Detecting VLSI Layout and Connectivity Errors in a Query Window

Ananda Swarup Das* Prosenjit Gupta † Kannan Srinathan ‡

Abstract

The VLSI layout designing is a highly complex process and hence a layout is often subjected to Layout Verification that includes (a) Design Rule Checking to check if the layout satisfies various design rules and (b) Connectivity Extraction to check if the components of the layout are properly electrically connected. In this work we study two geometric query problems which have applications in the above layout verification phase.

1 Introduction

A VLSI chip consists of millions of transistors. Often for the ease of fabrication, these transistors are grouped together to form blocks. Each block has pins on its periphery. Each pin is supposed to carry a signal which is denoted by a net id associated with it. All the pins of the same net id should be connected by wires which is done in the routing phase of VLSI physical design life cycle. The routing phase is again divided into two phases namely (a) global and (b) detailed. In the global routing phase the regions through which the routing is to be carried out are decided. The actual wiring is done in the detailed routing phase. The detailed routing phase is again of two types namely (i) detailed unrestricted routing and (ii) detailed restricted routing. In the detailed restricted routing phase, either channels or switch-boxes are used for routing. In this work, we will assume that channels are used for routing. However, our work has applications even when switch-boxes are used. In fact our first problem has applications even when detailed unrestricted routing is used.

Channels are basically routing regions and are abstracted as rectangles with two sides being open. We present a pictorial representation of channels in Figure 1. In a channel, pins are placed on the either side of the boundaries. The dotted horizontal lines in Figure 1 are the tracks. The thick black horizontal segments on the tracks are the trunks. Pins are connected to the trunks using branches and two trunks on two different tracks are connected using doglegs.

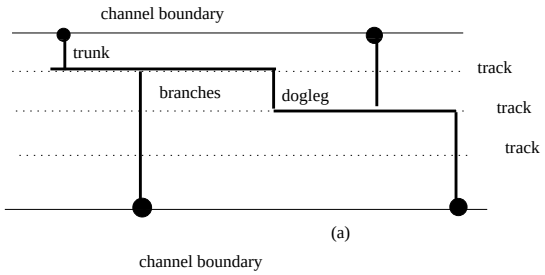


Figure 1: The figure depicts a channel. Pins are on either of the boundaries of the channel.

It should be noted that no two pins of different signal ids should be connected to the same trunk. Also, no two trunks being connected to pins of different signal ids should be connected. As stated before, all the pins of same net id should be electrically connected. Consider Figure 1. Let all the pins are of same signal id and hence they are connected. Once the routing phase is over, a designer is often interested to find (a) if there exist two pins which should have been connected but are not (b) if two pins of different signal ids are connected to the same trunk and (c) no two trunks connected to pins of different signal ids are connected by a dogleg. While working with a layout editor, a designer often zooms into a particular section of the layout to find errors within his/her window of interest. In this work we design data structures using which a designer can efficiently decide the presence of any of the three conditions within the query of interest.

2 Assumptions

In an abstract sense, each pin can be considered as a node of a graph. Two nodes of the graph share an undirected edge if they are directly connected by wires. Assume a, b, c to be the nodes of a graph where a, b share an edge and b, c share an edge. We consider that the nodes a, b, c all are connected as a signal originating from the pin a can reach c through b. See Figure 2 (1).

Each graph is like a connected component. It can be noticed that if two pins are not connected, they will belong to two different connected components. Each connected component can be assigned a unique id. In our work, we assume that (a) each pin is given to us as a point in plane, its net id as *sid* and the id of the

*International Institute of Information Technology , Hyderabad, India, anandaswarup@gmail.com

†Heritage Institute of Technology, Kolkata, India , prosenjit_gupta@acm.org

‡International Institute of Information Technology , Hyderabad, India, srinathan@iiit.ac.in

