Theoretical Computer Science 638 (2016) 91–97

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Fast construction of wavelet trees ☆

J. Ian Munro [a], Yakov Nekrich [a,*], Jeffrey S. Vitter [b]

[a] Cheriton School of Computer Science, University of Waterloo, Canada
[b] Department of Electrical Engineering & Computer Science, University of Kansas, United States

ARTICLE INFO

Article history:
Received 31 March 2015
Received in revised form 10 October 2015
Accepted 8 November 2015
Available online 1 December 2015

Keywords:
Wavelet trees
Compressed data structures
Compressed sequences

ABSTRACT

In this paper we describe a fast algorithm that creates a wavelet tree for a sequence of symbols. We show that a wavelet tree can be constructed in $O(n\lceil \log\sigma/\sqrt{\log n}\rceil)$ time where n is the number of symbols and σ is the alphabet size.

1. Introduction

Wavelet tree, introduced in [1], is one of the most extensively studied succinct data structures. Wavelet trees are frequently chosen as a space-efficient data structure that supports access, rank and select queries on a sequence of symbols. An access query access(i, X) returns the i-th symbol in a sequence X; a rank query $\text{rank}_a(i, X)$ computes how many times a symbol a occurs in the prefix $X[1..i]$ of X; select query $\text{select}_a(i, X)$ finds the position where a occurs for the i-th time. Since wavelet trees can efficiently support operations rank and select, they can be used in succinct representations of graphs [2], strings, points and other geometric objects on a grid, full-text indexes [1,3], data structures for document retrieval [4], XML documents [5], and binary relations [6]. It was also shown that wavelet trees and their variants can be used to answer various queries on points and other geometric objects [7]. We refer to recent extensive surveys of Navarro [8] and Makris [9] for a description of these and other applications of wavelet trees. In this paper we describe the first algorithm that constructs a wavelet tree in $o(n\log\sigma)$ time. nº We show how to construct a wavelet tree in $O(n\lceil \frac{\log\sigma}{\sqrt{\log n}}\rceil)$ time.

Let X be a sequence of length n over an alphabet of size σ. We can assume w.l.o.g. that the i-th element $X[i]$ of X is an integer in the range $[1, \sigma]$. Essentially constructing a wavelet tree for a sequence X requires re-grouping the bits of X into a bit sequence of total length $n\log\sigma$. Since different bits of an element $X[i]$ are stored in different parts of the bit sequence, it appears that we need $\Omega(n\log\sigma)$ time to construct a wavelet tree. In this paper we show that the cost of the straightforward solution can be reduced by an $O(\sqrt{\log n})$ factor. The main idea of our method is usage of bit parallelism, i.e. we use bit operations to keep $\Omega(1)$ elements of X in one word and perform certain operations on elements packed into one word in constant time. Suppose that we can pack L symbols of a sequence X into one machine word. Then we can generate the wavelet tree for the resulting sequence of symbols in $O(n(\log\sigma/L))$ time by processing $O(L)$ symbols in constant time.

☆ An early version of this work appeared in SPIRE 2014.

* Corresponding author.
E-mail addresses: imunro@uwaterloo.ca (J. Ian Munro), ynekrich@uwaterloo.ca (Y. Nekrich), jsv@ku.edu (J.S. Vitter).

http://dx.doi.org/10.1016/j.tcs.2015.11.011
0304-3975/

Previous and related work. Since wavelet trees were introduced in 2003 [1], a large number of papers that use this data structure appeared in the literature [3,10–16]. A more extensive list of previous results can be found in surveys of Makris [9] and [8]. In spite of a significant number of previous papers, no results for constructing a wavelet tree in $o(n \log \sigma)$ time were previously described. Algorithms that generate a wavelet tree and use little additional workspace were considered by Claude et al. [17] and Tischler [18].

Chazelle [19] described a linear space ($O(n \log n)$-bit) geometric data structure that answers certain kinds of two-dimensional range searching queries. Data organization in [19] is the same as in wavelet tree. nº quite similar to the approach of wavelet trees. We remark, however, that the intended usage of the wavelet tree and Chazelle's data structure are different. The data structure of Chazelle [19] supports different kinds of geometric queries and uses $O(n \log n)$ space to store n two-dimensional points. On the other hand, the wavelet tree, as described in [1] and later works, uses $n \log \sigma$ bits to store a sequence of size n over an alphabet of size σ; the space usage can also be reduced to nH_0 bits, where H_0 is the zero-order entropy of the original sequence. Some other linear-space geometric data structures [20] also use similar ways of structuring data. By the same argument, we need $O(n \log n)$ time to construct these data structures. Chan and Pătraşcu [21] showed that bit parallelism can be used to obtain linear-space data structures with faster construction time. In [21] they describe data structures that use linear space and can be constructed in $O(n\sqrt{\log n})$ time. Their approach is based on recursively reducing the original problem to several problems of smaller size. When point coordinates are sufficiently small, we can pack L points into one machine word and process data associated to L points in constant time. Very recently, the problem of constructing a wavelet tree was addressed by Babenko et al. [22]; the result presented in [22] and published after the conference version of this paper, is equivalent to our result.

