Range Searching in Categorical Data: Colored Range Searching on Grid [⋆]

Pankaj K. Agarwal[1], Sathish Govindarajan[1], and S. Muthukrishnan[2]

[1] Department of Computer Science, Duke University, Durham, NC 27708
{pankaj, gsat}@cs.duke.edu
[2] AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932.
muthu@research.att.com

Abstract. Range searching, a fundamental problem in numerous applications areas, has been widely studied in computational geometry and spatial databases. Given a set of geometric objects, a typical range query asks for reporting all the objects that intersect a query object. However in many applications, including databases and network routing, input objects are partitioned into categories and a query asks for reporting the set of categories of objects that intersect a query object. Moreover in many such applications, objects lie on a grid. We abstract the category of an object by associating a color with each object. In this paper, we present efficient data structures for solving the colored range-searching and colored point-enclosure problem on $U \times U$ grid. Our data structures use near-linear space and answer a query in $O(\log \log U + k)$ time, where k is the output size. As far as we know, this is the first result on colored range-searching for objects lying on a grid.

1 Introduction

We are given a set of geometric objects – points, lines, polygons – to preprocess. Given a query object, the *range searching* problem is to return the intersection of the query with the given set of objects. In the past few decades, range searching has been extensively studied. See [1, 16] for recent surveys. The fascination with range searching is because it has myriad applications in areas of database retrieval, computer aided design/manufacturing, graphics, geographic information systems, etc. Specifically, range querying is exceedingly common in database systems (Who are the students with GPAs greater than 3.8? Name the customers whose age is in $[20 - 40]$ and whose income is in greater than $100k$?). Every commercial database system in the market has optimized data structures for solving various range searching problems. Thus, range searching data structures remain one of the few data structures that actually get used in practice at commercial scale.

In this paper, we study an extension to the basic range searching problem. Our motivation arose from database applications where we observed that range searching

⋆ Research by the first two authors is supported by NSF under grants CCR-00-86013 EIA-98-70724, EIA-01-31905, and CCR-97-32787, and by a grant from the U.S.–Israel Binational Science Foundation. Part of this work was done when the second author was visiting the third author at DIMACS.

amid objects are common, but what was highly prevalent was range searching in which the objects have categories and the query often calls for determining the (distinct) list of categories on the objects that intersected the query object. Here are two examples:

(i) A canonical example is an analyst who is interested in "What are the sectors (telecom, biotech, automobile, oil, energy, ..) that had $5 - 10\%$ increase in their stock value?". Here each stock has a category that is the industry sector it belongs to, and we consider a range of percentage increase in the stock value. We are required to report all the distinct sectors that have had one or more of their stocks in desired range of growth, and not the specific stocks themselves. This is one-dimensional range searching in categorical data.[1]

(ii) As another example, consider a database of IP packets sent through an Internet router over time. IP packets have addresses, and organizations get assigned a contiguous range of IP addresses that are called subnets (eg., all hosts within Rutgers Univ, all hosts within `uunet`, are distinct examples of subnets, and all IP addressed within a subnet share a long common prefix). An analyst of the IP traffic may ask "Between time 10 AM and 11 AM today, which are the subnets to which traffic went from `uunet` through the given router?". Here the `uunet` specifies a range of IP addresses and time specifies an orthogonal range, and each IP packet that falls within the cross-product of these two ranges falls into a category based on the subnet of the destination of the packet. This is an example of a two-dimensional range searching in categorical data.

Thus range searching (in all dimensions) in categorical data is highly prevalent in database queries. Range searching in categorical data also arises in document retrieval problems [15] and in indexing multidimensional strings [11].

The range-searching problem in categorical data, as they arise in database applications, have the following common characteristics. First, the points (and endpoints of objects) are on a grid (eg., IP addresses are 32 bit integers and time, stock prices etc. are typically rounded). Second, while it is important to optimize the preprocessing time and space, query time is very critical. In data analysis applications, response time to a query must be highly optimized. Third, often the number of categories is very large (for example, in the router traffic example above, the number of subnets is in thousands). Finally, the dataset may be clustered along one of the dimensions or in a category, and hence, an efficient algorithm cannot afford to retrieve all points in the given range and search through them to find the distinct categories they belong to; the output set may be far smaller than the set of all points in the range. In other words, the naive algorithm of doing a classical range query first and followed by selecting distinct categories form the answer set will be inefficient.

