Chapter 8
Orthogonal Range Search Data Structures

Kazuki Ishiyama and Kunihiko Sadakane

Abstract We first review existing space-efficient data structures for the orthogonal range search problem. Then, we propose two improved data structures, the first of which has better query time complexity than the existing structures and the second of which has better space complexity that matches the information-theoretic lower bound.

8.1 Introduction

Consider a set P of n points in the d-dimensional space $\mathbb{R}^d$. Given an orthogonal range $Q = \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \left[l_1^{(Q)}, u_1^{(Q)}\right] \times \cdots \times \left[l_{d-1}^{(Q)}, u_{d-1}^{(Q)}\right]$, the problem of answering queries for information on $P \cap Q$, the subset of P contained in the range Q, is called the *orthogonal range search* problem, and is one of the fundamental problems in computational geometry.

The information obtained about $P \cap Q$ differs depending on the query. The most basic queries are the *reporting* query, which enumerates all the points in $P \cap Q$, and the *counting* query, which returns the number of points $|P \cap Q|$. There are other queries such as the *emptiness* query, which checks whether $P \cap Q$ is empty or not, and aggregate queries, which compute the summation, average, or variance of weights of points in the query range.

Applications of the orthogonal range search problem include database searches [21]. For example, assuming there is a database of employees of a company, then a query to count the number of employees whose duration of service is at least x_1 years and at most x_2 years, age is at least y_1 and at most y_2, and annual income is at least z_1 and at most z_2, can be formalized as an orthogonal range search problem. Other applications include geographical information systems, CAD, and computer graphics.

K. Ishiyama · K. Sadakane (✉)
Graduate School of Information Science and Technology, The University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
e-mail: sada@mist.i.u-tokyo.ac.jp

© The Author(s) 2022

N. Katoh et al. (eds.), *Sublinear Computation Paradigm*,
https://doi.org/10.1007/978-981-16-4095-7_8

In such applications, it is common to perform multiple queries on the same point set P. We therefore consider constructing the problem as an indexing problem: Given a point set P a priori, we first construct some data structure D from P. Then, when a query range Q is given, we answer the query using the data structure D.

8.1.1 Existing Work

In many existing works, the number n of points is regarded as a variable for evaluating time complexity and the number d of dimensions is regarded as a constant. However, in this chapter, we regard d as a variable too. For the computation model, we use w-bit word RAM where $w = \Theta(\lg n)$ bits. That is, a constant number of coordinate values can be treated in constant time. Then, it takes $O(d)$ time to check whether a point is inside a query range.

If more space than $\Theta(dn)$ words is allowed to be used for the space complexity of data structures and if we assume that d is a constant, then we can perform the counting and reporting queries in time polynomial to $\log n$. *Range trees* [2, 14, 15, 23] are such data structures. Range trees support counting queries in $O\left(d \log^{d-1} n\right)$ time and reporting queries in $O\left(d \log^{d-1} n + dk\right)$ time using $O\left(dn \log^{d-1} n\right)$ word space, where $k = |P \cap Q|$, that is, the number of points enumerated by a reporting query using the fractional cascading technique [15, 23]. Although these data structures are time-efficient, it is desirable to develop more space-efficient data structures.

Some data structures having linear space complexity have been proposed. For example, quad trees [6] were the first data structures used for orthogonal range search. Unfortunately, quad trees have terrible worst-case behaviors. To overcome this, kd-tree [1] is used. The query time complexity of the kd-tree is $O\left(d^2 n^{\frac{d-1}{d}}\right)$ for counting and $O\left(d^2 n^{\frac{d-1}{d}} + dk\right)$ for reporting [13].

These data structures store the coordinates of points separately in plain form, and therefore can be applied to the case of real-valued coordinates. However, if the coordinates take integer values from 0 to $n-1$, then there exist data structures with even smaller space complexity and query time complexity. For example, Chazelle [4] proposed a data structure for the two-dimensional case with linear space complexity and time complexity of $O(\lg n)$ for counting and $O(\lg n + k \lg^{\varepsilon} n)$ for reporting where $0 < \varepsilon < 1$ is any constant. Note that although the assumption that each coordinate value is an integer from 0 to $n-1$ seems too strict, as is explained in Sect. 8.2.2, any orthogonal range search problem in d-dimensional space can be reduced into one on the $[n]^d$ grid, and therefore the assumption does not create any difficulties.

There has also been research on succinct data structures for the orthogonal range search problem. The wavelet tree [9] is a data structure which was originally proposed for representing compressed suffix arrays, and it later turned out that wavelet tree can support various queries efficiently [18]. For the orthogonal range search problem, wavelet tree can support counting queries in $O(\lg n)$ time and reporting

queries in $O((1+k)\lg n)$ time [8]. Bose et al. [3] proposed improved succinct data structures that support counting queries in $O(\lg n/\lg\lg n)$ time and reporting queries in $O(((1+k)\lg n/\lg\lg n)$ for two-dimensional cases.

For higher dimensions, Okajima and Maruyama [20] proposed the KDW-tree, which is a succinct data structure for any dimensionality. The query time complexity of the KDW-tree is smaller than that of the kd-tree. If we assume d is a constant, counting queries take $O\left(n^{\frac{d-2}{d}}\lg n\right)$ time and reporting queries take $O\left(\left(n^{\frac{d-2}{d}}+k\right)\lg n\right)$ time. The KDW-tree has been shown to be practical by numerical experiments.

8.1.2 Our Results

We show space and time complexities of data structures for the orthogonal range search problem explained in Sect. 8.1.1 and our proposed data structures in Table 8.1. Note that these are for the case where the coordinates are integers from 0 to $n-1$, and the space complexities are measured in bits. Table 8.1 shows reporting time complexities. Counting time complexities can be obtained by letting $k=0$.

Our data structures are space-efficient for high-dimensional orthogonal range search problems.

Our first data structure has the same space complexity as the KDW-tree and better query time complexities. Note that the result in Table 8.1 is for the case of $d \geq 3$. If $d=2$, we can improve the $n^{\frac{d-2}{d}}$ term to $\lg n$. This result appeared in [11].

Note that, as shown in Sect. 8.2.1, the necessary space to represent a set of n points in d-dimensional space such that each coordinate takes an integer value from 0 to $n-1$ is $(d-1)n\lg n+\Theta(n)$ bits. This means that if we assume d is a constant, the space complexity of the KDW-tree and our first data structure does not match the information-theoretic lower bound asymptotically.

Table 8.1 Comparison of complexities. The results or KDW-tree and Ours 1 are for $d \geq 3$. Note that k is the number of points enumerated by a reporting query. The time complexities for counting queries are obtained by letting $k=0$ in the time complexities for reporting queries

Data structure	Dim.	Space (bits)	Query time
kd-tree [1]	d	$O(dn\lg n)$	$O\left(d^2 n^{\frac{d-1}{d}}+dk\right)$
Wavelet tree [9]	2	$n\lg n+o(n\lg n)$	$O((1+k)\lg n)$
Bose et al. [3]	2	$n\lg n+o(n\lg n)$	$O\left((1+k)\frac{\lg n}{\lg\lg n}\right)$
KDW-tree [20]	d	$d\{n\lg n+ o(n\lg n)\}$	$O\left(\left(\mathrm{poly}(d)\cdot n^{\frac{d-2}{d}}+dk\right)\lg n\right)$
Ours 1	d	$d\{n\lg n+ o(n\lg n)\}$	$O\left(\left(d^3 n^{\frac{d-2}{d}}+dk\right)\frac{\lg n}{\lg\lg n}\right)$
Ours 2	d	$(d-1)\{n\lg n+ o(n\lg n)\}$	$O(dn\lg n)$

Our second data structure uses $(d-1)n \lg n + (d-1) \cdot o(n \lg n)$ bits of space. This asymptotically matches the information-theoretic lower bound even if d is assumed to be a constant. Therefore, we can say this data structure is truly succinct. Unfortunately, the worst-case query time complexity is $O(dn \lg n)$, which is not fast in theory. However, this data structure is fast in practice for the case where the number d of dimensions is large but the number d' of dimensions used for a query is small. This kind of query often occurs in the database search applications shown in Sect. 8.1. This result appeared in [10].

8.2 Preliminaries

In this paper, we assume that coordinates of points are non-negative integers. As will be explained in Sect. 8.2.2, we sometimes assume that coordinates are integers from 0 to $n-1$. Therefore, we define [n] as the set $\{0, 1, \ldots, n-1\}$. For a d-dimensional space, we denote each dimension by dim. 0, dim. $1, \ldots,$ dim. $d-1$, coordinate values of a point by 0-th coordinate value, 1-th coordinate value, $\ldots, d-1$-th coordinate value. For a rooted tree, we assume the depth of the root node is 0. Throughout the paper, log x denotes the natural logarithm and lg x denotes the base 2 logarithm.