In this paper we show how bit parallelism can be applied to speed-up the construction of the standard wavelet tree data structure. Our simple two-stage approach improves the construction time of the wavelet tree by $O(\sqrt{\log n})$. After recalling the basic concepts in Section 2, we describe the main algorithm and its variants in Section 3. In Section 4 we show how we can construct secondary data structures stored in the wavelet tree nodes. Finally, in Section 5 we show how our result can be used to speed-up the construction algorithm for a geometric data structure that answers two-dimensional orthogonal range maxima queries.

2. Wavelet tree

Let X denote a sequence over alphabet $\Sigma = \{1, \ldots, \sigma\}$. The standard wavelet tree for X is a balanced binary tree with bit sequences stored in each internal node. These bit sequences can be obtained as follows: we start by dividing the alphabet symbols into two subsets Σ_0 and Σ_1 of equal size, $\Sigma_0 = \{1, \ldots, \sigma/2\}$ and $\Sigma_1 = \{\sigma/2+1, \ldots, \sigma\}$. Let X_0 and X_2 denote the subsequences of X induced by symbols from Σ_0 and Σ_1 respectively. The bit sequence $X(v_R)$ stored in the root v_R of the wavelet tree indicates for each symbol $X[i]$ whether it belongs to X_0 or X_1: $X(v_R)[i] = 0$ if $X[i]$ is in X_0 and $X(v_R)[i] = 1$ if $X[i]$ is in X_1. The left child of v_R is the wavelet tree for X_0 and the right child of v_R is the wavelet tree for X_1.

A symbol from an alphabet Σ can be represented as a bit sequence of length $\lfloor \log \sigma \rfloor$ or $\lceil \log \sigma \rceil$. Bit sequences $X(u)$ in the nodes of the wavelet tree consist of the same bits as the symbols in X, but the bits are ordered in a different way. The sequence $X(v_R)$ contains the first bit from each symbol $X[i]$ in the same order as symbols appear in X. Let v_l and v_r be the left and the right children of v_R. The sequence $X(v_l)$ contains the second bit of every symbol in X_0. That is, $X(v_l)$ contains the second bit of every symbol $X[i]$, such that the first bit of $X[i]$ is 0. $X(v_r)$ contains the second bit of every $X[i]$ such that the first bit of $X[i]$ is 1, etc.

Some generalizations of the wavelet tree often lead to improved results. We can consider t-ary wavelet tree for $t = \log^{\varepsilon} n$ and a small constant $\varepsilon > 0$. In this case the original alphabet Σ is divided into t parts $\Sigma_0, \ldots, \Sigma_{t-1}$. The sequence $X(v_R)$ in the root node is a sequence over an alphabet $\{0, \ldots, t-1\}$ such that $X(v_R)[i] = j$ iff $X[i]$ is a symbol from Σ_j for $1 \le j \le t$. Let X_j be the subsequence of X induced by symbols from Σ_j. The j-th child v_j of v_R is the root of the wavelet tree for X_j. The advantage of the t-ary wavelet tree is that the tree height is reduced from $O(\log \sigma)$ to $O(\log \sigma / \log \log n)$. Another useful improvement is to modify the shape of the tree so that the average leaf depth is (almost) minimized. Finally we can also keep the binary or t-ary sequences $X(u)$, stored in the nodes, in compressed form. Two latter improvements enable us to store a sequence X in asymptotically optimal space.

3. Constructing a wavelet tree

In this section we describe our algorithm for constructing a wavelet tree. Our method uses bit parallelism in a way that is similar to [21]. However a recursive algorithm employed in [21] to reduce the problem size is not necessary. Our algorithm consists of two stages. During the first stage we construct an L-ary wavelet tree $\mathcal{T}^g$ for $L = 2^{\sqrt{\log n}}$. That is, each internal node $u \in \mathcal{T}^g$ has L children. To avoid tedious details, we assume that L is an integer that divides σ. An L-ary wavelet tree can be defined in the same way as in Section 2. We partition the alphabet $\Sigma = \{1, \ldots, \sigma\}$ into L parts $\Sigma_1, \Sigma_2, \ldots, \Sigma_L$. Each Σ_i for $1 \le i \le L-1$ contains σ/L alphabet symbols; the last part Σ_L contains at most σ/L symbols. The root node u_R of $\mathcal{T}^g$ contains a sequence $X^g(u_R)$. Every element of $X^g(u_R)$ is a positive integer that does not exceed L. $X^g(u_R)[i] = j$ if $X[i]$ is a symbol from Σ_j. The child u_i of u is the root node of the wavelet tree for the subsequence X_i, where X_i is the subsequence of X induced by symbols from Σ_i. An L-ary tree can be constructed in $O(\log \sigma / L)$ time. During the second stage, we transform an L-ary tree into a binary tree. We replace each internal node u of $\mathcal{T}^g$ with a subtree $T(u)$ of height

$\ell = \log L$. $T(u)$ has at most $L-1$ internal nodes; leaves of $T(u)$ correspond to children of u in $\mathcal{T}^g$. If the sequence $X^g(u)$ contains m elements, then all binary sequences $X(v)$ in the nodes $v \in T(u)$ contain $m\ell$ nits. Since we can pack ℓ elements of $X^g(u)$ into one word, $T(u)$ can be constructed in $O(m)$ time. A more technical description is provided below. We start by showing in Lemma 1 how the wavelet tree can be constructed in linear time when elements are bounded by L. Then we show in Theorem 1 how a binary wavelet tree for any sequence X can be constructed following the method outlined above. The result for a balanced binary wavelet tree can be easily extended to a t-ary tree of an arbitrary shape. Finally we can also obtain the original sequence X from its wavelet tree by reversing the algorithm that constructs the wavelet tree.