In this paper, we study two very basic problems in range searching in categorical data, keeping the above characteristics in mind. The points (and end points of rectangles) come from an integer grid. We study the range-searching and point-enclosure problems. We present highly efficient algorithms for both problems, in particular, the query time is $O(\log \log U)$ where U is the grid size.

[1] Flip Korn of AT&T Research clarified that in classical database language, this may be thought of as GROUP BY based on the category of the stock following a range SELECT. Therefore, this is of fundamental interest in databases.

1.1 Problems

In abstraction, the problems of our interest are as follows. The notion of the category associated with an object is abstracted as the *color* associated with it.

1. *Colored range searching.* We are given P, a set of n colored points in $[0, U]^2$, for preprocessing. Given a query rectangle q whose endpoints lie on the grid $[0, U]^2$, the problem is to output the set of *distinct* colors of points contained in q.
2. *Colored point enclosure problem.* We are given P, a set of n colored rectangles whose endpoints lie on the grid $[0, U]^2$, for preprocessing. Given a query point $q = [q_1, q_2]$, where $q_1, q_2 \in [0, U]$, the problem is to output the set of *distinct* colors of rectangles that contain q.

The problems above are stated in two dimensions, and for most part of the paper, we focus on the two dimensional case. However, our approach gives best-known bounds for higher dimensional cases as well, and they are detailed in Section 4.

Typically, the data structure results on the grid are interpreted under the assumption that U is polynomial in n. In our problem, U is indeed polynomial in n since the input is static and we can renumber the points and corners into an integer set of size $1 \ldots O(n)$; any query point will first get mapped into the $O(n)$ range using a predecessor query (this is a standard trick, taking time $O(\log\log U)$) and then the query processing begins. As a result $U = O(n)$, provided we are willing to have $O(\log\log U)$ additive term in the preprocessing of the query. We refer to this technique as *input mapping* in the rest of the discussion.

1.2 Our Results

We present data structures for both problems that are highly optimized for query processing.

Colored range searching. We construct an $O(n \log^2 U)$ sized data structure that answers colored range searching queries in $O(\log\log U + k)$ time, where k is the output size.

Previously known result for this problem did not use the grid properties and produced a data structure that uses $O(n \log^2 n)$ size and answers a query in $O(\log n + k)$ time [14, 13]. Previous algorithms that work on the grid only studied the classical range searching problem and produced a $O(n \log^\epsilon n)$ size data structure with $O(\log\log U + k)$ query time [3]; the query time of such a data structure for colored range searching can be arbitrarily bad.

Colored point enclosure. For any integer parameter ℓ, we can construct a $O(n\ell^2 \log_\ell^2 U)$-size data structure that can answer a colored point enclosure queries in $O(k \log_\ell U)$ time, where k is the output size.

We can obtain a space-time tradeoff for our data structure by choosing appropriate values for ℓ. For example, setting $\ell = U^\epsilon$ and using the input mapping, we get a data structure of $O(n^{1+\epsilon})$-size that can answer colored point enclosure queries in

$O(k + \log\log U)$ time where k is the output size. Most of the previous results make an additional assumption that $U = O(n^c)$ for some constant c. If we make this assumption, the query time reduces to $O(k)$. Note that the $\log\log U$ term in the query is due to input mapping, which is not required if $U = O(n^c)$. This result, surprisingly, matches the best known bound for what appears to be a simpler problem of point enclosure problem on the grid without considering the colors, i.e., the standard point enclosure problem [10]. Other previous results for the colored point enclosure problem do not assume an underlying grid and produce a data structure that uses $O(n \log n)$ space and answers queries in $O(\log n + k)$ time [13].

1.3 Technical Overview of Our Results

We use contrasting techniques to obtain the two results.