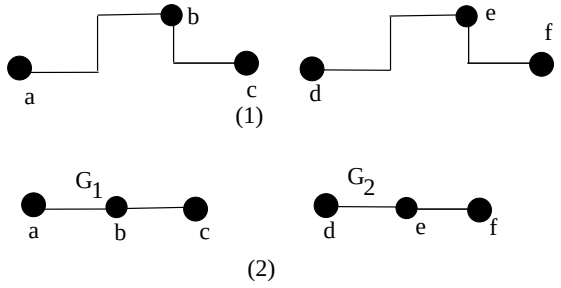


Figure 2: (1) The black nodes are the pins. The pins a, b, c are connected, so are the pins d, e, f. (2) G_1 is the graph with the nodes a, b, c and G_2 is the graph corresponding to d, e, f.

connected component to which it belongs as the *gid*, (b) each trunk is given as a horizontal segment and is assigned a color which is equal to the *sid* of one of the pins it is connected to, (c) each branch is given as a vertical segment and is assigned a color which is equal to the *sid* of the only pin it is connected to and (d) each dogleg is given as a vertical segment and is assigned a color which is equal to the *sid* of one of the two trunks it is connected to.

3 Problem Definitions

In the rest of the paper, we refer to the following condition as *Condition 1:*

Condition 1: $sid(p_1) = sid(p_2)$, but $gid(p_1) \neq gid(p_2)$.

In this work, we study the following two problems

Problem 3.1 *We are given a set S of n points in $\mathbb{R}^2$. Each point $p \in S$ has two colors,namely, $sid(p)$ and $gid(p)$, associated with it along with its coordinates. We need to preprocess them into a data structure such that given a query rectangle $q = [a, b] \times [c, d]$ we can decide if there exist two points $(p_1, p_2) \in S \cap q$ such that $sid(p_1) = sid(p_2)$ but $gid(p_1) \neq gid(p_2)$.*

Problem 3.2 *Let H and V be respectively the sets of horizontal and vertical segments in $\mathbb{R}^2$. Each horizontal segment $h \in H$ (resp. $v \in V$) has a color namely, $sid(h)$ (resp. $sid(v)$) associated with it. The $sid(h)$ (resp $sid(v)$) is not necessarily unique. We need to preprocess H and V into a data structure such that given a query rectangle $q = [a, b] \times [c, d]$, we can efficiently decide if there exists a pair of horizontal-vertical segments (h, v) such that $h \cap v \cap q \neq \emptyset$ and $sid(h) \neq sid(v)$.*

4 Solutions for the Problem 3.1

4.1 Solution 1: A Simple Idea

4.1.1 Preprocessing

We divide the points of the set S into subsets $S_1, \ldots, S_k$ where the subset S_i contains the points with *sid* equal to i. Next, we sort the points in S_i according to their *gid*s. For every pair of points $p, m \in S_i$, we create a 4-d point (p_x, p_y, m_x, m_y) if $gid(p) \neq gid(m)$ and store it into a data structure D for range searching in $\mathbb{R}^4$.

4.1.2 Query Algorithm

Given a query rectangle $q = [a, b] \times [c, d]$, we search the data structure D with the query $q' = [a, b] \times [c, d] \times [a, b] \times [c, d]$. If we find any point in q', we return "YES", else we return "No".

Lemma 1 *Using the data structure of [8] for range searching in $\mathbb{R}^4$, a data structure of size $O(n^2(\frac{\log n}{\log \log n})^3)$ can be constructed such that given a query rectangle q we can decide in $O(\frac{\log^2 n}{\log \log n})$ time, if there is any instance of Condition (1) inside q.*

4.2 Solution 2: Improving the Storage Space While Trading-off Query Time

4.2.1 Preprocessing:

For each point $p \in S$ we create two points namely p_1 and p_2. We set the coordinates of both the points to (p_x, p_y). We then color p_1 with the color $sid(p)$. The point p_2 is colored with a composite color which uniquely represents the chromatic pair $< sid(p), gid(p) >$. We store the points p_1 in a set S_L and the points p_2 in S_B. We preprocess the points in S_L and S_B into two data structures D_L and D_B respectively. D_L and D_B are instances of generalized two-dimensional orthogonal range counting.