Next, we define two concepts used in this chapter. The first one is containment degree. This is the concept of an inclusion relationship between two orthogonal ranges introduced in [20]. For two d-dimensional orthogonal ranges $Q = \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \ldots \times \left[l_{d-1}^{(Q)}, u_{d-1}^{(Q)}\right]$ and $R = \left[l_0^{(R)}, u_0^{(R)}\right] \times \cdots \times \left[l_{d-1}^{(R)}, u_{d-1}^{(R)}\right]$, we define $\mathrm{CDeg}(R, Q)$ as

$$\mathrm{CDeg}(R, Q) = \# \left\{ i \in [d] \;\middle|\; \left[l_i^{(R)}, u_i^{(R)}\right] \subseteq \left[l_i^{(Q)}, u_i^{(Q)}\right] \right\}$$

and call it the containment degree of R with respect to Q. This is the number of dimensions, in each of which R is contained in Q. The containment degree is an important concept for analyzing time complexities of orthogonal range search algorithms.

Next, we explain z-value. This is a projection of multi-dimensional data onto one-dimensional data as proposed by Morton [17]. Consider a point $p = (p_0, p_1, \ldots, p_{d-1})$ in the d-dimensional space where the coordinate values are integers. If coordinate values are expressed as l-bit binary numbers $p_0 = b_0^0 b_0^1 \cdots b_0^{l-1}$, $p_1 = b_1^0 b_1^1 \cdots b_1^{l-1}, \ldots, p_{d-1} = b_{d-1}^0 b_{d-1}^1 \cdots b_{d-1}^{l-1}$, the z-value $z(p)$ of point p is defined as

$$z(p) = b_0^0 b_1^0 \cdots b_{d-1}^0 b_0^1 b_1^1 \cdots b_{d-1}^1 \cdots b_0^{l-1} b_1^{l-1} \cdots b_{d-1}^{l-1}.$$

In the case of a two-dimensional space, if we arrange grid points in increasing order of z-value, we see a z-shape curve as shown in Fig. 8.1. We therefore call the value z-value.

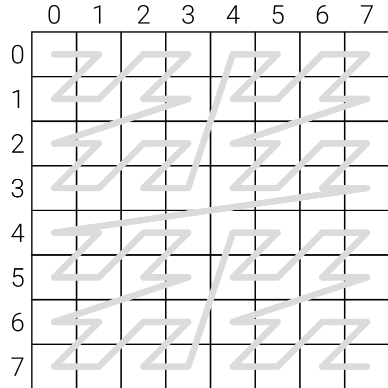

Fig. 8.1 Curve obtained by joining grid points in z-valueorder in two-dimensional space

8.2.1 Succinct Data Structures and Information-Theoretic Lower Bound

Succinctness of data structures was proposed by Jacobson [12] and is one of the criteria for measuring space complexities of data structures. It is defined as follows.

Let n be the number of different values that an object can take. Then, we need at least $\lceil \lg n \rceil$ bits of space to represent the object. If the space complexity $S(n)$ of a data structure representing the object satisfies $S(n) = \lg n + \mathrm{o}(\lg n)$ bits, we say the data structure is *succinct* and $\lceil \lg n \rceil$ bits is the *information-theoretic lower bound* of the size of representations of the object. Note that succinct data structures not only offer data compression, but also support some efficient queries. For orthogonal range search, a naive algorithm supports linear time queries by scanning an array containing coordinate values of points. Succinct data structures are therefore expected to answer queries in sublinear time.

The space complexity of $\lg n + \mathrm{o}(\lg n)$ bits in the definition of succinct data structures indicates that the size of auxiliary indexing data structures added to the data is negligibly small compared with the size of the data itself ($\lg n$ bits). In other words, the space complexity of succinct data structures asymptotically matches the information-theoretic lower bound when $n \to \infty$.

We compute the information-theoretic lower bound for representing a set of points with integer coordinates. Assume that i-th coordinate value takes integer values from 0 to $U_i - 1$. Because the number of grid points is $\prod_{i=0}^{d-1} U_i$, the number of different sets of n points is

$$\binom{\prod_{i=0}^{d-1} U_i}{n}.$$

By using Stirling's approximation formula

$$\log n! = n \log n - n + \mathrm{O}(\log n),$$

we obtain

$$\begin{aligned}
\log \binom{U}{n} &= \log U! - \log(U-n)! - \log n! \\
&= U \log U - U - (U-n)\log(U-n) + (U-n) - n\log n + n + \mathrm{O}(\log U) \\
&= U \log \frac{U}{U-n} + n \log \frac{U-n}{n} + \mathrm{O}(\log U) \\
&= U \log \left(1 + \frac{n}{U-n}\right) + n \log \frac{U}{n}\left(1 - \frac{n}{U}\right) + \mathrm{O}(\log U) \\
&= U \left(\frac{n}{U-n} - \Theta\left(\left(\frac{n}{U-n}\right)^2\right)\right) + n \log \frac{U}{n} - \Theta\left(\frac{n^2}{U}\right) + \mathrm{O}(\log U) \\
&= n \log U - n \log n + \Theta(n).
\end{aligned}$$

Therefore, the information-theoretic lower bound of the size for representing the point set is

$$\lg \binom{\prod_{i=0}^{d-1} U_i}{n} = \sum_{i=0}^{d-1} n \lg U_i - n \lg n + \Theta(n).$$

Note that storing coordinate values of the points explicitly using $\sum_{i=0}^{d-1} \lceil \lg U_i \rceil$ use $n \lg n$ bit more space than the information-theoretic lower bound.

8.2.2 Assumptions on Point Sets

Because data structures such as kd-tree or range trees that have linear or larger space complexities usually store the coordinates of points in a plain format, we do not care whether they are integers or real values. However, if we consider succinct data structures, we usually assume that coordinates values are integers from 0 to $n-1$. We also assume that for any points $p, q \in P$ and any $i \in [d]$, the i-th coordinate value p_i of p and the i-th coordinate value q_i of q are different. Although this assumption may appear to be unrealistic and too strong, for the orthogonal range search problem, it is known that an arbitrary point set on $\mathbb{R}^d$ can be transformed into a point set on $[n]^d$ [7].

Consider a set P of n points on $\mathbb{R}^d$. We create another point set P' on $[n]^d$ as follows. The set P' also contains n points and there is a one-to-one correspondence between points in P and points in P'. Assume that $p \in P$ corresponds to $p' \in P'$. Then, the i-th coordinate value p'_i of p' is then defined from the i-th coordinate value p_i of p as

$$p'_i = \#\{q \in P \mid q_i < p_i\}. \tag{8.1}$$

That is, the i-th coordinate value of p' is the number of points in P such that the i-th coordinate value is smaller than p_i. This is called the rank value of p with respect to the i-th coordinate value, and the transformation is called the transformation into rank space. We use arrays $C_0, C_1, \ldots, C_{d-1}$ each of length n. The array C_i stores the i-th coordinate values of points in P in increasing order.

By using the point set P' on the rank space and the arrays C_i $(i = 0, \ldots, d-1)$ that contain the original coordinate values of the points in P, we can reduce the problem of orthogonal range search on the original point set P into that on P'. Assume that a query range $Q = \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \cdots \times \left[l_{d-1}^{(Q)}, u_{d-1}^{(Q)}\right] \subset \mathbb{R}^d$ is given for a point set P. From the construction of P', there exists a range $Q' = \left[l_0^{(Q')}, u_0^{(Q')}\right] \times \cdots \times \left[l_{d-1}^{(Q')}, u_{d-1}^{(Q')}\right] \subset [n]^d$ such that

$$p \in Q \iff p' \in Q'.$$

The boundaries of this Q' are computed by

$$l_i^{(Q')} = \#\left\{p \in P \;\middle|\; p_i < l_i^{(Q)}\right\}$$
$$u_i^{(Q')} = \#\left\{p \in P \;\middle|\; p_i \le u_i^{(Q)}\right\} - 1.$$

These are computed in $O(d \lg n)$ time by binary searches on the arrays C_i. Then, the counting query is performed by using Q'. For the reporting query, after finding a point $p' \in P'$ which is included in the query range Q' in the rank space, we need to recover the original coordinates of the point $p \in P$. This is done in $O(d)$ time using the arrays C_i containing the coordinates of the original points by

$$p_i = C_i[p'_i].$$

Thus, an orthogonal range search problem on $\mathbb{R}^d$ can be transformed into that on $[n]^d$. Note that if coordinates are transformed as in Eq. (8.1), the identical coordinate values in $\mathbb{R}^d$ are transformed into identical coordinate values in $[n]^d$. By shifting values by one for the identical coordinate values, we can transform the coordinate values so that for any two distinct points $p', q' \in P'$ and any $i \in [d]$, the i-th coordinate value p'_i of p' is different from the i-th coordinate value q'_i of q'.

If the original points have integer coordinate values, we can reduce the space [19]. Consider the case where P is a point set on $[U]^d$, that is, each coordinate value takes an integer value from 0 to $U - 1$. In this case, the point set P' in the rank space does not change. However, we store the coordinates of the original point set P in a different way. We store them using multi-sets $M_0, M_1, \ldots, M_{d-1}$, each of which corresponds to one of the d dimensions. The multi-set M_i stores the i-th coordinate value of the points in P. We use the data structure of [22] to store multi-sets.