For the range searching problem, best-known results typically use recursive divide and conquer on one dimension after another [4–6]. We start with a one dimensional result. There are at least three known solutions [14, 13, 15] for the colored range searching problem in one dimension. We use their intuition but present yet another solution which, in contrast to the others, we are able to make dynamic as well as persistent. Then we use a sweep-line approach to obtain our result in $2D$ (rather than the recursive divide and conquer).

For the point-enclosure problem, the previously known approach [10] relies on building a one dimensional data structure using exponential trees and then extending it to two dimensions by dynamizing and making it persistent. We build the desired one dimensional data structure efficiently for colored point enclosure (this may be of independent interest), but then generalize it to two dimensions directly to obtain two-dimensional exponential trees. This gives us our bound for the colored point enclosure problem.

In Section 2, we present our results for colored range searching in two dimensions. In Section 3, we present our results for colored point enclosure in two dimensions. In Section 4, we present extensions to higher dimensions.

2 Colored Range Searching in 2D

In this section, we describe a data structure to solve the colored range-searching problem on $U \times U$ grid. In Section 2.1, we present a structure to answer the 1-dimensional colored range query. In Section 2.2 we extend the data structure to answer two dimensional colored range queries. The main idea of the extension is as follows. We make the 1-dimensional structure partially persistent and use the sweep-line paradigm to answer three-sided queries. We then extend this structure to answer four-sided range queries.

2.1 Colored Range Searching in $1D$

Let P be a set of n colored points in $[0, U]$. Let C denote the set of distinct colors in the point set P. We first solve the colored range-searching problem for the special case of semi-infinite query q i.e, $q = [q_1, \infty]$. For each color $c \in C$, we pick the point $p_c \in P$ with color c having the maximum value. Let P^{max} denote the set of all such points. Let L be the linked list of points in P^{max}, sorted in non-increasing order. We can answer the colored range query by walking along the linked list L until we reach the value q_1 and report the colors of the points encountered. It can be shown that there exists a point $l \in P$ of color c in interval q if and only if there exists a unique point $m \in P^{max}$ of color c in interval q. We can solve the other case, i.e, $q = [-\infty, q_2]$ in a similar manner.

We now build a data structure to answer a general colored range query $q = [q_1, q_2]$. The data structure is a trie T [2] built on the values of points $p \in P$. For each node $v \in T$, let P_v denote the set of points contained in the subtree of T rooted at v. At each internal node v, we store a secondary structure, which consists of two semi-infinite query data structures L_v and R_v corresponding to the queries $[q, \infty]$ and $[-\infty, q]$. Note that the structures as described above are sorted linked lists on max/min coordinates of each color in P_v^{max} and P_v^{min}. For each non-root node $v \in T$, let $B(v) = 0$ if v is a left child of its parent and $B(v) = 1$ if v is a right child of its parent. To efficiently search in the trie T, we adopt the hash table approach of Overmars [17]. We assign an index I_v, called node index, for each non-root node $v \in T$. I_v is an integer whose bit representation corresponds to the concatenation of $B(w)$'s, where w is in the path from root to v in T. Define the level of a node v as the length of the path from the root to v in T. We build a static hash table H_i on the indices I_v of all nodes v at level i, $1 \leq i \leq \log U$ [12]. We store the pointer to node $v \in T$ along with I_v in the hash table. The hash tables H_i uses linear space and provides $O(1)$ worst case lookup. The number of nodes in the trie T is $O(n \log U)$. Since each point $p \in P$ might be stored at most once at each level in the lists R_v, L_v, and the height of the trie T is $O(\log U)$, the total size of the secondary structure is $O(n \log U)$. Thus the size of the entire data structure is $O(n \log U)$.

The trie T can be efficiently constructed level by level in a top down fashion. Initially, we sort the point set P to get the sorted list of P_{root}. Let us suppose we have constructed the secondary structures at level $i - 1$. Let z be a node at level i and let v and w be the children of z in T. We partition the sorted list of points in P_z into sorted list of points in P_v and P_w. We then construct in $O(|P_v|)$ time the lists L_v and R_v at node v by scanning P_v once. We then construct the hash table H_i on indices I_v for all nodes v in level i. The total time spent in level i is $O(n)$. Hence the overall time for constructing the data structure is $O(n \log n + n \log U) = O(n \log U)$.