4.2.2 Query Algorithm:

Given a query rectangle q, we first find the distinct colors of the points of the set S_L that are present in q. This is done by querying D_L with q. Let the number of distinct colors of the points of the set S_L present in q be n_L. Next, we find the distinct colors of the points of the set S_B that are present in q by searching the data structure D_B. We call this value as n_B. If $n_L < n_B$, we return "YES". Else, we return "No".

Lemma 2 *There exists an instance of Condition (1) inside the query rectangle iff $n_L < n_B$.*

Hence we have the following result

Lemma 3 *There exists a data structure ([1]) of size $O(n^2 \log^2 n)$ such that given a query rectangle q we can decide in $O(\log^2 n)$ time, if there is any instance of Condition (1) inside q. A space-time trade of data structure ([3]) with a space bound of $O((n/r)^2 \log^6 n + n \log^4 n)$ and a query time of $O(r \log^7 n)$ such that $1 \leq r \leq n$ is also possible.*

4.3 Solution 3: Further Improving the Storage Space

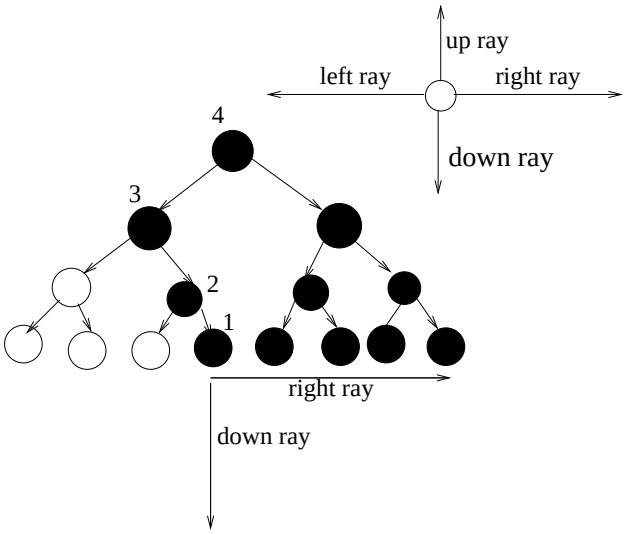


Figure 3: The arrow marks indicate that each of the rays are extending towards ∞. The right ray will be allocated to all the nodes marked black. The down ray will be allocated to the nodes marked 1, 2, 3 and 4

In this section, we propose a solution which needs $O(n^2)$ storage space. Using the data structure, we can solve the Problem 3.1 in $O(\log^2 n)$ time.

4.3.1 Preprocessing Stage 1→ Assignment of rays:

1. Consider the point set S. From each point $p \in S$, we shoot four rays namely two horizontal rays, one traveling $-\infty$, the other traveling $+\infty$ and two vertical rays one traveling $-\infty$, the other traveling $+\infty$.

2. We call the horizontal rays traveling towards $-\infty$ as *left rays*, horizontal rays traveling towards $+\infty$ as *right rays*, vertical rays traveling towards $+\infty$ as *up rays* and the vertical rays traveling $-\infty$ as *down rays*. See Figure 3.

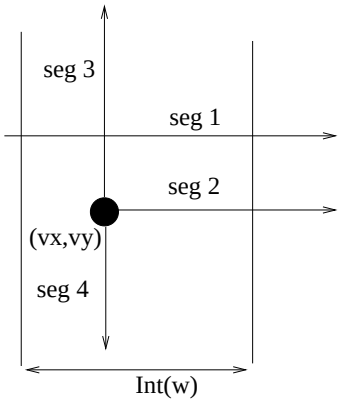
3. Let us sort the points of the set S in terms of their x coordinates. Construct a balanced binary search tree T_x whose leaf nodes corresponds to the elementary intervals being induced by the x coordinates of the points of the set S.

4. Each internal node $\mu \in T_x$ stores an interval $Int(\mu)$ which is union of the elementary intervals being stored in the leaf nodes of the subtree rooted at μ.