Lemma 8.1 *There exists a data structure using* $n \lg(U/n) + O(n)$ *which supports a selectm query on a multi-set* M_i *in constant time.*

A selectm query on a multi-set M finds the j-th smallest element in M. That is, $C_i[j]$ is obtained by finding the j-th smallest element in array C_i. Therefore, if a query range Q on $[U]^d$ is given, it can be transformed into a query range Q' on the rank space by binary searches using selectm queries, and the original coordinate values are obtained by d many selectm queries.

Assume that there exists a succinct data structure D' for a point set P' on $[n]^d$. Then, the space complexity of D' is $(d-1)n \lg n + (d-1) \cdot o(n \lg n)$ bits, as shown in Sect. 8.2.1. If we add d data structures of Lemma 8.1, the total space complexity becomes $dn \lg U - n \lg n + (d-1) \cdot o(n \lg n)$ bits. This is succinct for the point set P on $[U]^d$. Therefore, if there exists a succinct data structure for a point set on $[n]^d$, we can construct a succinct data structure for a point set on $[U]^d$. From here onward, we consider only point sets on $[n]^d$.

8.3 kd-Tree

kd-tree [1] is a well-known data structure that partitions the space recursively. It is used not only for the orthogonal range search problem, but also for the nearest neighbor search problem.

8.3.1 Construction of kd-Trees

We explain the algorithm for constructing a kd-tree of a point set P for the two-dimensional case. First, we find the point p for which the x-coordinate is the median of the point set P, and store p at the root of the kd-tree. Next, we divide the set $P \setminus \{p\}$ into two: the set P_{left} that stores points with x-coordinates smaller than that of p, and the set P_{right} that stores points with x coordinates larger than that of p. We add two children $v_{\text{left}}, v_{\text{right}}$ to the root of the kd-tree. Next, from P_{left} (P_{right}), we find p_{left} (p_{right}) for which the y-coordinate is the median of the set, and we store p_{left} (p_{right}) in v_{left} (v_{right}). Similarly, we divide the set $P_{\text{left}} \setminus \{p_{\text{left}}\}$ ($P_{\text{right}} \setminus \{p_{\text{right}}\}$) into two subsets according to y-coordinates, find medians with respect to x-coordinates, and store them in children of v_{left} (v_{right}), and repeat this recursively. Figure 8.2 shows an example of partitioning a point set.

For a d-dimensional space, we partition the space based on the first dimension, the second dimension, and so on. After partitioning the space based on the d-th dimension, we use the first dimension again.

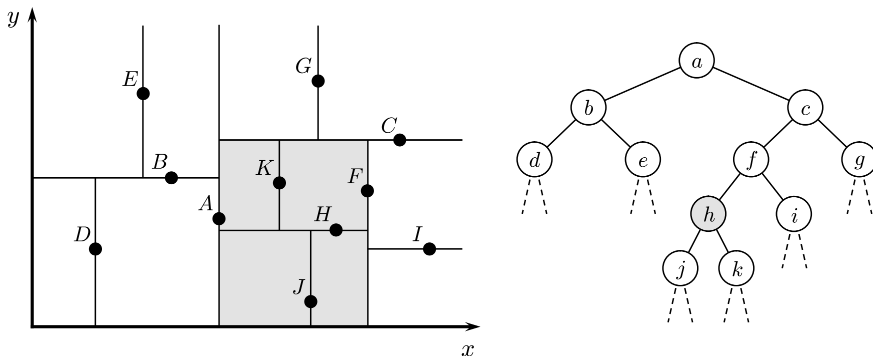

Fig. 8.2 Partitioning of a space based on the point set (left) and the corresponding kd-tree (right). Points $A, B, C, \ldots$ correspond to nodes $a, b, c, \ldots$. The range corresponding to node h is shown in gray in the left figure

8.3.2 Range Search Algorithm

An important concept for understanding range searches using a kd-tree is the correspondence between nodes of the kd-tree and ranges. In Sect. 8.3.1, we explained that each node of the kd-tree stores a point. We can also consider that each node corresponds to an orthogonal range. Let $V(v)$ denote the point in P stored in node v and $R(v)$ denote the corresponding range. Then $R(v)$ is defined as follows:

– For the root node r of the kd-tree, the range $R(r)$ is the whole space.
– For a node v at depth l, the range $R(v_{\text{left}})$ for the left child v_{left} of v is obtained as follows. We partition $R(v)$ into two by the hyperplane that is perpendicular to the (l mod d)-th axis and contains $V(v)$. Then, $R(v_{\text{left}})$ is the range with the smaller (l mod d)-th coordinate value and $R(v_{\text{right}})$ is the range with the larger (l mod d)-th coordinate value.

For example, in Fig. 8.2, the range $R(h)$ corresponding to node h is the gray area.

The algorithm for reporting queries using a kd-tree is as follows. The algorithm searches the space by traversing tree nodes from the root. Each time a node v is visited, the algorithm checks whether the corresponding point $V(v)$ ($\in P$) is contained in the query range Q or not. If the range $R(v)$ is fully contained in the query range Q, the algorithm outputs all the points stored in the sub-tree rooted at v. If $R(v)$ and Q has no intersection, the algorithm terminates the search of the sub-tree. For a counting query, instead of outputting all the points when $R(v)$ is contained in Q, the algorithm finds and accumulates the size of the sub-tree rooted at v. Although it may seem impossible to execute the algorithm since the range $R(v)$ for node v is not explicitly stored in the kd-tree, if the range $R(v)$ for node v is known, then we know the coordinate values of the hyperplane partitioning the range from the coordinate values of point $V(v)$, and we can compute $R(v_{\text{left}})$ and $R(v_{\text{right}})$. Therefore, we can execute the algorithm by keeping the range $R(v)$ during the search.

8.3.3 Complexity Analyses

The time complexity of kd-trees is analyzed in [13]. A counting query takes $O\left(d \cdot n^{\frac{d-1}{d}} + 2^d\right)$ time. In general, we assume d is a constant and write the complexity as $O\left(n^{\frac{d-1}{d}}\right)$. For a reporting query, we output all coordinates of points in Q. Because a point can be output in constant time, the query time complexity is $O\left(n^{\frac{d-1}{d}} + k\right)$.

If $d \geq \lg n$, the height of the kd-tree is at most d, and therefore the space is partitioned at most d times. Then, it is necessary to traverse all the nodes and a query takes $O(n)$ time.

8.4 Wavelet Tree

Wavelet tree is a succinct data structure supporting various queries on strings and integer sequences efficiently. It was originally proposed for representing compressed suffix arrays [9], but it later became known that wavelet tree can support more operations [18]. Orthogonal range search in two-dimensional space is one of these operations [16].

8.4.1 Construction

The two-dimensional point sets P that can be represented directly using wavelet tree are those where the coordinates take integer values from 1 to n and the x-coordinate values are all distinct. As explained in Sect. 8.2.2, without loss of generality, we can transform any point set into a point set in $[n]^d$ space. For such a two-dimensional point set P, consider an integer sequence C that contains the y-coordinates of the points in increasing order of x-coordinates. For example, for the point set in Fig. 8.3, the corresponding integer sequence C is 4, 2, 7, 5, 0, 3, 1, 6. For this sequence C, we construct a wavelet tree as follows.

First, we consider that the root of the wavelet tree corresponds to C. Note that we do not store C directly in the wavelet tree. We then focus on the most significant (highest) bit of the $\lceil \lg n \rceil$-bit binary representation of each integer in C. If it is 0 (1), the integer is moved into the left (right) child of the root. We consider that each child node of the root corresponds to an integer sequence containing the numbers in the original array C in the same order. For example, in the example in Fig. 8.3, integers from 0 to 3 go to the left child, and integers from 4 to 7 go to the right child. Therefore, the left child corresponds to an integer sequence 2, 0, 3, 1, and the right child 4, 7, 5, 6.

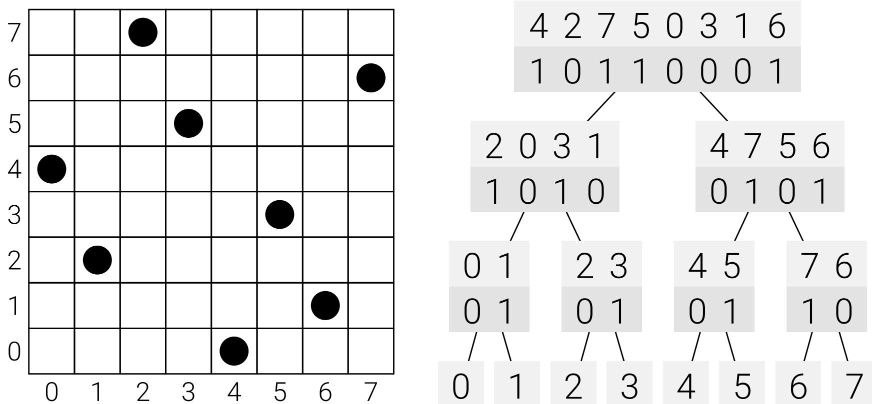

Fig. 8.3 A two-dimensional point set P (left) and the corresponding wavelet tree (right)

Next, for each integer sequence of child nodes, we focus on the second most significant bit of the binary representation of each number. We move a number with 0 bit to the left, and a number with 1 bit to the right. Similarly, we repeat this until the integer sequence of a node consists of the identical integer.

