Extended Essay

Comparison Between the Run Time Efficiency of Scan-line Filling Algorithms Realized by Segment Tree and VEB Tree

Subject: IB Computer Science
Exam Session: May 2018
Word Count: 3996

Table of Contents:

I. Introduction

II. Background

1. Scan-line Filling Algorithm
2. Segment tree
3. VEB Tree

III. Realizing Scan-line Filling Algorithm through Two Different Data Structures

1. Realizing Scan-line Filling Algorithm by Segment tree
2. Realizing Scan-line Filling Algorithm by VEB tree

IV. Proofs and Comparisons between the Theoretical Run Time Requirements of the Two Realizations

1. Scan-line Filling Realized by Segment Tree
2. Scan-line Filling Realized by VEB Tree
3. Comparison

V. Testing Real Time Performances of Two Realizations

1. Testing Environment
2. Testing Methods and Chosen Inputs
3. Results and Comparisons
   - i. Random Input
   - ii. Neat Input
   - iii. Extreme Input

VI. Conclusion

VII. Limitation and Evaluation

VIII. Bibliography

IX. Appendix

1. Java Implementation of VEB Node Structure
2. Java Implementation of VEB Tree Methods
3. Java Implementation of Segment Tree Structure
4. Java Implementation of Segment Tree Methods

I. Introduction:

Scan-line Filling Algorithm is an important algorithm in computer graphics which can calculate total areas of overlapping regions. The basic scan-line filling algorithm is frequently used in algorithm contests, where real time scan-line rendering is used in televisions, allowing televisions to generate image without framebuffers. [1]

Because of the development of technology and the expansion of data, times and resources required to process data become increasingly important. Initially, scan-line filling algorithm is realized through segment tree, which has run time requirement $O(\log n)$. (In this essay, $log_2 n$ is written as $\log n$) With such data structure, scan-line filling algorithm has run time requirement $O(n \log n)$. [2]

However, VEB tree[3] is a tree data structure that is more efficient than segment tree, which ensures an efficiency of $O(\log \log n)$ for all operations.

So, this extended essay will realize scan-line filling algorithm by VEB tree, prove and solve the run time requirement of scan-line filling algorithms realized by VEB tree and then test such algorithms through experiments. Experiment trials are conducted based on randomly generated data, which represents normal situations, and on neat data

[1] "Scanline Rendering." *Wikipedia*, Wikimedia Foundation, 28 June 2017, en.wikipedia.org/wiki/Scanline_rendering#Use_in_realtime_rendering.
[2] "Understanding Segment Tree Step by Step." *Understanding Segment Tree Step by Step*, Cnblogs, 1 Dec. 2013, 22:30, www.cnblogs.com/TenosDoIt/p/3453089.html.
[3] Van Emde Boas Tree (VEB Tree) is a data structure proposed by Dutch computer theorist Peter Van Emde Boas in 1975.

and extreme data, which represents special cases. Through both theoretical and real time comparisons, this extended essay discusses in what situations and to what extent can VEB tree change or improve the run time requirement of scan-line filling algorithms.

II. Background:

1. Scan-line Filling Algorithm

Scan-line Filling Algorithm is an algorithm that process areas of regions, such as Figure 1 below, which contains 5 regions overlapping with each other.

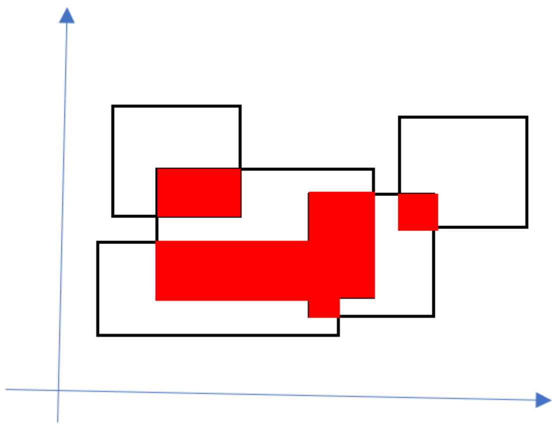


Figure 1: A Typical Graph that can be Analyzed by Scan-line Filling

Because of the overlapping regions, Scan-line Filling algorithm is responsible for avoiding calculating overlapping areas repeatedly. So, the concept of "Scan-line" is introduced to "scan" the graph from bottom to top.