Let $[q_1, q_2]$ be the query interval. Let z_1(resp. z_2) be the leaf of T to which the search path for q_1(resp. q_2) leads. We compute the least common ancestor v of z_1 and z_2 in $O(\log \log U)$ time by doing a binary search on the height of the trie [17]. Let w and z be the left and right child of v. All the points $p \in [q_1, q_2]$ are contained in P_w or P_z. Hence we perform the query $[q_1, q_2]$ on P_w and P_z. Since all points in P_w have value $\leq q_2$ the queries $[q_1, q_2]$ and $[q_1, \infty]$ on P_w gives the same answer. Similarly, queries $[q_1, q_2]$ and $[-\infty, q_2]$ on P_z gives the same answer. Thus we perform two semi-infinite queries $[q_1, \infty]$ and $[-\infty, q_2]$ on P_w and P_z respectively, and the final output is the union of the output of the two semi-infinite queries. Each color in the output list is reported at most twice.

Theorem 1. *Let P be a set of n colored points in $[0, U]$. We can construct in $O(n \log U)$ time, a data structure of size $O(n \log U)$ so that a colored range searching query can be answered in $O(\log \log U + k)$ time, where k is the output size.*

Remark. The previous best-known result for 1-dimensional colored range query uses $O(n)$ space and $O(\log \log U + k)$ query time [15]. Their solution relies on Cartesian trees and Least Common Ancestors(LCA) structures. While Least Common Ancestor structures can be dynamized [7], it remains a challenge to make the Cartesian trees dynamic and persistent. In contrast, our 1-dimensional structure above can be made dynamic and persistent and hence can be extended to 2-dimensions using the sweep-line paradigm, as we show next.

2.2 Colored Range Searching in $2D$

We first consider the case when the query is 3-sided i.e., $q = [x_1, x_2] \times [-\infty, y_2]$. Let M denote the $1D$ data structure described in the previous section. Our approach is to make M dynamic and partially persistent. Then we use the standard sweep-line approach to construct a partially persistent extension of M so that a 3-sided range query can be answered efficiently.

First we describe how to dynamize M. Note that M consists of the trie T with secondary structure L_v, R_v at each node v of T and static hash tables H_i on node indices I_v for all nodes v on a given level i. We use the dynamic hash table of Dietzfelbinger et al [8] instead of the static hash table H_i used in the $1D$ structure. We maintain for each node v, a static hash table $\mathcal{H}_v$ on the colors of points in P_v. We also maintain a balanced binary search tree TL_v (resp. TR_v) on L_v (resp. R_v).

To insert a point p with color c, we insert p in the trie T. We also may have to insert p in the secondary structures L_v, R_v for all nodes v in the root to leaf search path of p in T. We insert p in L_v(resp. R_v) if P_v^{max} (resp. P_v^{min}) has no points with color c, or if the value of p is greater (resp. less) than the value of point with color c in P_v^{max} (resp. P_v^{min}). We can check the above condition using $\mathcal{H}_v$. If the condition is satisfied, we insert p into L_v(resp. R_v) in $O(\log n)$ time using the binary search tree TL_v (resp. TR_v). Finally if a new trie node v is created while inserting p into T, we insert the index I_v into the appropriate hash table H_i. Since we might insert p in all the nodes in the root-leaf path of p in T, the insert operation costs $O(\log n \log U)$ time. We do not require to delete points from M since the sweep-line paradigm would only insert points.

We now describe how to make the above structure partially persistent:

(i) The trie structure is made persistent by using the node copying technique [9] (creating copies of nodes along the search path of p). We number the copies of node v by integers 1 through n. Let c_v denote this copy number of node v.
(ii) We make the lists L_v and R_v persistent across various versions of v using [9].
(iii) We make the hash table H_i, corresponding to each level i of the trie T, persistent by indexing each node v using the static index I_v and the copy number c_v of node v, i.e., new index I'_v is bit concatenation of I_v and c_v.

The color hash table $\mathcal{H}_v$ and the binary search trees TL_v, TR_v need not be made persistent since they are not used by the query procedure.