Note that we do not store integer sequences in nodes of the wavelet tree. In each node, we store a bit string of the same length as the corresponding integer sequence. The i-th bit of the bit string is 0 (1) if the i-th integer in the integer sequence goes to the left (right) child. In other words, a bit string stored in a node of depth l is the concatenation of the $(l+1)$-th highest bit of each integer in the integer sequence corresponding to the node. In the example in Fig. 8.3, the integer sequence corresponding to the root node is 4, 2, 7, 5, 0, 3, 1, 6, and because integers from 0 to 3 go to the left child and integers from 4 to 7 go to the right child, the bit string stored in the root node is 1, 0, 1, 1, 0, 0, 0, 1. Note that we do not store bit strings at leaf nodes. We show the information stored in the wavelet tree in the right tree in Fig. 8.3. Only bit strings drawn above the dark gray rectangles, that is, those in the lower row of each node, are stored.

Note that although it may seem impossible to recover the original information (the integer sequence) from these bit strings, it is possible. Consider the recovery of the fourth integer of the wavelet tree in Fig. 8.3 (right). From the bit string stored in the root node, we know that the first bit of the integer is 1. Because this 1 bit corresponding to the fourth integer is the third 1 in the bit string, we know that the integer to be recovered corresponds to the third bit of the bit string in the right child of the root node. If we look at the third bit of the right child, we know that the second bit of the integer is 0. Further, this 0 bit is the second 0 in the bit string, the integer to be recovered corresponds to the second bit of the left child of the current node. Finally, from the second bit of the left child, we know the last bit of the integer to be recovered is 1. Therefore, the fourth integer is 101 in binary, that is, 5. This is shown in Fig. 8.4.

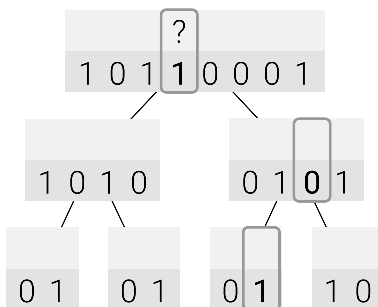

Fig. 8.4 In the wavelet tree in Fig. 8.3, we recover the fourth integer. By looking at the bits enclosed in boxes, we know that the fourth integer is 101 in binary, that is, 5

In this recovery operation, we need to compute the number of zeros/ones in the first i bits of a bit string. This operation is also used in the range search algorithm in the next section. If we look at bits one by one from the beginning of a bit string, it takes $O(n)$ time, which is too slow. We therefore represent the bit string of each node by the following data structure [5, 12].

Lemma 8.2 *For a bit string of length n, there exists a data structure using* $n + o(n)$ *bits which answers a rank/select query in constant time, where the rank query* $\mathrm{rank}_b\,(B, i)$ *is to count the number of b bits (*$b = 0, 1$*) in the bits from* $B[0]$ *to* $B[i]$ *(*$i \geq 0$*) of a bit string B, and the select query* $\mathrm{select}_b\,(B, i)$ *is to return the position of the i-th b (*$i \geq 1$, $b = 0, 1$*) in a bit string B.*

The select query is also necessary for range searches using a wavelet tree.

8.4.2 Range Search Algorithm

We explain how to solve the two-dimensional range search problem using a wavelet tree. First, we explain the counting query, which is performed by a recursive function as in Algorithm 1. For a query range $Q = [l, r] \times [b, t]$, the argument of the function is $\text{WTC}\textsc{ounting}(l, r, b, t, v_{\text{root}}, 0, 2^{\lceil \lg n \rceil} - 1)$, where v_{root} is the root node of the wavelet tree. The left (right) child of node v is represented by v_{left} (v_{right}). The bit string stored in node v is represented by $v.B$.

We explain the algorithm in Fig. 8.5 using the example of searching a range $Q = [1, 6] \times [1, 4]$ for the point set P in Fig. 8.3.

The search algorithm traverses the tree from the root. During the search, the algorithm keeps the interval I of an integer sequence (or bit string) corresponding to an interval of the x-coordinate of the query range. In the example in Fig. 8.5, we focus on the interval $I = [1, 6]$ at the root node. To move to the left child, we need to compute the interval corresponding to the query range. This is done by a rank query that counts the number of zeros from the beginning of the bit string to a specified position. In the bit string stored in the root node, the number of zeros from the beginning to the 0-th position (in general, if the interval is $I = [l, r]$, to $(l - 1)$-th

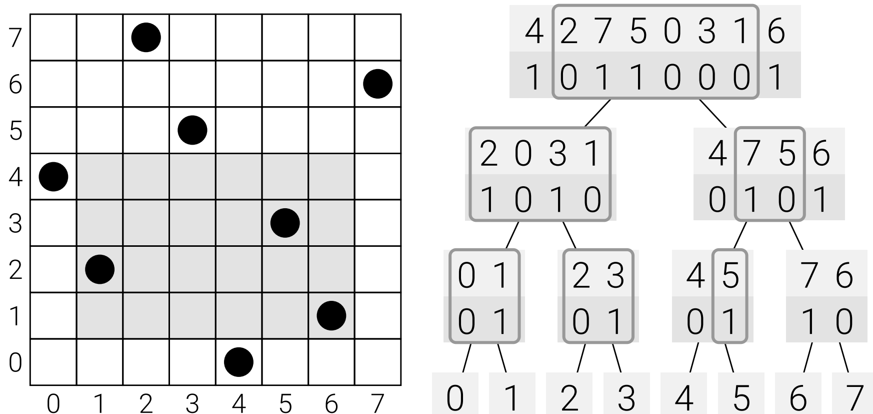

Fig. 8.5 Behavior of the algorithm when searching a range of $[1, 6] \times [1, 4]$ for the two-dimensional point set in Fig. 8.3

Algorithm 1 WTCOUNTING($x_1, x_2, y_1, y_2, v, a, b$)

Input: A node v of the wavelet tree and an interval $[x_1, x_2]$ in the corresponding bit string, the interval $[a, b]$ of y coordinate corresponding to node v, and the interval $[y_1, y_2]$ of y coordinate for the query range.
Output: The number of points stored in the sub-tree rooted at v and contained in Q.
1: **if** $x_1 > x_2$ **then**
2:    **return** 0
3: **else if** $[a, b] \cap [y_1, y_2] = \varnothing$ **then**
4:    **return** 0
5: **else if** $[a, b] \subseteq [y_1, y_2]$ **then**
6:    **return** $x_2 - x_1 + 1$
7: **end if**
8: $x_1^l \leftarrow \mathrm{rank}_0\,(v.B, x_1 - 1)$
9: $x_2^l \leftarrow \mathrm{rank}_0\,(v.B, x_2) - 1$
10: $x_1^r \leftarrow x_1 - x_1^l$
11: $x_2^r \leftarrow x_2 - x_2^l - 1$
12: $m \leftarrow \lfloor (a + b)/2 \rfloor$
13: **return** WTCOUNTING($x_1^l, x_2^l, y_1, y_2, v_{\mathrm{left}}, a, m$)
        +WTCOUNTING($x_1^r, x_2^r, y_1, y_2, v_{\mathrm{right}}, m + 1, b$)

---

position) is 0, so we know the interval corresponding to the query starts at position 0. Because the number of zeros from the beginning to the 6-th position (in general, if the interval is $I = [l, r]$, to r-th position) is four, we know the interval ends at position 3. Thus, we obtain the interval $I = [0, 3]$ for the left child. Similarly, for the right child, by using rank queries counting the number of ones, we can obtain the interval $I = [1, 2]$.

We repeat this process by going down the tree maintaining an interval. When we reach a leaf, we can determine if the y-coordinate of the point is included in the query range. However, we can sometimes determine this at an earlier stage. For example,

in Fig. 8.5, after obtaining the interval $I = [1, 2]$ at the left child of the root, for the right child of the current node the interval of the y-coordinate corresponding to the node is [2, 3], which is completely included in the interval [1, 4] of the y-coordinate of the query range. Therefore, for the two points we focus on at this node, both the x- and y-coordinates are included in the query range, and we found two points in the query range. However, after computing the interval $I = [1, 2]$ for the right child of the root, the interval of the y-coordinate corresponding to the right child of the current node is [6, 7], which has no intersection with the interval [1, 4] of the y-coordinate of the query range. We do not need to further search the sub-tree.