5. Now consider all the right rays, up rays and the down rays. To the node $\mu \in T_x$, we allocate the right ray emanating from the point p if $Int(\mu) \cap [p_x, \infty) \neq \emptyset$ where p_x is the x coordinate of the point p. Refer to Figure 3.

6. For each up ray (resp. down ray), we first find the leaf node storing the x coordinate of the up ray or the down ray. Then, starting from the leaf node τ, we allocate it to all the ancestors of the leaf node.

4.3.2 Preprocessing Stage 2 → Classification of rays at the node w :



Segments allocated to node w

Figure 4: In this figure, *seg* 1 is a right ray completely covering $Int(w)$. *seg* 2 is a right ray partially overlapping with $Int(w)$. *seg* 3 is an up ray and *seg* 4 us a down ray.

1. Consider a node $w \in T_x$ and a ray assigned to the node w. Let the x coordinate of the end point of the ray be p_x. With the concerned ray, the following are the possibilities:

   - $Int(w) \subseteq [p_x, \infty)$ that is the interval of node w is completely contained in the semi-infinite interval $[p_x, \infty)$, the horizontal projection of the ray.
   - $Int(w) \cap [p_x, \infty) \neq \emptyset$, that is the interval $Int(w)$ is not completely contained in $[p_x, \infty)$ but they are overlapping.

- The ray is an up ray or a down ray.

2. Let $L_{w,F}$ is the list of rays whose horizontal projection completely contains $Int(w)$. Let $L_{w,P}$ is the list of rays whose horizontal projections are partially overlapping with $Int(w)$. Let $L_{verti,w}$ be the list of up and down rays allocated to w or in other words whose x coordinates are stored in the leaf nodes of the subtree rooted at w.

See Figure 4. In that figure, the horizontal ray denoted by *seg* 1 belongs to $L_{w,F}$, *seg* 2 belongs to $L_{w,P}$. The up ray and the down ray will belong to $L_{verti,w}$.

4.3.3 Preprocessing Stage 3 → Creation of 2-d and 3-d points:

1. Consider any horizontal ray $h \in L_{w,F}$. Let the coordinates of the end point of the h be (h_x, h_y).
2. Check if there is any vertical ray $v \in L_{verti,w}$ such that $sid(h) = sid(v)$ but $gid(h) \neq gid(v)$.
3. Let there be some vertical rays v. We will denote the coordinates of the end points of v as (v_x, v_y).
   - Among all the down rays (respectively up rays) select the one whose v_y is just above (respectively just below) h_y. Let that v_y be denoted as $v_{y,1}$ for the down ray and $v_{y,2}$ for the up ray.
   - We create two 3-d points $(h_x, h_y, v_{y,1})$, $(h_x, v_{y,2}, h_y)$.
4. Similarly, for each horizontal ray $h \in L_{w,P}$ which is partially overlapping with $Int(w)$,
   - check if there is any vertical ray $v \in L_{verti,w}$ such that $sid(h) = sid(v)$ but $gid(h) \neq gid(v)$ and $h \cap v \neq \emptyset$.
   - Among all the down rays (respectively up rays) select the one whose v_y is just above (respectively just below) h_y. We will denote that as $v_{y,1}$ for the down ray and $v_{y,2}$ for the up ray.
   - We create two 2-d points $(h_y, v_{y,1})$, $(v_{y,2}, h_y)$.

See Figure 4. In that figure, the *seg* 1 will contribute to 3-d points and the *seg* 2 will contribute to 2-d points.

4.3.4 Preprocessing Stage 4 → Storing the 2-d and 3-d points:

1. We store the 3-d points in a 3-d dominance reporting data structure $D_{3,w}$ of [5].
2. The 2-d points are stored in a priority search tree $T_{PST,w}$ [7].

Lemma 4 *The storage space needed by the above data structure is* $O(n^2)$.