We perform a sweep-line in the $(+y)$-direction. Initially the sweep-line is at $y = -\infty$ and our data structure D is empty. Whenever the sweep line encounters a point in P, we insert p in D using the persistent scheme described earlier. When the sweep line reaches $y = \infty$, we have the required persistent structure D. Note that D consists of $D_i, 1 \leq i \leq n$, the different versions of the dynamic $1D$ structure got by inserting points in the sweep-line process. We also build a Van Embe Boas tree $\mathcal{T}$ [18] on the y-coordinates of P.

The node-copying technique [9], used to make the trie T and the lists L_v, R_v persistent, only introduces a constant space and query time overhead. Thus, the space used by our persistent data structure D is still $O(n \log U)$ and the query time to perform a $1D$ colored range-searching query on any given version D_i of D is $O(k)$ where k is the output size.

Let $[x_1, x_2] \times [-\infty, y]$ be a three sided query. We perform a predecessor query on y using the Van Embe Boas tree $\mathcal{T}$ to locate the correct version of the persistent structure. This version contains only those points of P whose y-coordinates are at most y, so we perform a 1-dimensional query on this version with the interval $[x_1, x_2]$, as described in Section 2.1. We locate the splitting node v of x_1, x_2 using the hash tables H_i and then walk along the correct version of the secondary structure, i.e. lists L_v, R_v. The predecessor query can be done in $O(\log\log U)$ time and we can perform the $1D$ query in $O(\log\log U + k)$ time where k is the number of distinct colors of points contained in the query.

Theorem 2. *Let P be a set of n colored points in $[0, U]^2$. We can construct a data structure of size $O(n \log U)$ in $O(n \log n \log U)$ time so that we can answer a 3-sided colored range-searching query in $O(\log\log U + k)$ time, where k is the output size.*

We now extend the 3-sided query structure to answer a general 4-sided query. The approach is exactly the same as the one in Section 2.1 where we extended a semi-infinite query structure to a 1D query structure. We build a trie T on y-coordinates of P. At each node v, we store two secondary structures U_v and W_v, which are the 3-sided query structures for point set P_v, built as described above. U_v corresponds to the 3-sided query $q = [x_1, x_2] \times [y, \infty]$ and W_v corresponds to the 3-sided query $q = [x_1, x_2] \times [-\infty, y]$. We also build a hash table H_i for all nodes v of a given level i of the trie T.

The query process is also similar to the 1-dimensional query. We locate the splitting node v in the search path of y_1 and y_2 using the hash table H_i in constant time. Let w and z be the left and right son of v respectively. We then perform two 3-sided queries $q = [x_1, x_2] \times [y_1, \infty]$ and $q = [x_1, x_2] \times [-\infty, y_2]$ on secondary structures U_w and W_z respectively. The final output is the union of the output of the semi-infinite queries. Each color is reported at most four times, at most twice in each 3-sided query.

Theorem 3. *Let P be a set of n colored points in $[0, U]^2$. We can construct a data structure of size $O(n \log^2 U)$ in $O(n \log n \log^2 U)$ time so that we can answer a 4-sided colored range-searching query in $O(\log\log U + k)$ time, where k is the output size.*

3 Colored Point Enclosure in 2D

In this section, we describe a data structure to solve the colored point-enclosure problem in $2D$. We first describe a data structure to solve the problem in $1D$. Then we extend this data structure to answer two dimensional colored point-enclosure queries.

The 1D structure. Let P be a set of n colored intervals whose endpoints lie on $[0, U]$. Fix a parameter $\ell \geq 0$. Starting with the interval $I = [0, U]$, we recursively partition I into ℓ equal sized intervals. The recursion stops when an interval I contains no endpoints of intervals in P. Let T be the tree that corresponds to this recursive divide and let I_v denote the interval corresponding to a node v in T. Let $p(v)$ denote the parent of v in T. There are at most $2nl$ leaves in T.