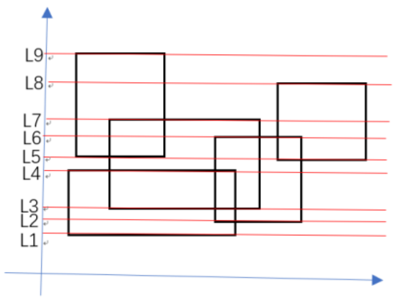


Figure 2: Scan-lines of an Overlapping Figure

For example, for a typical 5-region figure such as Figure 1, 9 scan-lines from bottom to top with names L1, L2, … L9 are created, as illustrated by Figure 2. A scan-line is created when an edge of a region is met, and then scan-lines are sorted with an increasing order regarding their heights. Each scan-line has the following properties: x_1, x_2, h and flag, as shown by Figure 3.

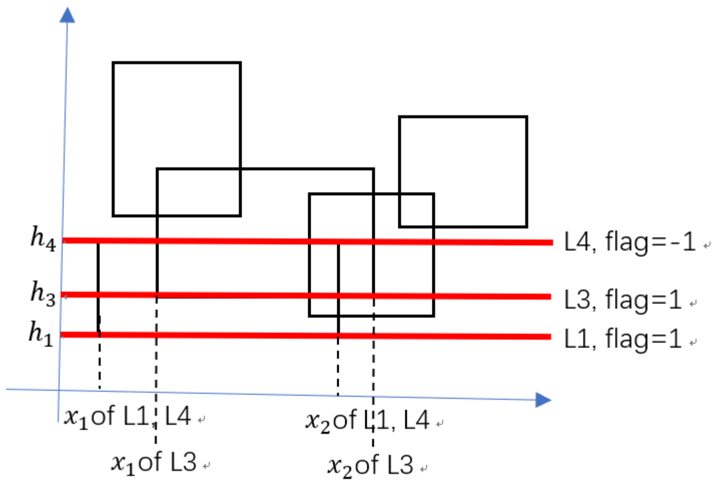


Figure 3: Properties of Scan-lines

x_1 of a scanline is the x-coordinate of the left end point of the edge where the scan-line intersects with one of the regions, while x_2 is the right end-point. H is the height, which is also the y-coordinate of a scan-line. The indicator flag indicates whether the scan-line meets the upper edge or the lower edge of a region.

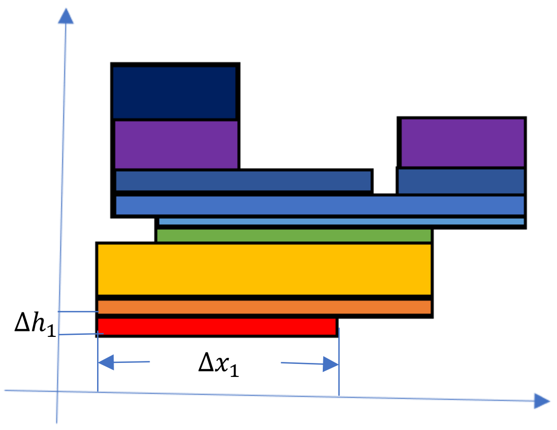


Figure 4: How scan-lines are used to calculate total area

With the help of scan-lines, the figure can be separated into areas that can be calculated by $A = \sum \Delta x \cdot \Delta h$. So, scan-lines are responsible to calculate Δx and Δh. A new set cnt stores all x-coordinates that acts as the indicator of which x-coordinate should be calculated into Δx. Whenever a lower edge is met, which means a scan-line with $flag = 1$ is met, the x-coordinates from x_1 to x_2 should be inserted into cnt. Else, the x-coordinates from x_1 to x_2 should be deleted from cnt. Then, Δx can be calculated from all x-coordinates that are inserted but not deleted from cnt. While Δh can be easily calculated by $h_{i+1} - h_i$, each colored region in Figure 4 can be calculated by $\Delta x \cdot \Delta h$, and the total area can be calculated by $A = \sum \Delta x \cdot \Delta h$

2. Segment Tree

The structure of segment tree is shown by the following diagram:

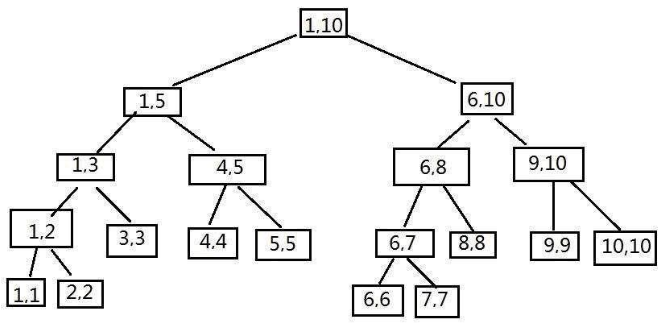


Figure 10: Segment Tree Structure[4]

Segment Tree is a data structure that stores data in its leaves and stores summaries in parent nodes. To store n elements, for example 10 elements, which are originally stored in an array B[], according to Figure 10, we should consider the numbers in Figure 10 indexes of B[]. Leaves of the tree, each containing a single number, are used to store data of B[1], B[2], ... , B[10]. Their parents store some properties of its children, such as the maximum, minimum values in its subtree or the sum of its subtree. For each non-leaf node in a segment tree, its left child maintains the first half of the array it is maintaining, and its right child maintains the second half. So, segment tree is a form of balanced binary tree. For example, for root node [1,10], which stores properties (maximum, minimum, sum) of all elements from B[1] to B[10], its left child stores properties from B[1] to B[5], and its right child stores properties from B[6] to B[10].

[4] "Segment Tree Summary." *Segment Tree Summary*, Cnblogs, 19 Aug. 2017, 15:17, www.cnblogs.com/yiyiyizqy/p/7396693.html.

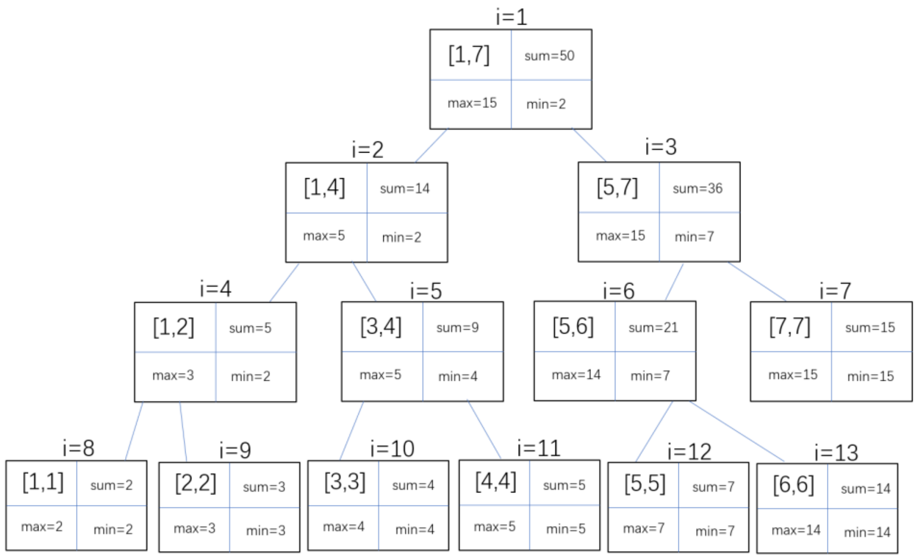


Figure 11: A Complete Segment Tree

Figure 11 is an example of a complete segment tree, which again stores A[]={2, 3, 4, 5, 7, 14, 15}. As we can see, all leaves have the same min, max and sum values, meaning that they store single values. Parent nodes store summaries of all its children. Left and right subtrees each maintain half of their parent.

Segment tree is realized through a node array tree[], which is an array that stores all tree nodes. Inside each node, we can store the interval, max, min and sum as integers. As shown in Figure 11, each node is given an index, where the indexes follow an increasing order from parents to children (from left to right among nodes with same hierarchies). So, for each node that is not leaf with index i, it's left and right children have indexes 2i and 2i+1 respectively.

Though common segment trees maintain properties such as max, min and sum, properties that can be maintained are not restricted to the three listed above. For scan-line filling algorithm, cnt introduced in *Section II.1* should be maintained, whereas max and min are not necessary.

*(Detailed realization of segment tree see Appendix A.3, A.4)

Methods of segment tree, such as update(), has run time requirement of $O(\log n)$. Segment tree, as a balanced binary tree with universe size u, has height log_2u. As only one branch of the tree is traversed in update(), the running time shall not exceed log_2u.