Lemma 1. *Let X be a sequence of L positive integers such that $L \le 2^{\sqrt{\log n}}$ and $X[i] \le 2^{\sqrt{\log n}}$ for all i, $1 \le i \le L$. A binary balanced wavelet tree for X can be constructed in $O(L)$ time using workspace $O(L)$. The algorithm employs a universal look-up table of $o(n)$ bits.*

Proof. We start by constructing a packed sequence $\overline{X}$; $\overline{X}$ consists of $\lceil L/\ell \rceil$ words and every word contains $\ell = \sqrt{\log n}$ elements of X. We initialize $\overline{X}(u_R) = \overline{X}$ for the root node u_R and visit all nodes in the depth-first order. When a node u is visited, we traverse $\overline{X}(u)$ and construct the bit sequence $X(u)$ that must be stored in the root u of the wavelet tree. We extract the first bit from each $\overline{X}[i]$ and append it to the end of $X(u)$. We also produce two sequences $\overline{X}(u_l)$ and $\overline{X}(u_r)$ unless u is a leaf node. If the first bit in $\overline{X}(u)[i]$ is 0, we append the value v to the end of $\overline{X}(u_l)$, where v is $\overline{X}(u)[i]$ without the first bit; if the first bit in $\overline{X}(u)[i]$ is 1, we append v to the end of $\overline{X}(u_r)$. Sequences $\overline{X}(u_l)$ and $\overline{X}(u_r)$ are also stored in packed form. When $X(u)$, $\overline{X}(u_l)$ and $\overline{X}(u_r)$ are generated, we can discard $\overline{X}(u)$.

The key observation is that each $\overline{X}(u)$ can be processed in $O(\lceil |\overline{X}(u)|/\ell \rceil)$ time using universal look-up tables $\mathbb{T}$ and $\mathbb{T}_1$. For any sequence of $\ell/4$ elements $\overline{Y}[1] \dots \overline{Y}[\ell/4]$ of $p \le \ell$ bits each, $\mathbb{T}$ can output (i) the bit sequence $Y[1] \dots Y[\ell/4]$, where $Y[i]$ is the first bit of $\overline{Y}[i]$ (ii) sequences $\overline{Y}_l$ and $\overline{Y}_r$ defined below. The sequence $\overline{Y}_l$ contains elements $\overline{Y}[i]$ whose first bit is 0 in the same order as in $\overline{Y}$; the sequence $\overline{Y}_r$ contains elements $\overline{Y}[i]$ whose first bit is 1 in the same order as in $\overline{Y}$. Another look-up table, $\mathbb{T}_1$ can produce for any sequence $\overline{Y}[1] \dots \overline{Y}[\ell/4]$ a sequence $\overline{Z}[1] \dots \overline{Z}[\ell/4]$, where $\overline{Z}[i]$ equals to $\overline{Y}[i]$ without the first bit. Using these two look-up tables, we can read $\ell/4$ elements of $\overline{X}(u)$ and produce the next $\ell/4$ elements of $X(u)$, $\overline{X}(u_l)$, and $\overline{X}(u_r)$ in $O(1)$ time. $\mathbb{T}$ and $\mathbb{T}_1$ contain one entry for each p, $1 \le p \le \ell$, and for each sequence of $\ell/4$ integers of p bits each. Hence both tables have $O(n^{1/4})$ entries and use $o(n^{1/2})$ bits. Since we spend $O(\lceil |X(u)|/\ell \rceil)$ time in each node u and the total length of all $X(u)$ is $O(L\ell)$, we can construct the binary wavelet tree for X in $O(L)$ time. □

Theorem 1. *Let X be a sequence of n positive integers such that $1 \le X[i] \le \sigma$ for $1 \le i \le n$. A binary balanced wavelet tree for X can be constructed in $O(n\lceil \frac{\log \sigma}{\sqrt{\log n}} \rceil)$ time.*

Proof. We employ the two-stage procedure described at the beginning of this section. During the first stage we construct a wavelet tree with node degree $L = 2^{\sqrt{\log n}}$. We will consider elements of X as binary sequences of length σ. For an integer v, we denote by $v.\mathrm{bits}(a..b)$ the bit sequence obtained by extracting bits at positions a, $a+1$, …, b from v (bit positions are in the left-to-right order so that the most significant bit is at position 1). The process of recursive alphabet division can be re-formulated as recursive division of symbols according to their prefixes. That is, elements of X are distributed among 2^ℓ subsequences according to their prefixes of length $\ell = \sqrt{\log n}$. Each subsequence is further divided into 2^ℓ subsequences, etc. Let $\ell = \sqrt{\log n}$. We process the sequence X and generate sequences X_α. Initially all X_α are empty. For every $j = 0, 1, \dots, \lceil \log \sigma / \sqrt{\log n} \rceil - 1$ we append $X[i].\mathrm{bits}(j\ell+1..j(\ell+1))$ to the sequence X_α for $\alpha = X[i].\mathrm{bits}(1..j\ell)$. Sequences X_α are stored in an L-ary wavelet tree $\mathcal{T}^g$. First ℓ bits of each $X[i]$ are kept in a sequence X_ϵ for an empty string ϵ. X_ϵ is stored in the root node that has 2^ℓ children labeled with bit sequences of length ℓ. The child that is labeled with α contains the sequence X_α. Every internal node also has 2^ℓ children that are labeled by bit sequences of length ℓ. Thus there are ℓ^i nodes of depth i. A node u of depth i contains the sequence X_α where α is the concatenation of node labels on the path from the root to u. We spend $O(n\lceil \frac{\log \sigma}{\sqrt{\log n}} \rceil)$ time to produce T^g.