We call an interval $p \in P$ *short* at a node v if (at least) one of the endpoints of p lies inside I_v. We call p *long* at v if $I_v \subseteq p$. At each node $v \in T$, we store C_v, the set of distinct colors of intervals of P that are long at v but short at $p(v)$. Since any interval $p \in P$ is short for at most two nodes (nodes v where interval I_v contains the endpoints of p) of any given level, the color of p is stored in at most 2ℓ nodes of any given level. Thus,

$$\sum_{v \in T} |C_v| \leq 2n\ell \log_\ell U.$$

Hence, T requires $O(n\ell \log_\ell U)$ space. We can adapt the above recursive partitioning scheme so that we can compute short and long intervals and the set C_v in $O(n\ell)$ time at each level. Hence, T can be constructed in $O(n\ell \log_\ell U)$ time.

Let q be a query point. The query procedure searches for q and visits all nodes v in T in the root-leaf search path of q in T. At each node, we report all the colors in C_v. Each color can be reported once in each level. Hence, the query time is $O(k \log_\ell U)$, where k is the output size.

We improve the query time to $O(\log_\ell U + k)$ using a simple technique. After we build the data structure as described above, we traverse every root-leaf path in T starting from the root and perform the following operation. Let $v_0, v_1, \ldots v_m$ be any root to leaf path in T. Starting with $i = 1$, we recursively calculate $C'_{v_i} = C_{v_i} \setminus C'_{v_{i-1}}$ for $1 \leq i \leq m$. We store C'_v at v. By definition, $C'_{v_i} \cap C'_{v_j} = \emptyset, 1 \leq i, j \leq m, i \neq j$. This implies that each color is present in at most one list C'_v in any root to leaf path of T. The above operation can be efficiently performed by traversing T in a top-down fashion level by level. The query time thus reduces to $O(\log_\ell U + k)$, where k is the output size.

Theorem 4. *Let P be a set of n colored intervals whose endpoints lie in $[0, U]$. We can construct a $O(n\ell \log_\ell U)$ sized data structure in $O(n\ell \log_\ell U)$ time so that a colored point-enclosure query can be answered in $O(\log_\ell U + k)$ time where k is the output size.*

The 2D structure. We now extend the 1D structure to 2D. Let P be a set of n colored rectangles whose endpoints lie on a $U \times U$ grid. The data structure is similar to a multi-resolution structure like quad-tree. Starting with the square $[0, U]^2$, we recursively divide the current square into ℓ^2 equal-sized squares, where ℓ is a parameter. The recursion stops when a square does not contain any endpoints of the input rectangles. Let T be the tree denoting this recursive partition, and let the square S_v correspond to the node v in T. Some of the notation used in the description of the $1D$ structure will be redefined below.

We call a rectangle r *long* at v if $S_v \subseteq r$; we call r *short* if at least one vertex of r lies in S_v; we say that r *straddles* v if an edge of r intersects S_v but r is not short at v. Figure 1 illustrates the above cases. If r straddles v, then either a horizontal edge or a vertical edge of r intersects S_v, but not both.

For each node v, let C_v denote the set of distinct colors of rectangles that are long at v but short at $p(v)$. Let Σ_v denote the set of rectangles that straddle v and are short at $p(v)$. Set $\chi_v = |C_v|$ and $m_v = |\Sigma_v|$. If a rectangle r is short at v, its color could be stored in all children of v, but it straddles at most 4ℓ children of v. Since a rectangle is short for at most four nodes of any given level,

$$\sum_{v \in T} \chi_v = O(n\ell^2 \log_\ell U) \text{ and } \sum_{v \in T} m_v = O(n\ell \log_\ell U).$$

We store C_v at v. We also store at v, two 1D point-enclosure data structures $T_v^{=}, T_v^{\parallel}$ as its secondary structures. Let $\Sigma_v^{=}$ (resp. $\Sigma_v^{\parallel}$) denote the set of rectangles in Σ_v whose horizontal (resp. vertical) edges intersect S_v. For a rectangle $r \in S_v$, let $r^{=}$ (resp. $r^{\parallel}$) denote the x-projection (resp. y-projection) of $r \cap S_v$. See Figure 1 (iii). Let $\mathcal{I}_v^{=} = \{r^{=} \mid r \in \Sigma_v^{=}\}$ and $\mathcal{I}_v^{\parallel} = \{r^{\parallel} \mid r \in \Sigma_v^{\parallel}\}$. $T_v^{=}$ (resp. $T_v^{\parallel}$) is the 1D point-enclosure data structure on the set of intervals $\mathcal{I}_v^{=}$ (resp. $\mathcal{I}_v^{\parallel}$). By Theorem 4, $T_v^{=}, T_v^{\parallel}$ require $O(m_v \ell \log_\ell U)$ space. Hence, T requires $O(n\ell^2 \log_\ell^2 U)$ space. T can be constructed in $O(n\ell^2 \log_\ell^2 U)$ time in a top-down manner.