3. VEB Tree

(1) Superimposing a Tree on an Array

Before introducing VEB tree, it is essential to firstly understand the implementation of superimposing a tree of constant height above a bit-vector.

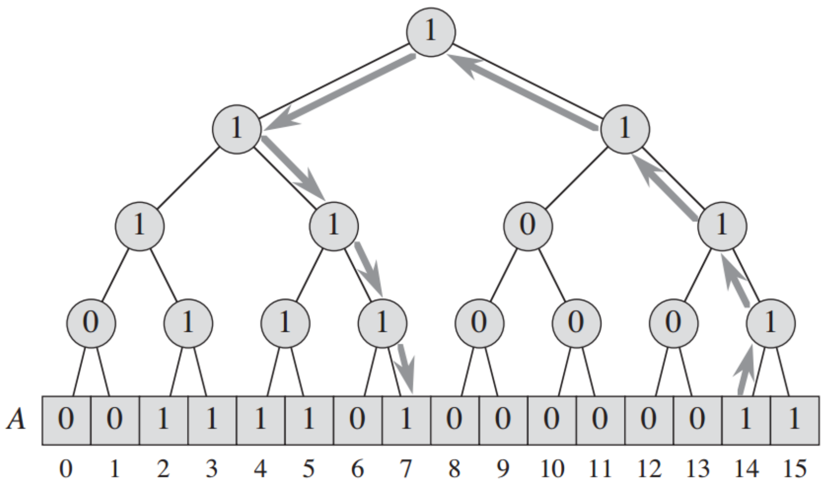


Figure 5: Super-imposing a Binary Tree upon a Bit-array[5]

Suppose array A includes u elements. A dynamic set of values are stored in this

[5] Cormen, Thomas H., et al. *Introduction to Algorithms*. China Machine Press, 2016.

array A[0, 1, 2, …, u-1]. The value of A[x]=1 if x is included in the dynamic set. In the case shown by the diagram, the array A contains 16 indexes and the dynamic set of {2, 3, 4, 5, 7, 14, 15} is stored. If all indexes of array A is taken as leaves of a binary tree, we can create a tree as Figure 5. Each parent node stores the "logic-or" of its children.

This data structure can already ensure an efficiency of $O(\log n)$ for operations of locating a value such as predecessor and successor. The path in the diagram shows the algorithm for locating the predecessor of 14. The worst case has the efficiency $O(\log n)$, as only one way up and one way down is analyzed.[6]

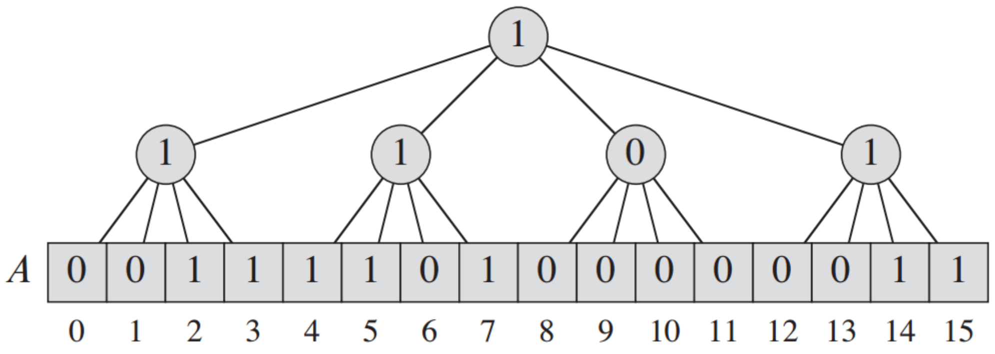


Figure 6: Super-imposing a tree with each Node having $\sqrt{A.length}$ Children[7]

If we make the length of the array A $u = 2^{2^k}$, then instead of a binary tree, we can build another tree, in which each parent node has $\sqrt{u}$ children, a tree of height 2 is built. Still, each node contains the "logic-or" of all its children. Similarly, we can further operate this tree into the data structure shown in the following diagram:

[7] Cormen, Thomas H., et al. *Introduction to Algorithms*. China Machine Press, 2016.