It remains to show how to construct a binary wavelet tree for each X_α. We divide X_α into subsequences $X_{\alpha,i}$ for $i = 1, \dots, \lceil |X_\alpha|/2^{\sqrt{\log n}} \rceil$, where $|X_{\alpha,i}| = 2^{\sqrt{\log n}}$ for $1 \le i \le \lfloor |X_\alpha|/2^{\sqrt{\log n}} \rfloor$ and $|X_{\alpha,i}| \le 2^{\sqrt{\log n}}$ for $i = \lceil |X_\alpha|/2^{\sqrt{\log n}} \rceil$. Then we apply Lemma 1 to each $X_{\alpha,i}$. □

The result of Theorem 1 can be easily extended to the case when the wavelet tree has an arbitrary shape.

Theorem 2. *Let X be a sequence of n positive integers such that $1 \le X[i] \le \sigma$ for $1 \le i \le n$. Any binary wavelet tree for X can be constructed in time $O(n\lceil \frac{h}{\sqrt{\log n}} \rceil + \sigma)$ where h is the average leaf depth.*

Proof. We assume in this theorem that the shape of the wavelet tree is already known. Let the *codeword* for a symbol $a \in \Sigma$ denote the bit string α obtained by following the path from the root to the leaf that contains a; we start with an empty string α and append 0 (1) to α every time when the left (respectively, the right) edge is taken. We start by replacing each element $X[i]$ with its codeword. Then we proceed exactly as in Theorem 1. If the codeword for a symbol $X[i]$ is of

length $l[i]$, then the bits of $X[i]$ will be stored at $\lceil l[i]/\sqrt{\log n}\rceil$ nodes of the L-ary wavelet tree. Hence the first stage takes $O(n\lceil \frac{h}{\sqrt{\log n}}\rceil)$ time and the total number of symbols in all sequences X_α is also $O(n\lceil \frac{h}{\sqrt{\log n}}\rceil)$. We showed in Theorem 1 that a wavelet tree for each X_α is constructed in linear time. Hence node u of an L-ary wavelet tree is transformed into a binary tree in $O(|X(u)|)$ time, where $|X(u)|$ denotes the number of symbols in $X(u)$. Hence the total time to construct the wavelet tree is $O(n\lceil \frac{h}{\sqrt{\log n}}\rceil)$. □

Besides our algorithm can be also modified for the case when the wavelet tree has arity $\log^\alpha n$ for a small constant α.

Theorem 3. *Let X be a sequence of n positive integers such that $1 \le X[i] \le \sigma$ for $1 \le i \le n$. A wavelet tree with node degree $\log^\alpha n$ for the sequence X can be constructed in time $O(n\lceil \frac{h\alpha \log\log n}{\sqrt{\log n}}\rceil + \sigma)$ where h is the average leaf depth.*

Proof. Let $\mathcal{T}$ denote the wavelet tree to be constructed. We extend $\mathcal{T}$ to a binary tree $\mathcal{T}^E$ by inserting some dummy nodes. Each node $u \in \mathcal{T}$ with descendants $u_1, \ldots, u_d$ is extended to a full binary tree[1] of height $\alpha \log\log n$ with root u and leaves $u_1, \ldots, u_d$. The nodes of the original tree will be called *data nodes*; all other nodes will be called *auxiliary nodes*. Our procedure constructs wavelet tree $\mathcal{T}^E$ in the same way as in Theorem 2, but we generate sequences $X(u)$ only for the data nodes u. Suppose that we visit a node and generate sequences $\overline{X}(u_l)$, $\overline{X}(u_r)$. If u_l and u_r are auxiliary nodes, then $\overline{X}(u_l)$ and $\overline{X}(u_r)$ contain the elements of $\overline{X}(u)$; unlike Theorem 2, the leftmost bits of $\overline{X}(u)$ are not removed. We simply assign the elements of $\overline{X}(u)$ to $\overline{X}(u_l)$ and $\overline{X}(u_r)$ according to the t-th bit of $\overline{X}(u)$ where t is the distance from u_l to its lowest ancestor that is a data node. If u_l and u_r are data nodes, we generate $X(u_l)$ and $X(u_r)$ according to the value of the d-th bit in $\overline{X}(u)$ for $d = \alpha \log\log n$; depending on the value of the d-th bit in $\overline{X}(u)$ we append $\mathrm{lshift}(X(u)[i])$ to $X(u_l)$ or $X(u_r)$, where $\mathrm{lshift}(v)$ denotes the value of v with $\alpha \log\log n$ leftmost bits removed. If a data node u is visited, we also generate a sequence $X(u)$ such that $X(u)[i] = \overline{X}(u)[i].\mathrm{bits}(1..\alpha \log\log n)$. That is, we retrieve the $\alpha \log\log n$ leftmost bits from each $\overline{X}(u)[i]$ and store them in $X(u)$; we note that $\overline{X}(u)[i]$ do not change when u is visited. □

Finally we can also restore the original sequence X from its wavelet tree.