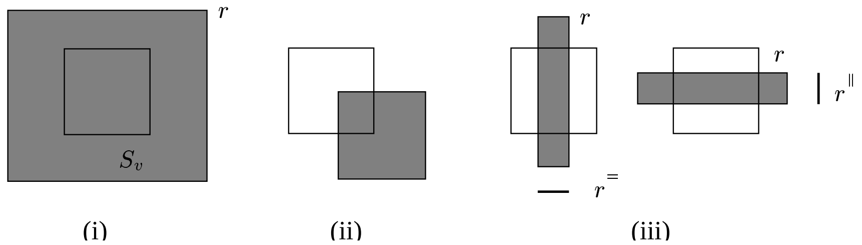

Fig. 1. (i) r long at v; (ii) r short at v; (iii) r straddles v.

Let $q = (q_x, q_y) \in [0, U]^2$ be a query point. The query procedure searches for q in T and visits all nodes v in T in the root-leaf search path of q in T. At each node, we report all the colors in the list C_v. For a rectangle $r \in \Sigma_v^{=}$, $q \in r$ if and only if $q_x \in r^{=}$. Similarly for $r \in \Sigma_v^{\parallel}$, $q \in r$ if and only if $q_y \in r^{\parallel}$. Therefore we query the 1D structures $T_v^{=}$ and $T_v^{\parallel}$ with q_x and q_y respectively. The query time for the $1D$ structure is $O(\log_\ell U + k)$, where k is the number of distinct colors of intervals containing the query point p. Since we make exactly two $1D$ queries at each level and each color can be duplicated once in each level of T, the total query time is $O(k \log_\ell U)$, where k is the number of distinct colors of rectangles containing the query point q.

Theorem 5. *Let P be a set of n colored rectangles whose endpoints lie in $[0, U]^2$. We can construct a $O(n\ell^2 \log_\ell^2 U)$ sized data structure in $O(n\ell^2 \log_\ell^2 U)$ time so that a colored point-enclosure query can be answered in $O(k \log_\ell U)$ time where k is the output size.*

Setting $\ell = U^\epsilon$ and using input mapping, we get

Theorem 6. *Let P be a set of n colored rectangles whose endpoints lie on the grid $[0, U]^2$. We can construct a $O(n^{1+\epsilon})$ sized data structure in $O(n^{1+\epsilon})$ time so that a colored point-enclosure query can be answered in $O(\log\log U + k)$ time where k is the output size.*

Note that if we make the assumption that $U = O(n^c)$, we can remove the input mapping and thus the above query time gets reduced to $O(k)$, where k is the output size.

4 Extensions

We show how to extend our solutions to higher dimensions. We will first sketch how to extend our two dimensional colored point-enclosure data structure to higher dimensions. Using this, we construct a data structure to solve the colored range searching problem in higher dimensions as well.

Colored point enclosure in higher dimensions. We show how to extend the above data structure to answer point-enclosure in d-dimensions. Let P be a set of n colored hyper-rectangles whose endpoints lie on the grid $[0, U]^d$. We construct a tree T using a recursive partitioning scheme. At each node v of T, we define long and short hyper-rectangles and hyper-rectangles that straddle at v. For each node v, let C_v denote the set of distinct colors of rectangles that are long at v but short at $p(v)$. At v, we store C_v and a family of $(d-1)$-dimensional point-enclosure structures as secondary structure. The query process is similar to the 2D case. We omit the details of the structure for lack of space. Using the same analysis as in Section 3, we can show that the space used by the data structure is $O(n\ell^d \log_\ell^d U)$ and the query time is $O(k \log_\ell^{d-1} U)$, where k is the output size. This gives us an analogous result as in Theorem 5, for any dimension d. In particular, for a fixed d, setting $\ell = U^\epsilon$ and using input mapping, we obtain the following.