As observed above, in a range search using a wavelet tree, if the query range is $Q = [l, r] \times [b, t]$, we first focus on points for which the x-coordinates are contained in Q, that is, contained in the range $[l, r] \times [0, n-1]$. Next, the process of traversing down the tree corresponds to partitioning the range into two according to the y-coordinate. If an obtained range is completely contained in the query range, or does not intersect with the query range, we terminate searching the sub-tree.

For counting queries, it is sufficient to sum the number of points. For reporting queries, the extra work of computing the coordinates of the points is also required. This is shown in Algorithm 2.

The outline of the reporting query is the same as the counting query. In Algorithm 1, we obtain the number of points in Line 2. We change it one by one to output coordinates of points corresponding to the interval $[x_1, x_2]$ of the bit string $v.B$. The x- and y-coordinates of each point are obtained by WTREPORTX

Algorithm 2 WTREPORTING$(x_1, x_2, y_1, y_2, v, a, b)$

Input: A node v of the wavelet tree, the interval $[x_1, x_2]$ of the bit string stored in it, the interval $[a, b]$ of y coordinates corresponding to the range for v, and the interval $[y_1, y_2]$ of y coordinates for the query range.
Output: Coordinates of point stored in the sub-tree rooted at v and contained in Q.

```
1: if x_1 > x_2 then
2:    terminate
3: else if [a, b] ∩ [y_1, y_2] = ∅ then
4:    terminate
5: else if [a, b] ⊆ [y_1, y_2] then
6:    for i = x_1 to x_2 do
7:       x ← WTREPORTX(v, i)
8:       y ← WTREPORTY(v, i, a, b)
9:       Output (x, y)
10:   end for
11: end if
12: x_1^l ← rank_0(v.B, x_1 − 1)
13: x_2^l ← rank_0(v.B, x_2) − 1
14: x_1^r ← x_1 − x_1^l
15: x_2^r ← x_2 − x_2^l − 1
16: m ← ⌊(a + b)/2⌋
17: WTREPORTING(x_1^l, x_2^l, y_1, y_2, v_left, a, m)
18: WTREPORTING(x_1^r, x_2^r, y_1, y_2, v_right, m + 1, b)
```

Algorithm 3 WTREPORTX(v, i)

Input: A node v of the wavelet tree and an integer i.
Output: The x coordinate value of the point corresponding to the i-th bit of the bit string stored in v.

```
1: if v is the root then
2:    return i
3: else if v is the left child of v_parent then
4:    i ← select_0 (v_parent.B, i + 1)
5:    return WTREPORTX(v_parent, i)
6: else
7:    i ← select_1 (v_parent.B, i + 1)
8:    return WTREPORTX(v_parent, i)
9: end if
```

Algorithm 4 WTREPORTY(v, i, a, b)

Input: A node v of the wavelet tree, the interval $[a, b]$ of y coordinate corresponding to the range for v, and an integer i.
Output: The y coordinate value of the point corresponding to the i-th bit of the bit string stored in v.

```
1: if a = b then
2:    return a
3: else if v.B[i] = 0 then
4:    i ← rank_0 (v.B, i) − 1
5:    return WTREPORTY(v_left, i, a, ⌊(a + b)/2⌋)
6: else
7:    i ← rank_1 (v.B, i) − 1
8:    return WTREPORTY(v_right, i, ⌊(a + b)/2⌋ + 1, b)
9: end if
```

and WTREPORTY, respectively. The algorithm WTREPORTY for computing the y-coordinate (Algorithm 4) is similar to the algorithm for recovering a value of the original integer array explained in Sect. 8.4.1. We compute the y-coordinates by traversing down the tree using rank queries.

In contrast, the algorithm WTREPORTX for computing the x-coordinate (Algorithm 3) traverses up the tree using select queries. We explain this by example. In Fig. 8.5, assume that at node v, which is the right child of the left child of the root, we find that points corresponding to the interval $I = [0, 1]$ are contained in the query range. Consider the computation of the x-coordinate of the point corresponding to the bit $v.B[1]$. First, the node v we focus on is the right child of its parent. We find the position of the second 1 in the parent by a select query. Then we know that the point corresponds to the bit $v'.B[2]$ in the parent node v'. Next, because the current node is the left child of the parent (the root), we find the position of the third 0 in the bit string of the parent by a select query. Now we know that the point corresponds to the bit $r.B[5]$ at the root node r. That is, the x-coordinate of the point is 5.

As shown above, we can traverse the nodes of the wavelet tree using rank and select queries on bit strings. For range searches, we traverse down the tree from the root computing the intervals of the x-coordinate corresponding to the query range.

If we find a node where the corresponding interval of the y-coordinate is contained in the query range, we answer the query by computing the length of the interval or coordinate values by traversing the tree.

8.4.3 Complexity Analyses

We now analyze the space complexity of the wavelet tree and query time complexities for the orthogonal range search problem.

First, we analyze the space complexity. The height of the wavelet tree is $\lceil \lg n \rceil$. The total length of bit strings stored in the nodes with the same depth is always n. Therefore, the total length of all the bit strings in the wavelet tree is $n \lg n$. We can concatenate all the bit strings and store only a long bit string. Then it is not necessary to store the tree structure of the wavelet tree. By using the data structure of Lemma 8.2 for this long bit string, the space complexity is $n \lg n + o(n \lg n)$ bits in total.

Next, we consider query time complexities. For a counting query, we consider the number of visited nodes. In the wavelet tree, each time we traverse an edge toward a leaf, points with small y-coordinates go to the left child, and points with large y-coordinates go to the right child. At leaves we can consider that all points are sorted in increasing order of y-coordinates. This means that leaf nodes corresponding to the interval of y-coordinates of the query range exist in a consecutive place in the wavelet tree. Now, consider the set M of nodes of the wavelet tree defined as follows. The set M contains a maximal node v such that the y-coordinates corresponding to the leaf nodes in the sub-tree rooted at v are contained in the query range, that is, the y-coordinates of the leaves in the sub-tree of v are contained in the query range but the sub-tree of the parent of v contains some node for which the corresponding y-coordinate is not contained in the query range. This is the set of nodes from which we do not further search the sub-tree for a counting query using the wavelet tree, and in Fig. 8.6, it is shown as dark gray nodes.

Let A be the set of nodes that are ancestors of nodes of M. This is the set of nodes visited before reaching nodes of M which are shown as light gray nodes in Fig. 8.6. The number of nodes visited in a counting query is then $|A| + |M|$. We now consider the size of M and A.

For the size of the set M, the following lemma holds.

Lemma 8.3 *It holds* $|M| = O(\lg n)$.

Proof (Lemma 8.3) The set M is constructed as follows. Let M' be the set of leaf nodes of the wavelet tree corresponding to the interval of y-coordinates in the query range. For the nodes of M', if two nodes v_1 and v_2 have a common parent node v, we remove v_1 and v_2 from M' and add v to M'. By repeating this process until there are no such pairs of nodes, the set M' coincides with M.

For each depth of the wavelet tree, the number of nodes of depth belonging to M is then at most two, because if there exist more than two nodes, two of them must have the same parent. This completes the proof that $|M| = O(\lg n)$.

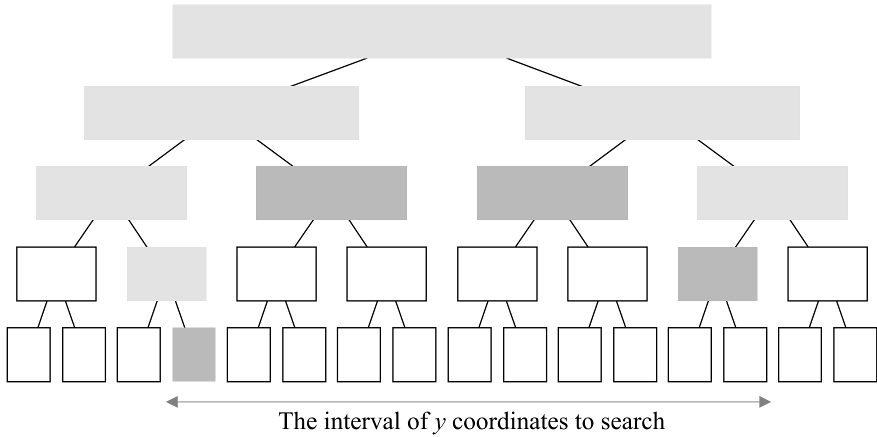

Fig. 8.6 Nodes visited by a counting query. We traverse light gray nodes, and when we reach a dark gray node, we do not further search the nodes below it

For the size of the set A, the following lemma holds.

Lemma 8.4 *It holds* $|A| = \mathrm{O}(\lg n)$.

Proof (Lemma 8.4) Consider a node v in the set A. In the set of leaf nodes in the sub-tree rooted at v, there must exist a leaf node where the corresponding y-coordinate is included in the query range and a leaf node where the corresponding y-coordinate is not included in the query range. Therefore, for each depth of the wavelet tree, there are at most two such nodes in A, because if there exists more than two such nodes, for a node in the middle, the corresponding y-coordinates of the leaves in the sub-tree rooted at that node are contained in the query range. This completes the proof that $|A| = \mathrm{O}(\lg n)$.

