by k, we compute $l = (j - i + 1)/k$; if $T[\text{minpos}_{T[][l]}(j - l + 1, j)][l] \geq k$ then the answer to the given query is* **yes***; otherwise the answer is* **no***.*

free queries

We define the matrix H, with n rows and columns, as follows:

$$H[i][j]=\begin{cases} k, & \text{where } k \text{ is the greatest natural number such that } w[i..j] \\ & \quad \text{contains a } k\text{-repetition, given that } i<j \\ 1, & \text{if } i=j \\ 0, & \text{if } i>j \end{cases}$$

We compute the elements of H by dynamic programming in $\mathcal{O}(n^2)$ time, if we are already able to answer **rep** queries.

Algorithm FREE

1. *Preprocess the partial word w as described in* **Algorithm REP**.
2. *Construct the matrix H for the partial word w, as described above.*

q. *The answer to a query* **free**(i,j,k) *is obtained as follows: if $H[i][j] \geq k$ then the answer is* **no**, *otherwise the answer is* **yes**.

o-free queries

We compute the matrix O with n rows and n columns, defined by:

$$O[i][j] = 1 \text{ if } w[i..j] \text{ contains an overlap, and } O[i][j] = 0 \text{ otherwise.}$$

The values stored in this matrix are computed in time $\mathcal{O}(n^2)$, by dynamic programming, if we are already able to answer **rep** queries..

Algorithm O-FREE

1. *Preprocess the partial word w as described in* **Algorithm REP**.
2. *Construct the matrix O for the partial word w, as described above.*

q. *The answer to a query* **o-free**(i, j) *is obtained as follows: if $O[i][j] = 0$ then the answer is* **yes**, *otherwise the answer is* **no.**

prim queries

We compute the matrix Pr with n rows and n columns, defined by:

$$Pr[i][j] = 1 \text{ if } w[i..j] \text{ is primitive, and } Pr[i][j] = 0 \text{ otherwise.}$$

To compute the value $Pr[i][j]$:

- We go through all the positive divisors d of $(j - i + 1)$ and check if $w[i..j]$ is a d-repetition using **rep** queries.
- If there exists at least one number d such that $w[i..j]$ is a d-repetition we set $Pr[i][j] = 0$; otherwise, we set $Pr[i][j] = 1$.

Algorithm PRIMITIVE

1. *Preprocess the partial word w as described in* **Algorithm REP**.
2. *Construct the matrix Pr, as described above.*

q. *The answer to a query* **prim**(i, j) *is obtained as follows: if $Pr[i][j] = 0$ then the answer is* **no**, *otherwise the answer is* **yes**.

prim queries - time complexity

A brute force implementation of the algorithm computing Pr achieves a time complexity of $\mathcal{O}(n^2\sqrt{n})$.
However, the number of divisors we need to analyze in order to compute all the values of Pr is $(\sum_{i=1}^{n}\sigma(i))$, where $\sigma(i)$ is the number of positive divisors of i.
And $\sum_{i=1}^{n}\sigma(i) \in \mathcal{O}(n\log_2 n)$, because $((\sum_{i=1}^{n}\frac{1}{i})/(\log_2 n))$ converges to a positive constant.
Therefore, the first step in the computation of the elements of the matrix Pr is to keep, in an array, the lists of divisors for every $l \in \{1,\ldots,n\}$.
This data structure will take $\mathcal{O}(n\log_2 n)$ space and can be computed in $\mathcal{O}(n\log_2 n)$ using a sieve method: we go through all the numbers i from 1 to n and add i to the lists of divisors of the numbers i, $2i$, ..., $\lfloor n/i \rfloor i$.

Then, we compute $Pr[i][j]$ as we described above, but searching the divisors of the length of a given factor in the newly computed array, and obtain the overall complexity $\mathcal{O}(n^2\log_2 n)$ for the computation of the elements of the matrix Pr.

Conclusion Queries

Theorem

A given partial word w, of length n, can be processed in time $\mathcal{O}(n^2)$, respectively $\mathcal{O}(n^2 \log_2 n)$, in order to be able to answer **rep**, **free**, **o-free** *and, respectively,* **prim** *queries, in time $\mathcal{O}(1)$.*

Updates

- Adding a symbol to the partial word w reflects in adding a row and a column to the matrix T, and updating the structures needed to answer min *pos* queries for each of its rows. Time: $\mathcal{O}(n \log_2 n)$ in the worst case (both approaches), $\mathcal{O}(n)$ amortized (second approach).

Updates

- Adding a symbol to the partial word w reflects in adding a row and a column to the matrix T, and updating the structures needed to answer min *pos* queries for each of its rows. Time: $\mathcal{O}(n \log_2 n)$ in the worst case (both approaches), $\mathcal{O}(n)$ amortized (second approach).
- Then, the matrices constructed for each type of queries (e.g., the matrix H. O, Pr constructed above) are updated accordingly. Each such update consists in adding at most one row and one column to each of these matrices (computed as in the static case). Time: $\mathcal{O}(n)$.
- We apply the same algorithms to answer queries in constant time.

Updates

- Adding a symbol to the partial word w reflects in adding a row and a column to the matrix T, and updating the structures needed to answer min *pos* queries for each of its rows. Time: $\mathcal{O}(n \log_2 n)$ in the worst case (both approaches), $\mathcal{O}(n)$ amortized (second approach).
- Then, the matrices constructed for each type of queries (e.g., the matrix H. O, Pr constructed above) are updated accordingly. Each such update consists in adding at most one row and one column to each of these matrices (computed as in the static case). Time: $\mathcal{O}(n)$.
- We apply the same algorithms to answer queries in constant time.
- The most time consuming part in the above updating the structures needed to answer min *pos* queries for the rows of the matrix T, but this can be done as described in the previous slides.