Theorem 7. *Let P be a set of n colored hyper-rectangles whose endpoints lie on the grid $[0, U]^d$, for a fixed d. We can construct a $O(n^{1+\epsilon})$ size data structure in $O(n^{1+\epsilon})$ time so that a colored point-enclosure query can be answered in $O(\log\log U + k)$ time, where k is the the output size.*

Note that we can reduce the above query time to $O(k)$ by removing the input mapping, if we make an assumption that $U = O(n^c)$ for some constant c.

Colored range searching in higher dimensions. Our approach is to first solve the more general colored rectangle intersection problem in higher dimensions, which is defined as follows. We wish to preprocess a set P of n hyper-rectangles, whose endpoints lie in $[0, U]^d$, into a data structure so that we can report the list of distinct colors of hyper-rectangles in P that intersect a query hyper-rectangle r. We present a simple reduction between the rectangle-intersection problem in d dimensions and the point-enclosure problem in $(2d)$- dimensions.

Let the hyper-rectangle r be represented as $[x_{11}, x_{12}] \times \cdots \times [x_{d1}, x_{d2}]$. We map r to a hyper-rectangle r' in $(2d)$- dimensions given by $[-\infty, x_{12}] \times [x_{11}, \infty] \times \cdots \times [-\infty, x_{d2}] \times [x_{d1}, \infty]$. The query hyper-rectangle $q = [y_{11}, y_{12}] \times \cdots \times [y_{d1}, y_{d2}]$ is mapped to a $(2d)$- dimensional point $q' = (y_{11}, y_{12}, \ldots, y_{d1}, y_{d2})$. It is easy to see that a hyper-rectangle r intersects a hyper-rectangle q iff the hyper-rectangle r' contains the point q'. This observation gives us a reduction between rectangle-intersection problem and point-enclosure problem. Applying the reduction and using Theorem 7, we obtain the following.

Theorem 8. *Let P be a set of n colored hyper-rectangles whose endpoints lie on the grid $[0, U]^d$. We can construct a $O(n^{1+\epsilon})$ sized data structure in $O(n^{1+\epsilon})$ time so that a colored rectangle-intersection query can be answered in $O(\log\log U + k)$ time, where k is the output size*

The colored rectangle-intersection problem reduces to a colored range-searching problem when the input hyper-rectangles are points. Hence we get analogous results as Theorem 8 for colored range-searching in higher dimensions too. Note that in 2D, we can answer a colored range query in $O(\log\log U + k)$ time using a $O(n \log^2 U)$ size data structure, as described in Section 2.

References

1. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison Wesley Press, 1983.
3. S. Alstrup, G. Brodal, and T. Rauhe. New data structures for ortogonal range searching. In *Proc. 41th Annual IEEE Symp. Foundations of Comp. Sci.*, pages 198–207, 2000.
4. J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.
5. B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
6. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, June 1988.
7. R. Cole and R. Hariharan. Dynamic LCA queries. In *Proc. 10th Annual Symposium on Discrete Algorithms*, pages 235–244, 1999.
8. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23:738–761, 1994.
9. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

10. D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proc. 12th Annual Symp. on Discrete Algorithms*, pages 827–835, 2001.
11. P. Ferragina, N. Koudas, D. Srivastava, and S. Muthukrishnan. Two-dimensional substring indexing. In *Proc. of Intl Conf. on Principles of Database Systems*, pages 282–288, 2001.
12. M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with o(1) worst case access time. *J. Assoc. Comput. Mach.*, 31:538–544, 1984.
13. J. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: Counting,reporting and dynamization. In *Proc. 3rd Workshop on Algorithms and Data structures*, pages 237–245, 1993.
14. R. Janardan and M. Lopez. Generalized intersection searching problems. *J. of Comp. Geom. and Appl.*, 3:39–70, 1993.
15. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual Symposium on Discrete Algorithms*, 2002.
16. J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 725–764. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
17. M. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9:254–275, 1988.
18. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.