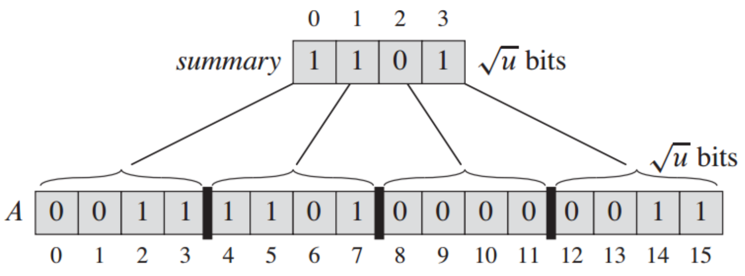


Figure 7: Using an Array to Store Parent Nodes[8]

An array of summary[] is used to store the "or-logic" of its children. Each element of summary[] stores the "or-logic" of its $\sqrt{u}$ children called a cluster. Each cluster contains $\sqrt{u}$ bits and summary[] also contains $\sqrt{u}$ bits. We can use an integer i to represent the number of cluster.

This structure is already similar to the structure of VEB tree, and it has time $O(\sqrt{u})$ for operations of minimum, maximum and delete:

-To find the minimum (maximum) value, find the leftmost (rightmost) entry in summary that contains a 1, say summary[i], and then do a linear search within the ith cluster for the leftmost (rightmost) 1.

-To delete x, let $i = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$. Set A[x] to 0 and then set summary[i] to the logical-or of the bits in the ith cluster.

Here, we introduce two important functions $\mathrm{high}(x)$, $\mathrm{low}(x)$, in which:

$$\mathrm{high}(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$$

$$\mathrm{low}(x) = x \bmod \sqrt{u}$$

Because each cluster has $\sqrt{u}$ elements, $\mathrm{high}(x)$ shows the index of cluster that x belongs to, and $\mathrm{low}(x)$ shows the index of x inside its own cluster.

[8] Cormen, Thomas H., et al. *Introduction to Algorithms*. China Machine Press, 2016.

(2) VEB Tree structure

Similar to imposing a tree on a bit array, each node of VEB Tree has a structure as shown in the following diagram.

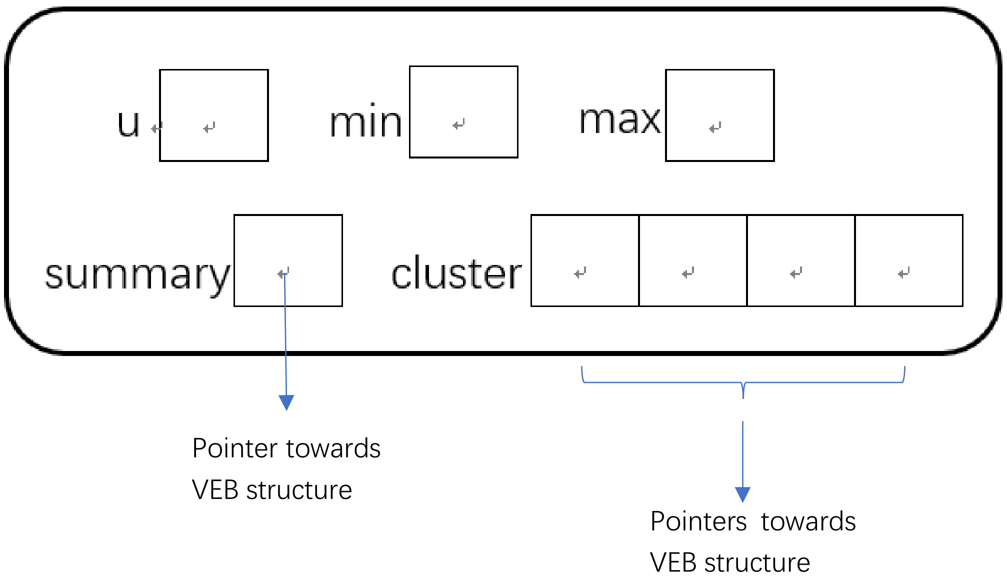


Figure 8: VEB Tree Node Structure

U stores an integer value, which illustrates the number of total elements in the part of bit array that this tree is imposed on.

Min (max) stores an integer value of the minimum (maximum) index in its subtree.

The summary and cluster pointers, which point towards other VEB structures, work similarly as the arrays discussed in superimposing a tree, which are the indicators of whether there are elements in their subtrees.