Conclusion Updates and Queries

Theorem

A given partial word w, of length n, can be processed in time $\mathcal{O}(n^2)$, respectively $\mathcal{O}(n^2 \log_2 n)$, in order to be able to answer **rep**, **free**, **o-free** *and, respectively,* **prim** *queries, in time $\mathcal{O}(1)$. If update operations, in which a new symbol is added to the rightmost end of w, are applied, the data structures constructed in the processing above can be updated in at most $\mathcal{O}(n \log_2 n)$ time (or $\mathcal{O}(n)$ amortized time), respectively at most $\mathcal{O}(n \log_2 n)$ time, per update, and still answer* **rep**, **free**, **o-free** *and, respectively,* **prim** *queries, in time $\mathcal{O}(1)$.*

Further Work in this Direction

- More efficient algorithms? (We conjecture that we can obtain linear worst case time complexity for the updates!)
- How the complexity of the preprocessing time decreases in the case of algorithms which answer queries in non-constant (yet small, for example logarithmic) time?
- Other update operations? (Replacing a $\diamond$ with a given symbol? Replacing a symbol with $\diamond$?)
- We can identify efficiently all the factors of the partial word w that are k-repetition, k-free, overlap-free and primitive. Thus we can count how many different partial words are factors of w and are also k-repetition, k-free, overlap-free and primitive.

Introduction: Subword Complexity

- Very simple problem for full words: count the number of distinct factors of a given word!

Introduction: Subword Complexity

- Very simple problem for full words: count the number of distinct factors of a given word! Efficient solution: linear time, using suffix arrays. We do not actually check all the factors, just count them!

Introduction: Subword Complexity

- Very simple problem for full words: count the number of distinct factors of a given word! Efficient solution: linear time, using suffix arrays. We do not actually check all the factors, just count them!
- In the case of partial words: count the number of distinct full words that are compatible with at least a factor of a given partial word! (for short: count the number of distinct factors of a partial word).

Introduction: Subword Complexity

- Very simple problem for full words: count the number of distinct factors of a given word! Efficient solution: linear time, using suffix arrays. We do not actually check all the factors, just count them!
- In the case of partial words: count the number of distinct full words that are compatible with at least a factor of a given partial word! (for short: count the number of distinct factors of a partial word).
  The full-words-approach doesn't seem to work: in that case we can order the words, here we can't. We conjecture that we cannot do much better than constructing all the words and counting the distinct ones.

Introduction: Subword Complexity

- Very simple problem for full words: count the number of distinct factors of a given word! Efficient solution: linear time, using suffix arrays. We do not actually check all the factors, just count them!
- In the case of partial words: count the number of distinct full words that are compatible with at least a factor of a given partial word! (for short: count the number of distinct factors of a partial word).
  The full-words-approach doesn't seem to work: in that case we can order the words, here we can't. We conjecture that we cannot do much better than constructing all the words and counting the distinct ones.
- Natural approach (compute subword complexity): count the number of the distinct full factors of a fixed length of the partial word, sum them up.

Introduction: Subword Complexity

- Very simple problem for full words: count the number of distinct factors of a given word! Efficient solution: linear time, using suffix arrays. We do not actually check all the factors, just count them!
- In the case of partial words: count the number of distinct full words that are compatible with at least a factor of a given partial word! (for short: count the number of distinct factors of a partial word).
  The full-words-approach doesn't seem to work: in that case we can order the words, here we can't. We conjecture that we cannot do much better than constructing all the words and counting the distinct ones.
- Natural approach (compute subword complexity): count the number of the distinct full factors of a fixed length of the partial word, sum them up.
- Other problems: count the number of the distinct full words of length L, compatible with a word from a list of partial words of length L, count the number of distinct k-repetitions full words which are compatible with a factor of a given partial word w (for example distinct squares), etc.

Initial Problems

Problem (1)

Given a list of partial words $S = \{w_1, w_2, \ldots, w_k\}$, over alphabet V, each partial word having the same length L, find a word $v \in V^L$ such that v is not compatible with any of the words in S.

This problem is *NP*-Complete (reduction from CNF-SAT). As a consequence:

Problem (2)

Given a list of partial words $S = \{w_1, w_2, \ldots, w_k\}$, over alphabet V, each partial word having the same length L, count the distinct words $v \in V^L$ such that v is compatible with at least one of the words in L.

Problem 1 is $\#P$-Complete.

Further Problems

Problem (3)

Given a partial word w, over alphabet V, find a word $v \in V^L$ such that v is not compatible with any of the factors of length L of w.

This problem is NP-Complete (reduction from Problem 1). As a consequence:

Problem (4)

Given a partial word w, over alphabet V, count the distinct words $v \in V^L$ such that v is compatible with at least a factor of length L of w.

Problem 4 is $\#P$-Complete, thus it is hard to compute the subword complexity of a partial word.

$\#P$-Complete Problems that don't have an associated NP-Complete Problem

Problem (5)

Given a partial word w, over alphabet V, count the number of distinct k-repetitions full words which are compatible with a factor of w.

Problem (6)

Given a partial word w, over alphabet V, count the distinct full factors, over $V \setminus \{\#\}$ (where $\#$ is a symbol of $alph(w)$), of w.

Problem 5 and 6 are $\#P$-Complete (Turing reductions from Problem 3).

Open Problems

Problem (?)

Given a partial word w, over alphabet V, count the distinct full factors of w.

We conjecture that Problem ? is $\#P$-Complete.
Of course, it is interesting how complex would be to count full factors of w that verify other combinatorial properties (overlap-free, k-free, primitive, etc.). However, we strongly believe that these problems are also $\#P$-Complete.

THANK YOU!

The speaker's attendance at this conference was sponsored by the *Alexander von Humboldt Foundation*.