Theorem 4. *We can obtain a sequence X from its binary wavelet tree $\mathcal{T}$ in $O(n\frac{h}{\sqrt{\log n}} + \sigma)$ time, where h is the average leaf depth and n is the length of X. We can obtain a sequence X from its wavelet tree $\mathcal{T}$ with node degree $\log^\alpha n$ in $O(n\frac{h\alpha \log\log n}{\sqrt{\log n}} + \sigma)$ time.*

Proof. We say that a node $u \in \mathcal{T}$ is special, if its depth is divisible by $\ell = \sqrt{\log n}$. Our algorithm consists of two stages. First, we create sequences $\overline{X}(u)$ stored in special nodes u, so that each $\overline{X}(u)[i]$ is an integer of at most ℓ bits. That is, we turn $\mathcal{T}$ into a wavelet tree $\mathcal{T}_b$ such that each internal node of $\mathcal{T}_b$ has up to 2^ℓ children. nº and the maximal leaf depth in $\mathcal{T}_b$ is bounded by $O(\frac{\log\sigma}{\sqrt{\log n}})$. The total bit length of all sequences stored in the nodes of $\mathcal{T}_b$ equals the total bit length of all sequences stored in the nodes of $\mathcal{T}$. Thus the total space usage does not increase. The procedure for converting $\mathcal{T}$ into $\mathcal{T}_b$ works as follows. For every node u, such that both its children are special nodes, we assume that $\overline{X}(u) = X(u)$. Then we work up the tree and produce $\overline{X}(v)$ for ancestors v of u until a special node is reached. Suppose that sequences $\overline{X}(u_l)$ and $\overline{X}(u_r)$ for children of a node w are already produced. We generate $\overline{X}(w)$ according to the following rule: if $X(w)[i] = 0$, then $\overline{X}(w)[i] = 0\overline{X}(u_l)[i]$; if $X(w)[i] = 1$, then $\overline{X}(w)[i] = 1\overline{X}(u_r)[i]$. The total time to construct $\mathcal{T}_b$ is $O(n\frac{h}{\sqrt{\log n}} + \sigma)$.

Finally we collect the values of $\overline{X}(u)[i]$ in special nodes u and obtain the sequence of integers X by concatenating those values. The procedure starts in the root node u_R of $\mathcal{T}_b$. Depending on the value of $\overline{X}(u_R)[i]$ we visit the corresponding child u of u_R in $\mathcal{T}_b$ (we observe that u_R can have up to $2^{\sqrt{\log n}}$ children) and retrieve the next element e_u in $\overline{X}(u)$. Then we replace $\overline{X}(u_R)[i]$ with the concatenation of $\overline{X}(u_R)[i]$ and e_u. Proceeding in the same way for nodes of $\mathcal{T}_b$ on all levels, we obtain the values of the original sequence X in $\overline{X}(u_R)$. □

4. Rank and select queries in wavelet trees

In this section we consider the problem of storing a sequence $S = s_1 s_2 \ldots s_n$ over an alphabet σ that supports the following queries:

- $\mathrm{access}(i, S)$ returns $S[i]$
- $\mathrm{rank}_a(i, S)$ computes the number of times a occurs in $S[1..i] = s_1 \ldots s_i$
- $\mathrm{select}_a(i, S)$ finds the position j of the i-th occurrence of a, i.e., $\mathrm{select}_a(i, S) = j$ such that $\mathrm{access}(j, S) = a$ and $\mathrm{rank}_a(j, S) = i$.

[1] To simplify the description we assume that $\log^\alpha n$ is a power of 2.

Wavelet trees support rank, select, and access queries in $O(\log\sigma)$ time. This is achieved by augmenting sequences $X(u)$, stored in the nodes, with data structures that answer rank and select queries. If queries on sequences $X(u)$ are answered in constant time, then queries on the original sequence X are answered in $O(\log\sigma)$ time. The sequence $X(u)$ stored in a node of a binary wavelet tree is a sequence of bits. In the case of a t-ary wavelet tree, sequences $X(u)$ are over an alphabet of size t. Thus a wavelet tree reduces rank, select, access queries on a sequence X to $O(\log\sigma)$ queries on binary sequences or $O(\log\sigma/\log t)$ queries on t-ary sequences. For details we refer to e.g., [8]. It remains to show how data structures for $X(u)$ can be constructed quickly.

Rank and select on binary and t-ary sequences. We will describe below several results on constructing rank-select data structures for sequences over a small alphabet. We remark that the data structures are not new and are based on standard techniques. However, we show that these data structures can be constructed in less than linear time, provided that the original sequence is available in packed form.

We show in the following Theorem that the data structure of Jacobson [23] can be constructed in $O(m/\log n)$ time.