From the above discussion, the number of nodes visited in a counting query is $|A| + |M| = \mathrm{O}(\lg n)$. When we visit a new node, we use a constant number of rank queries. Because a rank query takes constant time (Lemma 8.2), the time complexity of a counting query using the wavelet tree is $\mathrm{O}(\lg n)$.

For a reporting query, it is necessary to compute coordinates of points in the query range. As explained in Sect. 8.4.2, x-coordinates are computed by traversing up the tree and y-coordinates are computed by traversing down the tree, with the coordinates of each point computed by visiting $\mathrm{O}(\lg n)$ nodes. Moving to an adjacent node in the wavelet tree is done by a constant number of rank/select queries, and each rank/select query takes constant time (Lemma 8.2). Therefore, the coordinates of a point are obtained in $\mathrm{O}(\lg n)$ time, and the time complexity for a reporting query using the wavelet tree is $\mathrm{O}((1 + k) \lg n)$, where k is the number of output points.

We obtain the following theorem.

Theorem 8.1 *The space complexity of the wavelet tree representing a two-dimensional point set on* $[n]^2$ *is* $n \lg n + o(n \lg n)$ *bits, and a counting query takes* $O(\lg n)$ *time, and a reporting query takes* $O((k+1) \lg n)$ *time, where* k *is the number of points to enumerate.*

As shown in Sect. 8.2.1, the information-theoretic lower bound for a point set on $[n]^2$ is $n \lg n + O(n)$ bits. Therefore, the wavelet tree is a succinct data structure.

Theorem 8.2 *Let* P *be a set of points on* $M = [1..n] \times [1..n]$ *in which all points have distinct* x*-coordinates. Then, there exists a data structure using* $n \lg n + o(n \lg n)$ *bits that answers a counting query in* $O(\lg \lg n)$ *time and a reporting query in* $O((1+k) \lg n / \lg \lg n)$ *time, where* k *is the number of points to output.*

8.5 Proposed Data Structure 1: Improved Query Time Complexity

This data structure uses the idea of adding data structures to the kd-tree to improve the query time complexity [20]. First, we explain the idea of [20] in Sect. 8.5.1. Next, we explain the algorithm of range search in Sect. 8.5.3, and analyze the time complexity in Sect. 8.5.4.

8.5.1 Idea for Improving the Time Complexity of the kd-Tree

The method proposed in [20] improves the query time complexity of the kd-tree by adding d many wavelet trees to the kd-tree such that the term $n^{(d-1)/d}$ is replaced by $n^{(d-2)/d}$ ($\lg n$ if $d = 2$), at the cost of increasing the total complexity by a factor of $O(\lg n)$. Note that we assume point sets are on $[n]^d$.

First, we construct the kd-tree for a given set P of points in the d-dimensional space. Next, we label the nodes of the kd-tree with numbers based on the inorder traversal of a binary tree defined as follows:

- If the root node has a left child, we traverse the sub-tree rooted at the node.
- Examine the root node.
- If the root node has a right child, we traverse the sub-tree rooted at the node.

Figure 8.7 shows an example of a point set (left) and numbers assigned based on the inorder traversal of the kd-tree of the set (right).

Next, we make point sets P_i ($i = 0, \ldots, d-1$) with n points on $[n]^2$. The two-dimensional point set P_i is created as follows. If a point p in the original d-dimensional point set P has the i-th coordinate value p_i and the inorder position of the node of the kd-tree containing p is j, we add point (j, p_i) to P_i. Figure 8.8 shows the point sets P_0, P_1 created from the point set in Fig. 8.7.

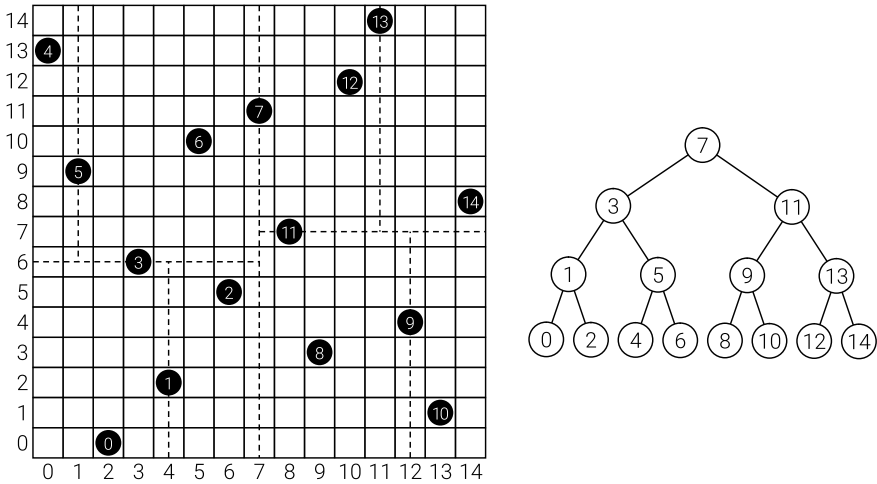

Fig. 8.7 A two-dimensional point set (left) and the corresponding kd-tree (right). The numbers of nodes are assigned by an inorder traversal of the kd-tree. The dashed lines in the left figure show the partition of the space by the kd-tree

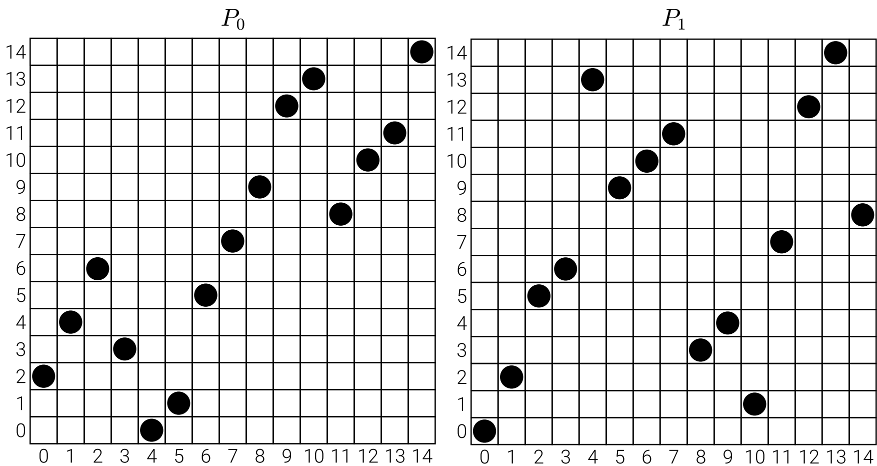

Fig. 8.8 Two-dimensional point sets obtained from the point set in Fig. 8.7

From these two-dimensional point sets $P_0, \ldots, P_{d-1}$, we construct wavelet trees $W_0, \ldots, W_{d-1}$. The wavelet trees W_i can be thought of as constructed from an integer sequence A_i containing the i-th coordinate value of points in P in the order of the kd-tree.

These data structures can be used for range searches as follows. Given a query range Q, we perform the original search using the kd-tree. In the original algorithm, as explained in Sect. 8.3, we traverse the kd-tree and shrink the range $R(v)$, and when

$\mathrm{CDeg}(R(v), Q) = d$ (i.e., $R(v) \subseteq Q$), we know that all the points in the sub-tree rooted at v are contained in Q. By using the d wavelet tree, we can terminate the search when $\mathrm{CDeg}(R(v), Q) = d - 1$. Assume that when a node v is visited, $R(v)$ is contained in Q for all dimensions except for i. The inorder numbers of nodes in the sub-tree rooted at node v have consecutive values. Let $[a, b]$ be the interval for the numbers. Then, the points in this interval are contained in Q except for dim. i. This implies that points in P_i that are contained in the range $[a, b] \times [l_i^{(Q)}, u_i^{(Q)}]$ are contained in Q even for dim. i. Therefore, after finding the node v, it is sufficient to search the range $[a, b] \times [l_i^{(Q)}, u_i^{(Q)}]$ of P_i using wavelet trees W_i.

The number of nodes of the kd-tree visited by this method is $\mathrm{O}\big(n^{(d-2)/d}\big)$ ($\mathrm{O}(\lg n)$ for the case $d = 2$). The search of the last dimension using the wavelet tree takes $\mathrm{O}(\lg n)$ time for a counting query. Therefore, the time complexity for a counting query using the kd-tree is improved to $\mathrm{O}\big(n^{(d-2)/d} \lg n\big)$ ($\mathrm{O}\big(\lg^2 n\big)$ for the case $d = 2$).

8.5.2 Index Construction

We now explain the proposed data structure. First, we construct the kd-tree for a given point set P. Note that this kd-tree is temporarily built in order to construct our data structure, and is not included in the final structure. Next, as in Sect. 8.5.1, we number the nodes of the kd-tree by an inorder traversal, and create d many two-dimensional point sets $P_0, \ldots, P_{d-1}$. For each P_i, we create the data structure of [3]. Let B_i be this data structure. Finally, we discard the kd-tree. The final data structure consists of $B_0, \ldots, B_{d-1}$.