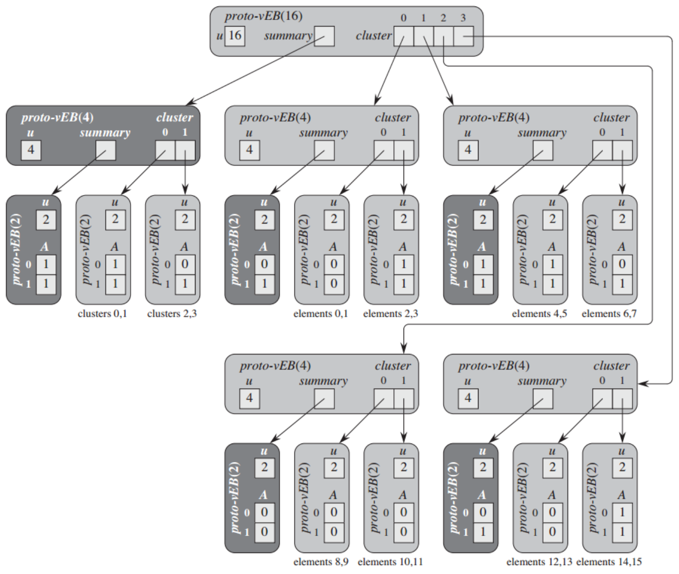


Figure 9: VEB Tree Structure[9]

For example, Figure 9 is an example of a full VEB tree representing data {2, 3, 4, 5, 7, 14, 15}. Each node except leaves contain a summary pointer which summarize the behaviors of its subtree. When u=2, which means that the leaves are reached, the binary values of 0 and 1 indicates whether the element is included or not.

*(Detailed realization of VEB tree see Appendix A.1, A.2)

(3) Run Time Requirement of VEB Tree Methods

Here, we take method contain(), which returns whether an element is stored or not, as an example. To search for an element, a recursive structure should be implemented in order to search through each layers of the trees.

[9] Cormen, Thomas H., et al. *Introduction to Algorithms*. China Machine Press, 2016.

```
1.  public boolean contain(Node V, int x) {
2.          if (V.min == x || V.max == x)
3.              return true;
4.          else if (V.u == 2)
5.              return false;
6.          else
7.              return contain(V.cluster[high(x)],low(x));
8.      }
```

Line 2 to 5 refer to the base cases of this method. Line 2 to 3 refer to the case where the element is found, and line 4 to 5 refer to the case where one of the leaves of the tree is reached and the element is not found.

Suppose a function $T_{(u)}$ that calculates the running time requirement, we have the following equation for $T_{(u)}$ due to the nature of recursion:

$$T_{(u)} = T_{(\sqrt{u})} + O(1)$$

Substituting u by $\mathrm{u} = 2^n$, we have

$$T_{(2^n)} = T_{(2^{\frac{n}{2}})} + O(1)$$

Now if we define a new function $F_{(k)}$ such that $F_{(n)} = T_{(2^n)}$, we obtain