Theorem 5. *A bit sequence B of length m can be stored in data structure that answers* rank, select, *and* access *queries in constant time. This data structure uses $m + O(m\log\log n/\log n)$ bits and can be constructed in $O(m/\log n)$ time. The construction algorithm relies on a universal table of size $o(n)$.*

Proof. B is divided into blocks of $d_1 = \log^2 n$ bits. We compute and store the number of 0's and the number of 1's in the first i blocks for $i = 1,\ldots,m/\log^2 n$. Since the number of blocks is $O(m/\log^2 n)$, this information takes $O(m/\log n)$ bits. We assume that B is kept in packed form, so that this information can be computed in $O(m/\log n)$ time. Each block is divided into sub-blocks of size $d_2 = \log n/2$. We compute and store the number of 0's and the number of 1's in the first j sub-blocks of a block for each block and for $j = 1,\ldots,2\log n$. The number of 0's or 1's in a block is at most $\log^2 n$. Hence, we can keep information about 0 and 1's in the first j sub-blocks of a block in $O(\log\log n)$ bits. The total number of sub-blocks is $O(m/\log n)$. Hence, all sub-block counts take $O(m\log\log n/\log n)$ bits.

Using a pre-computed universal table of size $O(n^{1/2}\log n)$ we can find the number of 0's and the number of 1's within the first t positions of a sub-block for $t = 1,2,\ldots,\log n/2$ in $O(1)$ time. A rank query $\mathrm{rank}_0(i,B)$ is answered by finding the block j_1 that contains the i-th bit and the sub-block j_2 within the j_1-th block that contains the i-th bit. We also find the position j_3 of the i-th bit within that sub-block. Now $\mathrm{rank}_0(i,B) = c_1 + c_2 + c_3$ where c_1 is the number of 0's in the first $j_1 - 1$ blocks, c_2 is the number of 0's in the first $j_2 - 1$ sub-blocks of the j_1-th block, and c_3 is the number of 0's among the first j_3 bits in the j_2-nd sub-block of the j_1-th block. Queries $\mathrm{rank}_1(i,B)$ are answered in the same way.

The data structure for rank queries can be created in $O(m/\log n)$ time. Using the same universal table, we can compute the number of 0's and the number of 1's in each sub-block in $O(1)$ time. Using this information, we count the number of 0's in the first t sub-blocks of each block for $t = 1,\ldots,\log^2 n$. Then we count the number of 0's in the first i blocks for $i = 1,2,\ldots,m/\log^2 n$. The total number of sub-blocks in all blocks is $O(m/\log n)$ and the total number of blocks is $O(m/\log^2 n)$. Since we spend $O(1)$ time in every sub-block, auxiliary data structures for rank queries can be computed in $O(m/\log n)$ time.

The data structure for select queries is based on a similar approach. Suppose that we want to answer queries $\mathrm{select}_0(i,B)$. We divide B into chunks, so that each chunk contains $\log^2 n$ 0-bits. If the size of a chunk exceeds, $\log^4 n$, we say that this chunk is sparse; otherwise a chunk is dense. We keep left boundaries of each chunk in an array. If a chunk is sparse, we also keep the position of the t-th 0-bit in that chunk for $t = 1,\ldots,\log n$. A dense chunk is divided into words, so that each word consists of $\log n/2$ bits. We keep the number of 0-bits in every word in a data structure M. The number of 0-bits in every word is at most $\log n/2$, the number of words in a chunk is $O(\log^3 n)$. We can implement M so that it uses $O(\log\log n)$ bits per word; moreover we can find the word that contains the t-th 0-bit in a block and the number of 0-bits in the first d words for any t, d in time $O(1)$.

A query $\mathrm{select}_0(i,B)$ is answered by finding the starting position of the i_0-th chunk for $i_0 = \lfloor i/\log n\rfloor$. Let $i_1 = i - i_0\cdot\log n$. Clearly $\mathrm{select}_0(i,S) = j$ where j is the position of the i_1-th 0 in the i_0-th chunk. If this chunk is sparse, then the position of the i_1-th 0 is stored. Otherwise we find the word W_j that contains the i_1-th 0-bit using M. We can find the position of the i_1-th bit in W_j using a universal look-up table.

There are $O(n/\log^2 n)$ chunks and $O(n/\log^4 n)$ sparse chunks. The number of 0-bits in sparse chunks is $O(n/\log^2 n)$; hence, we can store positions of all 0-bits in $O(n/\log n)$ bits. We can create the array that contains left boundaries of all chunks and positions of 0-bits in all chunks in $O(m/\log n)$ time. The time needed to create data structures M for all dense chunks is proportional to the number of words in all dense chunks. Hence all M are created in $O(m/\log n)$ time.

Data structures that support rank_1 and select_1 on B are implemented in the same way. □

Theorem 6. *A bit sequence B of length m can be stored in data structure that answers* rank, select, *and* access *queries in constant time. This data structure uses $mH_0(B) + O(m\log\log n/\log n)$ bits and can be constructed in $O(m/\log n)$ time, where $H_0(B)$ is the zero-order entropy of B. The construction algorithm relies on a universal table of size $o(n)$.*

Proof. The only difference is that the bit sequence B itself is stored in compressed form. We employ the method of Raman et al. [24] that splits the sequence into pieces of size $\Theta(\log n)$ and keeps all pieces in $mH_0(B) + O(m\log\log n/\log n)$ bits. □

Theorem 7. *A sequence B of length m over an alphabet $\{1, 2, \ldots, \log^{\varepsilon} n\}$, where $\varepsilon > 0$ is a constant, can be stored in data structure that answers* rank, select, *and* access *queries in constant time. This data structure uses $mH_0(B) + O(m \log^{\varepsilon} n \log\log n / \log n)$ bits and can be constructed in $O(m/\log^{1-\varepsilon} n)$ time, where $H_0(B)$ is the zero-order entropy of B. The construction algorithm relies on a universal table of size $o(n)$.*