8.5.3 Range Search Algorithm

We explain the algorithm for a reporting query using the data structure explained in the previous section. The pseudocode is shown in Algorithm 5. This algorithm simulates a search of the kd-tree using $B_0, \ldots, B_{d-1}$. We explain it in comparison with the search algorithm of the kd-tree. Note that we explain the algorithm assuming the inorder number of each node v of the kd-tree is also assigned to the point $V(v)$ stored in v. That is, if we say a point with number j, it is the point stored in the node with inorder position j. We also assume that for an interval $[a, b]$ of point numbers, $R([a, b])$ denotes the range containing points that have numbers in $[a, b]$. In Algorithm 5, the interval $[a, b]$ of point numbers always corresponds to the interval of inorder numbers of nodes in the sub-tree rooted at a node v of the kd-tree. Therefore, $R([a, b])$ coincides with $R(v)$.

If we use the kd-tree, we shrink the focused range $R(v)$ by going down the tree. In the proposed method, by shrinking the interval $[a, b]$ of point numbers, we reduce the corresponding range $R([a, b])$. Because the kd-tree stores the point $V(v)$ corresponding to a node v, we can obtain the information of the point used for

Algorithm 5 REPORTING($[a, b]$, Q)

Input: An interval of point numbers $[a, b]$ and a query range Q.
Output: Points with numbers in $[a, b]$ and which are contained in range Q.
1: **if** $Deg(R([a, b]), Q) = d - 1$ **then**
2: For the last dimension i such that $R([a, b])$ is not yet contained in Q, search $[a, b] \times \left[l_i^{(Q)}, u_i^{(Q)}\right]$ of P_i using B_i and enumerate points contained in Q. For each point, compute coordinates using $B_0, \ldots, B_{d-1}$.
3: **else if** $R([a, b])$ has no intersection with Q **then**
4: terminate
5: **else if** $a = b$ **then**
6: Examine the point with number a. If it is in Q, output it.
7: **else**
8: $c \leftarrow \lceil (a + b)/2 \rceil$
9: Output the point with number c if it is in Q.
10: REPORTING($[a, c - 1]$, Q)
11: REPORTING($[c + 1, b]$, Q)
12: **end if**

partitioning the space. In contrast, in the proposed method, points are not explicitly stored. However, if the focused interval $[a, b]$ coincides with the interval of inorder numbers for the sub-tree rooted at a node v, we find $c = \lceil (a + b)/2 \rceil$ is the number of the points used for partitioning.[1] Furthermore, the intervals $[a, c - 1]$ and $[c + 1, b]$ correspond to the intervals of the numbers for sub-trees rooted at the left and right child of v, respectively. Therefore, by a recursive search of Algorithm 5, we can obtain the correct partitioning points.

For the range $R([a, b])$, we can compute the ranges after a partition from the range before partition and the coordinates of the point used for partitioning similarly to the case of the kd-tree.

8.5.4 Complexity Analyses

We now analyze the complexities of the algorithm. First, we consider its space complexity. We use d data structures of Bose et al. [3] each of which uses $n \lg n + o(n \lg n)$ bits as in Theorem 8.2. The total space complexity is then $dn \lg n + o(dn \lg n)$ bits.

Next, we consider the query time complexity. If we use the same analysis as in [20], assuming d is a constant, we can show the number of nodes corresponding to cells with containment degree of at most $d - 1$ is $O\left(n^{\frac{d-2}{d}}\right)$. Here, we derive the query time complexity using a novel analysis for non-constant d.

[1] In the kd-tree, at each depth, we partition the space by the median of the point set with respect to a dimension, and therefore $c = \lceil (a + b)/2 \rceil$ is the number of the point used for partitioning. If the point set contains an even number of points, we can obtain the correct partitioning point using a predetermined rule.

The proposed method partitions the space for each dimension in order, in the same fashion as for the kd-tree. As in [20], we define a series of partitions with respect to dim. 0 to dim. d-1 as a cycle. We then calculate the number $T_m(n, d)$ of nodes at which the containment degree with respect to Q is at most $d-2$ in the m-th cycle. When the $(m-1)$-th cycle has finished, the space is partitioned into $2^{d(m-1)}$ many cells. Among them, we count the number of cells for which the containment degree with respect to Q is at most $d-2$. These cells contain a $(d-2)$-dimensional face of Q (an edge of a cuboid if $d=3$). A $(d-2)$-dimensional face of a d-dimensional orthogonal range Q is obtained by choosing two dimensions from the d-dimensions and choosing the upper side or the lower side of the range for each of the two dimensions. Therefore, Q has $\binom{d}{2}2^2$ many $(d-2)$-dimensional faces. When the $(m-1)$-th cycle has finished, because each dimension is partitioned into 2^{m-1} cells, the number of cells containing a $(d-2)$-dimensional face is at most $2^{(m-1)(d-2)}$. Then after the $(m-1)$-th cycle, the number of cells to be searched is at most

$$\binom{d}{2}2^2 \cdot 2^{(m-1)(d-2)}.$$

In the sub-trees rooted at these nodes, the number of nodes in the m-th cycle is $2^d - 1$. Therefore, it holds that

$$\begin{aligned} T_m(n, d) &\le \left(2^d - 1\right)\binom{d}{2}2^2 \cdot 2^{(m-1)(d-2)} \\ &< 2^3 d(d-1)2^{(d-2)m}. \end{aligned}$$

Let $N(n, d)$ be the number of nodes for which the containment degree with respect to Q is at most $d-2$. It then holds that

$$\begin{aligned} N(n, d) &= \sum_{m=1}^{\frac{1}{d}\lg n} T_m(n, d) \\ &< 2^3 d(d-1) \sum_{m=1}^{\frac{1}{d}\lg n} 2^{(d-2)m} \\ &= 2^3 d(d-1) \frac{2^{d-2}\left(2^{\frac{d-2}{d}\lg n} - 1\right)}{2^{d-2} - 1} \\ &= \mathrm{O}\left(d^2 \cdot n^{\frac{d-2}{d}}\right). \end{aligned}$$

We use the fact that the containment degree is weakly increasing as we traverse down the tree. In the proposed method, we terminate the search when the containment degree reaches $d-1$. The visited nodes are then those with containment degree of at most $d-2$ and their child nodes. There are at most $2N(n, d)$ such nodes. The

proposed method virtually traverses the kd-tree. It takes $\mathrm{O}\left(d \frac{\lg n}{\lg \lg n}\right)$ time to compute the coordinates of a point stored in a node. When a node for which the containment degree with respect to Q is $d-1$, we search the last one dimension in $\mathrm{O}\left(\frac{\lg n}{\lg \lg n}\right)$ time. The time complexity of a counting query is then $\mathrm{O}\left(d^3 n^{\frac{d-2}{d}} \frac{\lg n}{\lg \lg n}\right)$. For a reporting query, it takes $\mathrm{O}\left(d \frac{\lg n}{\lg \lg n}\right)$ time to compute the coordinates of a point. The total time complexity is then $\mathrm{O}\left(\left(d^3 n^{\frac{d-2}{d}} + dk\right) \frac{\lg n}{\lg \lg n}\right)$, where k is the number of points in Q. In summary, we obtain the following:

Theorem 8.3 *For an orthogonal range search problem on the* $[n]^d$ *space, there exists a data structure that has space complexity of* $dn \lg n + \mathrm{o}(dn \lg n)$ *bits and which answers a counting query in* $\mathrm{O}\left(d^3 n^{\frac{d-2}{d}} \frac{\lg n}{\lg \lg n}\right)$ *time and a reporting query in* $\mathrm{O}\left(\left(d^3 n^{\frac{d-2}{d}} + dk\right) \frac{\lg n}{\lg \lg n}\right)$*, where k is the number of points in the query range.*

8.6 Proposed Data Structure 2: Succinct and Practically Fast

The second proposed method is a data structure that is succinct and practically fast. In this method, we use $d-1$ many wavelet trees to represent a point set on $[n]^d$. In Sect. 8.6.1, we explain how to construct the data structure. In Sect. 8.6.2, we explain the algorithm for the orthogonal range search problem. In Sect. 8.6.3, we analyze the space and time complexities.

8.6.1 Index Construction

In this method, we assume that the points of P have distinct values in the 0-th coordinate value.

First, we create length-n integer arrays $A_1, \ldots, A_{d-1}$. The array A_i corresponds to dim. i, and stores the i-th coordinate value of the points in increasing order of the 0-th coordinate value. Next, for those arrays we create wavelet trees $W_1, \ldots, W_{d-1}$. The wavelet trees W_i can be considered to represent the two-dimensional point set P_i generated from the d-dimensional point set P by projecting the points onto the plane spanned by the 0-th axis and the i-th axis. Figure 8.9 shows an example.