4.3.5 Query Algorithm:

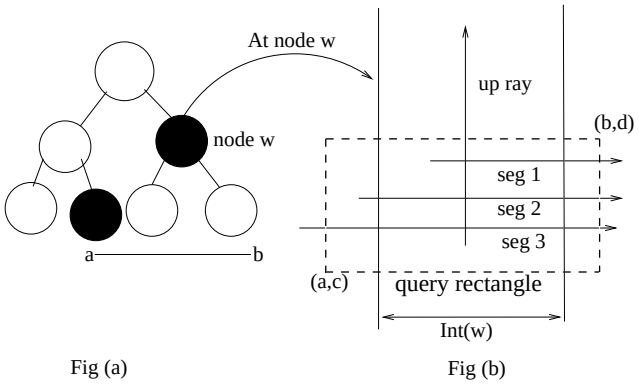


Figure 5: The segment $[a, b]$ of the query rectangle $q = [a, b] \times [c, d]$ is allocated to the nodes marked black. The scenario at a node w to whom $[a, b]$ is allocated.

1. Given a query rectangle $q = [a, b] \times [c, d]$ we first allocate the segment $[a, b]$ to the nodes of T_x. The rule that we follow for allocating $[a, b]$ to the node μ is $Int(\mu) \subseteq [a, b]$ but $Int(parent(\mu)) \nsubseteq [a, b]$. It should be noted that the way the segment $[a, b]$ is allocated is not the same as the way we allocate the horizontal rays. It should also be mentioned that the segment $[a, b]$ will be allocated to $O(\log n)$ nodes of the tree T_x.
2. Let S_{can} be the set of $O(\log n)$ canonical nodes to which the segment $[a, b]$ is allocated. At each node $w \in S_{can}$ first search the priority search tree $T_{PST,w}$ with the query $[c, \infty) \times (-\infty, d]$.
   - If we find a point in $T_{PST,w} \cap [c, \infty) \times (-\infty, d]$, we return "YES".
   - Else we search the 3-d dominance data structure $D_{3,w}$ with the query $[a, \infty) \times [c, \infty) \times (-\infty, d]$. If we find a point in $D_{3,w} \cap [a, \infty) \times [c, \infty) \times (-\infty, d]$, we return "YES".

Lemma 5 *The query time of the above algorithm is* $O(\log^2 n)$.

We therefore summarize the results of the Problem 3.1 with the following theorem

Theorem 6 *Given a set S of n points in $\mathbb{R}^2$ such that each point $p = (p_x, p_y)$ has two associated colors $gid(p), sid(p)$ associated with it,*

1. *There exists a data structure of size $O(n^2(\frac{\log n}{\log\log n})^3)$ can be constructed such that given a query rectangle q we can decide in $O(\frac{\log^2 n}{\log\log n})$ time, $sid(p_1) = sid(p_2)$ but $gid(p_1) \neq gid(p_2)$.*

2. *There exists a data structure of size $O(n^2 \log^2 n)$ such that given query rectangle q, in $O(\log^2 n)$ time we can decide if there exist two points p_1, p_2 in q such that $sid(p_1) = sid(p_2)$ but $gid(p_1) \neq gid(p_2)$.*

3. *A space-time trade off data structure with storage bound $O((n/r)^2 \log^6 n + n \log^4 n)$ and query time $O(r \log^7 n)$ is also possible. Here r is a user defined parameter.*

4. *We also have a data structure with storage space requirement of $O(n^2)$ and query time $O(\log^2 n)$ to answer the same query.*

5 Solution for the Problem 3.2

Consider a horizontal vertical segment intersection inside a query rectangle. There are two possible scenarios
(a) at least one end point of either of the segments is inside the query rectangle. We call this kind of intersections as intersections of type 1.
(b) Both the end points of both the segments are outside q that is the segments completely cross the query rectangle. We call this kind of intersections as intersections of type 2.

For dealing with the first situation, we consider the case where the lower end point of the vertical segment is inside the query rectangle. Similar arrangements have to be done for the upper end point.