Proof. We keep auxiliary structures that answer rank_a and select_a for every a such that $1 \le a \le \log^{\varepsilon} n$. These data structures are implemented as in Theorems 5 and 6 and need $O(m(\log\log n/\log n))$ bits. All auxiliary data structures use $O(m(\log\log n/\log^{1-\varepsilon} n))$ bits. Since data structures for a fixed symbol a can be constructed in $O(m/\log n)$ time, all auxiliary structures are constructed in $O(m/\log^{1-\varepsilon} n)$ time. □

Wavelet trees. In Section 3 we showed how bit sequences $X(u)$ stored in the nodes of the wavelet tree of a sequence X can be obtained. Using Theorems 5, 6, and 7, we can augment $X(u)$ with secondary data structures that enable us to answer rank, access, and select queries on X.

Corollary 1. *Let X be a sequence of n positive integers such that $1 \le X[i] \le \sigma$ for $1 \le i \le n$. We can construct a binary balanced wavelet tree T for a sequence X, such that T uses $nH_0(X) + o(n\log\sigma)$ bits and answers queries* rank, select, *and* access *in $O(\log\sigma)$ time. The wavelet tree T can be constructed in $O(n\lceil \frac{\log\sigma}{\sqrt{\log n}} \rceil)$ time.*

Proof. We construct a balanced binary wavelet tree as in Theorem 1. Sequences $X(u)$ are stored in compressed form. It can be shown that the total space usage of all $X(u)$ is $n\log\sigma + o(n\log\sigma)$ bits; see e.g., [8], Section 3.1. □

We can further improve the construction time if a wavelet tree of special shape is used.

Corollary 2. *Let X be a sequence of n positive integers such that $1 \le X[i] \le \sigma$ for $1 \le i \le n$. We can construct a wavelet tree T for a sequence X, such that T uses $n(H_0(X) + 2) + o(n\log\sigma)$ bits and answers queries* rank, select, *and* access *in $O(\log\sigma)$ time. The wavelet tree T can be constructed in $O(n\lceil \frac{H_0(X)}{\sqrt{\log n}} \rceil)$ time.*

Proof. Barbay and Navarro [25] describe a wavelet tree T, such that the average leaf depth in T is $O(H_0(X))$ and the maximum leaf depth is $O(\log\sigma)$. Furthermore the total number of bits in all sequences $X(u)$ stored in nodes of T is bounded by $n(H_0(X) + 2)$. We construct T using Theorem 2. Data structures for sequences $X(u)$ are constructed as in Theorem 5. □

We also remark that we can construct wavelet trees for X with node degree $\log^{\varepsilon} n$ where ε is a small positive constant. In this case the space usage and construction times are the same as in Corollaries 1 and 2, but queries are supported in $O(\log\sigma/\log\log n)$ time.

5. Range maxima queries

In this section we show how our algorithms can be used to efficiently construct data structures for a certain type of geometric queries on a plane. Let P denote the set of two-dimensional points, such that all points have distinct x- and y-coordinates and all coordinates are integers in $[1, n]$. A point p is dominated by a point p' if $p'.x > p.x$ and $p'.y > p.y$. A two-dimensional range maxima query Q asks for all point p in a range Q, such that p is not dominated by any other point p' in Q. We will show below how wavelet trees can be used to answer range maxima queries in $O((k+1)\log n)$ time, where k is the number of reported points. We will need one additional type of queries. A range predecessor query Q asks for the topmost point p in a query rectangle $Q = [a, b] \times [c, d]$. Gagie et al. [26] showed that wavelet tree (with secondary structures that support rank and select queries in its nodes) can be used to answer range predecessor queries in $O(\log n)$ time. Hence we can construct a data structure that supports range-next value queries in $O(\log n)$ time and uses $n\log n + o(n\log n)$ bits of space.

It is known that we can employ a data structure for range predecessor queries to report all maximal points in a two-dimensional range. Consider a query range $Q = [a, b] \times [c, d]$. Let $p_1 = (x_1, y_1)$ denote the highest point in Q. Let $p_2 = (x_2, y_2)$ denote the highest point in $Q_1 = [x_1 + 1, b] \times [c, y_1]$. For $i \ge 2$, we recursively define $p_i = (x_i, y_i)$ as the highest point in $Q_{i-1} = [x_{i-1} + 1, b] \times [c, y_{i-1}]$. When p_{i-1} and Q_{i-1} are known, we can find p_i by answering a range predecessor query. Our procedure for finding all maximal points in Q finds points $p_1, \ldots, p_s$ until the range $Q_s = [x_s + 1, b] \times [c, y_s]$ is empty. It is easy to check that $p_1, \ldots, p_s$ contains all maximal points in Q and only those points. Since $x_i > x_j$ for all $1 \le j < i \le s$ and p_i is the topmost point in $[x_{i-1} + 1, b] \times [c, d]$, every p_i is not dominated by any other point in Q. Hence all $p_1, \ldots, p_s$ are maximal points in Q. Suppose that there is a point $p' = (x', y')$ such that $x_{i-1} < x' < x_i$ and p' is a maximal point. Then p' is the topmost point in $Q_i = [x_{i-1} + 1, b] \times [c, y_i]$. Hence p' is identical with p_i. Thus $p_1, \ldots, p_s$ is exactly the set of maximal points in Q.