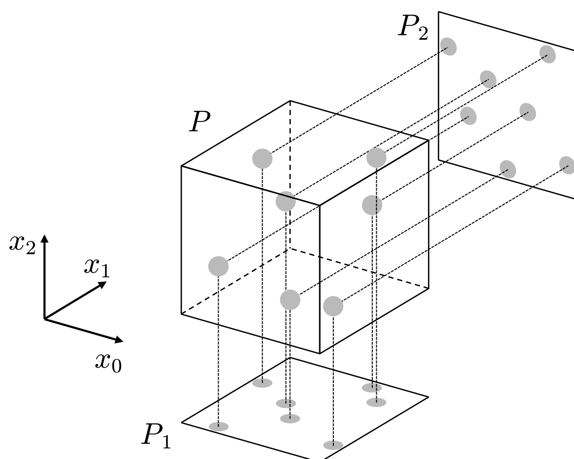

Fig. 8.9 A three-dimensional point set P and two-dimensional point sets P_1, P_2 generated by projecting P onto each plane

Algorithm 6 REPORT(Q)

Input: A query range $Q = \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \cdots \times \left[l_{d-1}^{(Q)}, u_{d-1}^{(Q)}\right]$.
Output: The coordinates of points of P contained in Q.
1: $D := \varnothing$
2: **for** $i = 1$ to $d - 1$ **do**
3: **if** $\left[l_i^{(Q)}, u_i^{(Q)}\right] \subsetneq [0, n - 1]$ **then**
4: $D = D \cup \{i\}$
5: $c_i := \text{COUNT}\left(P_i, \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \left[l_i^{(Q)}, u_i^{(Q)}\right]\right)$
6: **end if**
7: **end for**
8: Sort elements $i_1, \ldots, i_{|D|}$ of D in increasing order of c_i.
9: $A := \text{REPORTX}\left(P_{i_1}, \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \left[l_{i_1}^{(Q)}, u_{i_1}^{(Q)}\right]\right)$
10: **for** $i = i_2$ to $i_{|D|}$ **do**
11: **for all** $a \in A$ **do**
12: **if** The i-th coordinate of a point for which the 0-th coordinate is a and is not contained in $\left[l_i^{(Q)}, u_i^{(Q)}\right]$ **then**
13: $A = A \setminus \{a\}$
14: **end if**
15: **end for**
16: **end for**
17: **for all** $a \in A$ **do**
18: Obtain the coordinates of a point for which the 0-th coordinate is a and output them.
19: **end for**

8.6.2 Range Search Algorithm

Next, we explain how to solve the orthogonal range search problem using the data structure (Algorithm 6).

Assume that a query range $Q = \left[l_0^{(Q)}, u_0^{(Q)}\right] \times \cdots \times \left[l_{d-1}^{(Q)}, u_{d-1}^{(Q)}\right]$ is given. For each $i = 1, \ldots, d-1$ such that $\left[l_i^{(Q)}, u_i^{(Q)}\right] \neq [0, n-1]$, that is, the dimension i used for the search, we count the number of points of P_i that are contained in range $\left[l_0^{(Q)}, u_0^{(Q)}\right] \times \left[l_i^{(Q)}, u_i^{(Q)}\right]$ using wavelet trees W_i (counting query). Let m $(= |D|)$ be the number of i $(= 1, \ldots, d-1)$ such that $\left[l_i^{(Q)}, u_i^{(Q)}\right] \neq [0, n-1]$, and let $i_1, \ldots, i_m$ be the sorted ones in increasing order of the number of answers of counting queries.

Using wavelet trees W_{i_1}, we then enumerate only the x-coordinates of points of P_{i_1} contained in $\left[l_0^{(Q)}, u_0^{(Q)}\right] \times \left[l_{i_1}^{(Q)}, u_{i_1}^{(Q)}\right]$ and store them in a set A. For each element a of A and for each $i = i_2, \ldots, i_m$, we check whether the i-th coordinate of a point for which the 0-th coordinate is a is contained in the query range. The elements remaining in A correspond to points in the query range. The answer to a counting query is the cardinality of A. For a reporting query, we compute coordinates of the points and output them.

The reason we compute the number of points contained in each dimension by a counting query is twofold. Firstly, the x-coordinate (the 0-th coordinate) of points contained in the query range with respect to the i_1-th (and the 0-th) dimension can be output quickly at line 9 of the algorithm if the number of points to enumerate is small. Secondly, in the double loops from line 10 to line 16, we want to reduce the size of A as soon as possible.

8.6.3 Complexity Analyses

Consider the space and time complexities of the proposed method.

For the space complexity, we use $d-1$ many wavelet trees. Therefore, the space complexity is $(d-1)\lg n + (d-1) \cdot o(\lg n)$ bits.

For the query time complexity, let m be the number of wavelet trees used in a search. The time to perform m counting queries on wavelet tree is $O(m \lg n)$. We then sort m integers in $O(m \lg m)$ time. Next, we enumerate the x-coordinates of points contained in the query range for the dimension with the minimum number of points. Let $c_{i_1} = c_{\min}$ be the number of points to enumerate. This takes $O((1 + c_{\min}) \lg n)$ time. The time to check whether these points are contained in the query range for other dimensions is $O((m-1)c_{\min} \lg n)$. Let d' be the number of dimensions used in the query, then it holds that $m \leq d'$. Therefore, the query time complexity can be written as $O\big(d' c_{\min} \lg n + d' \lg d'\big)$.

8.7 Conclusion

In this chapter, we first reviewed data structures for high-dimensional orthogonal range search. We then proposed two data structures for the problem.

The first one simulates the search of the kd-tree using d succinct data structures for two-dimensional orthogonal range search data structures [3]. We improved the query time complexity of KDW-tree while keeping the same space complexity.

The second one is succinct and practically fast. The space complexity is $(d-1)n \lg n + (d-1) \cdot \mathrm{o}(n \lg n)$, which is succinct. The worst-case query time complexity is $\mathrm{O}(dn \lg n)$, which is not good. However, if the number d of dimensions is large but the number d' of dimensions used in a search is small, it runs fast in practice.

References

1. J.L. Bentley, Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
2. J.L. Bentley, Decomposable searching problems. Inf. Process. Lett. **8**(5), 244–251 (1979)
3. P. Bose, M. He, A. Maheshwari, P. Morin, Succinct orthogonal range search structures on a grid with applications to text indexing, in *Proceedings of the 11th Workshop on Algorithms and Data Structures (WADS 2009)* (Springer, 2009), pp. 98–109
4. B. Chazelle, A functional approach to data structures and its use in multidimensional searching. SIAM J. Comput. **17**(3), 427–462 (1988)
5. D. Clark, *Compact Pat Trees*. Ph.D. thesis (University of Waterloo, 1997)
6. R.A. Finkel, J.L. Bentley, Quad trees a data structure for retrieval on composite keys. Acta Inform. **4**(1), 1–9 (1974)
7. H.N. Gabow, J.L. Bentley, R.E. Tarjan, Scaling and related techniques for geometry problems, in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC 1984)* (ACM, 1984), pp. 135–143
8. T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. **426–427**, 25–41 (2012)
9. R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)* (SIAM, 2003), pp. 841–850
10. K. Ishiyama, K. Sadakane, Practical space-efficient data structures for high-dimensional orthogonal range searching, in *Proceedings of the 10th International Conference on Similarity Search and Applications (SISAP 2017)* (Springer, 2017), pp. 234–246
11. K. Ishiyama, K. Sadakane, A succinct data structure for multidimensional orthogonal range searching, in *Proceedings of Data Compression Conference 2017 (DCC 2017)* (2017), pp. 270–279
12. G.J. Jacobson, *Succinct Static Data Structures*. Ph.D. thesis (Pittsburgh, PA, USA, 1988)
13. D.T. Lee, C.K. Wong, Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta Inform. **9**(1), 23–29 (1977)
14. D.T. Lee, C.K. Wong, Quintary trees: a file structure for multidimensional datbase sytems. ACM Trans. Database Syst. **5**(3), 339–353 (1980)
15. G.S. Lueker, A data structure for orthogonal range queries, in *Proceedings of the 9th Annual Symposium on Foundations of Computer Science (SFCS 1978)* (IEEE, 1978), pp. 28–34
16. V. Mäkinen, G. Navarro, Position-restricted substring searching, in *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN 2006)* (Springer, 2006), pp. 703–714

17. G.M. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing* (International Business Machines Company, New York, 1966)
18. G. Navarro, Wavelet trees for all. J. Discrete Algorithms **25**, 2–20 (2014)
19. G. Navarro, *Compact Data Structures: A Practical Approach* (Cambridge University Press, 2016)
20. Y. Okajima, K. Maruyama, Faster linear-space orthogonal range searching in arbitrary dimensions, in *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX 2015)* (SIAM, 2015), pp. 82–93
21. J. O'Rourke, J.E. Goodman, *Handbook of Discrete and Computational Geometry* (CRC Press, 2004)
22. R. Raman, V. Raman, S.R. Satti, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), Article 43 (2007)
23. D.E. Willard, *Predicate-Oriented Database Search Algorithms*. Technical report (Harvard Univ Cambridge MA Aiken Computation Lab, 1978)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.