5.1 Preprocessing a data structure

5.1.1 Preprocessing Phase 1→ Creation of 2-d points:

Consider the lower end point p'' of a vertical segment v. Let the y coordinate of p'' be v_y. We find the horizontal segment h whose y projection is just above v_y and $sid(v) \neq sid(h)$. Let the y projection of h be h_y. We create a 2-d point (v_y, h_y). The step has to be repeated for all the vertical segments of the set V.

5.1.2 Preprocessing Phase 2→ Constructing a Segment tree:

Let M' be the sorted list of the x coordinates of the end points of the horizontal and vertical segments in H and V respectively. We construct a segment tree T_x whose leaf nodes correspond to the elementary intervals induced by the x coordinates of the set M'. Each internal node $\mu \in T_x$ stores an interval $Int(\mu)$ which is union of the elementary intervals being stored in the leaf nodes of the subtree rooted at μ. A horizontal segment $h \in H$ is allocated to a node $\mu \in T_x$ if $Int(\mu)$ is completely contained in the horizontal projection of the h whereas $Int(parent(\mu))$ is not. For each vertical segment v we locate the leaf node in T_x which stores the x coordinate of v.Then starting from that leaf node, we store a copy of v in all the ancestors of the leaf node including the root of T_x.

5.1.3 Preprocessing Phase 3→ Auxiliary dominating set finding data structures:

Consider a node $\mu \in T_x$. Let $L_{H,\mu}$ and $L_{V,\mu}$ be the set of horizontal and vertical segments allocated to the node μ. At the node μ, we do the following:

1. Consider a vertical segment $v \in L_{V,\mu}$ and consider the point (v_y, h_y) we have created in phase 1. We store the point in a priority search tree $T_{\mu,1}$. This has to be done for all the vertical segments in $L_{V,\mu}$.

2. Next, for each $v \in L_{V,\mu}$ whose y projection is $[v_{y_1}, v_{y_2}]$, we create a 2-d point (v_{y_1}, v_{y_2}). We store these points in a priority search tree $T_{\mu,2}$.

3. Finally, for each $v \in L_{V,\mu}$, we create a 3-d point $(v_{y_1}, v_{y_2}, sid(v))$. We store these points in a 3-d dominance reporting data structure $T_{\mu,3}$ of [5]. The $T_{\mu,3}$ will support queries of the form $[x, \infty) \times (-\infty, y] \times [z, \infty)$ and $[x, \infty) \times (-\infty, y] \times (-\infty, z]$.

5.1.4 Preprocessing Phase 4→ Range Minima data structure:

Consider any node $\mu \in T_x$. Consider the horizontal segments $h \in L_{H,\mu}$. We store the y coordinates of these horizontal segments in an array $Y_{sort,\mu}$ in a sorted order. Next, we create a 1-d range minima data structure RM_μ [6] such that given two indices of the array $Y_{sort,\mu}$, we can decide in $O(1)$ time if all the horizontal segments whose y coordinates are stored in between the two query indices in the array $Y_{sort,\mu}$ have the same sid.

5.2 Query Algorithm

Given a query rectangle $[a, b] \times [c, d]$, allocate the segment $[a, b]$ to a node μ of the segment tree T_x if $Int(\mu) \subseteq [a, b]$ but $Int(parent(\mu)) \nsubseteq [a, b]$. The segment $[a, b]$ will be allocated to $O(\log n)$ nodes of the tree T_x.

5.2.1 Intersections of type 1

Let S_{can} be the set of such nodes. Let us first focus on the intersections of type 1 that is when at least one of the end points of the intersecting segments are inside the query rectangle. We will explain our steps assuming that our focus is on the lower end points of the vertical segments.

1. Any vertical segment that intersects the query rectangle q is present in any of the nodes $u \in S_{can}$. We

therefore search the priority search tree $T_{u,1}$ $\forall u \in S_{can}$ with the query $q' = [c, \infty) \times (-\infty, d]$.