Theorem 8. *There exists a data structure that uses $n \log n + o(n \log n)$ bits and answers two-dimensional orthogonal range maxima queries in $O((k+1)\log n)$ time, where k is the number of reported points. The data structure can be constructed in $O(n\sqrt{\log n})$ time.*

For comparison, it is possible to answer orthogonal range maxima queries in $O((k+1)\log^{\varepsilon} n)$ time using an $O(n \log n)$-bit data structure [27] or in $O(\log n / \log\log n + k)$ time using $O(n \log^{1+\varepsilon} n)$-bit data structure [28]. In both cases the preprocessing time is $O(n \log n)$. Thus our result enables us to reduce the space usage and construction time at a cost of slightly increasing the query time.

6. Conclusions

In this paper we described fast algorithms for constructing a wavelet tree. We showed that this important data structure can be constructed in $O(n\lceil \frac{\log \sigma}{\sqrt{\log n}} \rceil)$ time. If the wavelet tree with a special shape is used, then construction cost can be further reduced.

The problem of designing faster algorithms (e.g., algorithms that work in $O(n)$ or $O(n \log\log n)$ time for an alphabet $\{1, \ldots, n\}$) remains open.

References

[1] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2003, 2003, pp. 841–850.
[2] F. Claude, G. Navarro, Extended compact Web graph representations, in: Algorithms and Applications, Ukkonen Festschrift, Springer, 2010, pp. 77–91.
[3] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Trans. Algorithms 3 (2) (2007) article 20.
[4] N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, in: Proc. 18th CPM, 2007, pp. 205–215.
[5] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, J. ACM 57 (1) (2009).
[6] J. Barbay, M. He, I. Munro, S.S. Rao, Succinct indexes for strings, binary relations and multi-labeled trees, ACM Trans. Algorithms 7 (4) (2011) article 52.
[7] G. Navarro, Y. Nekrich, L.M.S. Russo, Space-efficient data-analysis queries on grids, Theoret. Comput. Sci. 482 (2013) 60–72.
[8] G. Navarro, Wavelet trees for all, J. Discrete Algorithms 25 (2014) 2–20.
[9] C. Makris, Wavelet trees: a survey, Comput. Sci. Inf. Syst. 9 (2) (2012) 585–625, http://dx.doi.org/10.2298/CSIS110606004M.
[10] S. Lee, K. Park, Dynamic rank-select structures with applications to run-length encoded texts, in: Proc. 18th CPM, in: Lecture Notes in Comput. Sci., vol. 4580, 2007, pp. 95–106.
[11] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM Trans. Algorithms 4 (3) (2008) article 32.
[12] R. González, G. Navarro, Improved dynamic rank-select entropy-bound structures, in: Proc. 8th LATIN, in: Lecture Notes in Comput. Sci., vol. 4957, 2008, pp. 374–386.
[13] S. Lee, K. Park, Dynamic rank/select structures with applications to run-length encoded texts, Theoret. Comput. Sci. 410 (43) (2009) 4402–4413.
[14] R. González, G. Navarro, Rank/select on dynamic compressed sequences and applications, Theoret. Comput. Sci. 410 (2009) 4414–4422.
[15] M. He, I. Munro, Succinct representations of dynamic strings, in: Proc. 17th SPIRE, 2010, pp. 334–346.
[16] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, ACM Trans. Algorithms 10 (3) (2014) 16:1–16:39, http://dx.doi.org/10.1145/2601073.
[17] F. Claude, P.K. Nicholson, D. Seco, Space efficient wavelet tree construction, in: Proc. 18th International Symposium on String Processing and Information Retrieval, SPIRE 2011, 2011, pp. 185–196.
[18] G. Tischler, On wavelet tree construction, in: Proc. 22nd Annual Symposium on Combinatorial Pattern Matching, CPM 2011, 2011, pp. 208–218.
[19] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, SIAM J. Comput. 17 (3) (1988) 427–462.
[20] T.M. Chan, K.G. Larsen, M. Patrascu, Orthogonal range searching on the ram, revisited, in: Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13–15, 2011, 2011, pp. 1–10.
[21] T.M. Chan, M. Patrascu, Counting inversions, offline orthogonal range counting, and related problems, in: Proc. 21st Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2010, 2010, pp. 161–173.
[22] M.A. Babenko, P. Gawrychowski, T. Kociumaka, T.A. Starikovskaya, Wavelet trees meet suffix trees, in: Proc. of the 26th Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2015, 2015, pp. 572–591.
[23] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th Annual Symposium on Foundations of Computer Science, FOCS 1989, 1989, pp. 549–554.
[24] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. 13th Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2002, 2002, pp. 233–242.
[25] J. Barbay, G. Navarro, Compressed representations of permutations, and applications, in: Proc. 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, 2009, pp. 111–122.
[26] T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on wavelet trees and applications to information retrieval, Theoret. Comput. Sci. 426 (2012) 25–41, http://dx.doi.org/10.1016/j.tcs.2011.12.002.
[27] Y. Nekrich, G. Navarro, Sorted range reporting, in: Proc. 13th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2012, 2012, pp. 271–282.
[28] G.S. Brodal, K.G. Larsen, Optimal planar orthogonal skyline counting queries, in: Proc. 14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014, 2014, pp. 110–121.