2. While searching $T_{u,1}$ with q', if we find any point p in q', we return "Yes" and Exit.

5.2.2 Intersections of type 2

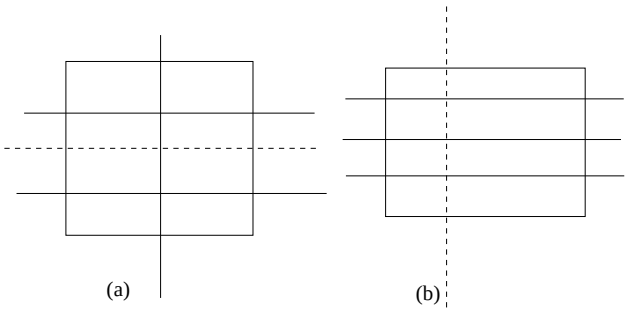


Figure 6: (a) All the line segments except the dotted horizontal segment are of same *sid*. (b) All the horizontal segments are of same *sid*. The dotted vertical segment is of different *sid*.

When the end points of the intersecting horizontal and vertical segments are outside the query rectangle, we do the following:

1. At each node $\mu \in S_{can}$ and the ancestors of the node μ up to root in the tree T_x, we search the array Y_{sort} at the respective nodes to find the the indices i, j such that $c \leq Y_{sort}[i] < Y_{sort}[j] \leq d$.

2. By using the range minima data structure RM at that respective node, we then decide if all the horizontal segments whose y projections are in between the indices i and j have the same *sid*.

3. If we find that all the horizontal segments allocated to the node μ or its ancestors are not of same *sid*, we come back to the node μ and search the data structure $T_{2,\mu}$ with the query $(-\infty, c] \times [d, \infty)$. If there is any point inside $(-\infty, c] \times [d, \infty)$, we return "YES" and Exit. See Figure 6 (a).

4. Suppose all the horizontal segments allocated to the node μ or its ancestors are of same *sid*, we then find that particular *sid*. Next, We come back to the node μ and search $T_{\mu,3}$ with $(-\infty, c] \times [d, \infty) \times [sid+1, \infty)$ and $(-\infty, c] \times [d, \infty) \times (-\infty, sid-1]$. If we find any point in either of the queries, we return "YES" and Exit. See Figure 6 (b).

Lemma 7 *The algorithm returns a "YES" iff there exist a pair of horizontal-vertical segments h,v, such that* $h \cap v \cap q \neq \emptyset$ *and* $sid(v) \neq sid(h)$.

Theorem 8 *There exists a data structure D of size $O(n \log n)$ such that given a query axes parallel rectangle q, we can decide in $O(\log^2 n)$ time if there exists a pair (h, v) where h is a horizontal segment and v is a vertical segment such that h intersects v inside q and $sid(h) \neq sid(v)$.*

References

[1] P. Gupta, R. Janardan, and M. Smid. *Further results on generalized intersection searching problems: counting, reporting, and dynamization.* In Journal of Algorithms, 19 , pp. 282–317, 1995.

[2] A. Agrawal, P. Gupta. *Incremental Analysis of Large VLSI Layouts.* In Integrations, 42(2), 205–210, 2009.

[3] H. Kaplan, N. Rubin, M. Sharir, E. Verbin. *Counting colors in boxes.* In Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 785–794, 2008.

[4] B. Chazelle, H. Edelsbrunner, L. J. Guibas and M. Sharir. *Algorithms for Bichromatic Line Segment Problems and Polyhedral Terrains.* In Algorithmica, 11, pp. 116–132, Springer Verlag, 1994.

[5] P. Afshani. *On Dominance Reporting in* $3D$ Proceedings of 16^{th}European Symposium on Algorithms (ESA), pp. 41–51, 2008.

[6] H. Yuan, M. Atallah. *Data Structures for Range Minimum Queries in Multidimensional Arrays.* In Proceedings of SODA, pp. 150–160, 2010.

[7] E. M. McCreight. *Priority Search Trees.* In SIAM Journal of Computing. vol. 14(2), pp. 257–276, 1985.

[8] P. Afshani, L. Arge, K. D. Larsen *Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements* In Proceedings of Symposium on Computational Geometry, pp. 240–246, 2010.

[9] *www.bwrc.eecs.berkeley.edu/Classes/icbook /magic/index.html*