$$F_{(n)} = F\left(\frac{n}{2}\right) + \mathrm{O}(1)$$

Because the size halves for each recursive step, we have

$$F_{(n)} \in O(\log n)$$

Substituting back to $T_{(u)}$, we have

$$T_{(u)} = T_{(2^n)} = F_{(n)} \in O(\log n)$$

Because $\mathrm{n} = \log_2 u$, so $T_{(u)} \in O(\log(\log_2 u))$

And Thus, we obtain the running time requirement for this method

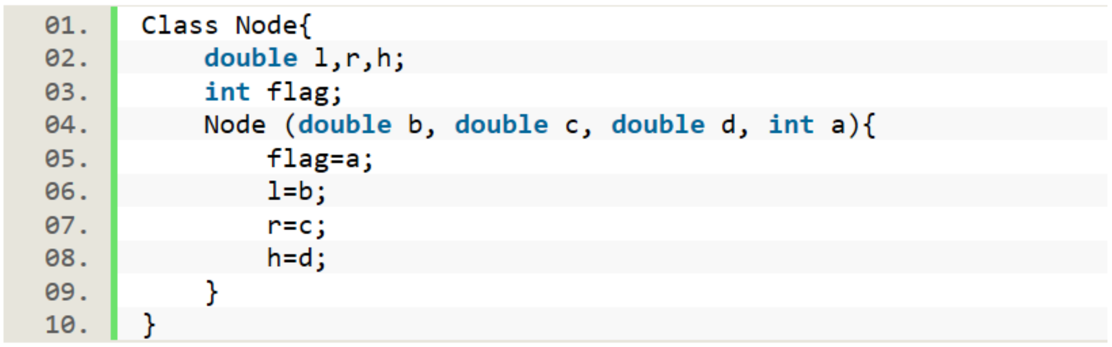
$$T_{(u)} \in O(\log\log u)$$

Similarly, other methods also have run time requirement $\mathrm{O}(\log\log u)$, because all recursive methods are run on a universe size of $\sqrt{u}$.

III. Realizing Scan-line Filling Algorithm by Segment Tree and VEB Tree

1. Realizing Scan-line Filling Algorithm by Segment tree

The following Classes are realizations of scan-line filling algorithm using segment tree. Class Node is the structure of scan-lines. Class ScanLineFillingSegmentTree is the body of such algorithm.



```
Class Node{
    double l,r,h;
    int flag;
    Node (double b, double c, double d, int a){
        flag=a;
        l=b;
        r=c;
        h=d;
    }
}
```

Class Node, as the structure for Scan-line, contains all necessary variables explained in *Section II.1*.

```
Class ScanLineFillingSegmentTree {
    final int MAXN=256;
    double[] X=new double[MAXN];
    LinkedList<Node> nodes=new LinkedList<Node>();
    int[] cnt=new int[MAXN*4];
    double[] sum=new double[MAXN*4];
    int[] source=new int[1024];
    int numberOfMaps;
    ScanLineFillingSegmentTree(int[] list, int a)
    {
        source=list;
        numberOfMaps=a;
    }
    void PushDown(int i,int l,int r)
    {
        int m=(l+r)/2;
        if(cnt[i]!=-1)
        {
            cnt[i*2]=cnt[i*2+1]=cnt[i];
            sum[i*2]=(cnt[i]!=0? (X[m+1]-X[l]):0);
            sum[i*2+1]=(cnt[i]!=0? (X[r+1]-X[m+1]):0);
        }
    }
    void PushUp(int i,int l,int r)
    {
        if(cnt[i*2]==-1 || cnt[i*2+1]==-1)
            {cnt[i]=-1;}
        else if(cnt[i*2] != cnt[i*2+1])
            {cnt[i]=-1;}
        else
            {cnt[i]=cnt[i*2];}
        sum[i]=sum[i*2]+sum[i*2+1];
    }
    void build(int i,int l,int r)
    {
        if(l==r)
        {
            cnt[i]=0;
            sum[i]=0.0;
            PushUp(i,l,r);
            return ;
        }
        int m=(l+r)/2;
        build(i*2,l,m);
        build(i*2+1,m+1,r);
        PushUp(i,l,r);
    }
    void update(int ql,int qr,int v,int i,int l,int r)
    {
        if(ql<=l && r<=qr)
        {
            if(cnt[i]!=-1)
            {
                cnt[i]+=v;
                sum[i]=(cnt[i]!=0? (X[r+1]-X[l]):0);
                return;
            }
        }
        PushDown(i,l,r);
        int m=(l+r)/2;
        if(ql<=m){update(ql,qr,v,i*2,l,m);}
        if(m<qr){update(ql,qr,v,i*2+1,m+1,r);}
        PushUp(i,l,r);
    }
    int binarySearch(double key,int n,double d[])
    {
        int l=1;
        int r=n;
        while(r>=l)
        {
            int m=(r+l)/2;
            if(d[m]==key)
                return m;
            else if(d[m]>key)
                r=m-1;
            else
                l=m+1;
        }
        return -1;
    }
81.      void main()
82.      {
83.          if(source[0]!=0)
84.          {
85.              int n=0;
86.              for(int i=0;i<numberOfMaps;i++)
87.              {
88.                  double x1,y1,x2,y2;
89.                  if(source[4*i+1]<source[4*i+3]){
90.                      x1=source[4*i];
91.                      y1=source[4*i+1];
92.                      x2=source[4*i+2];
93.                      y2=source[4*i+3];}
94.              else{
95.                      x2=source[4*i];
96.                      y2=source[4*i+1];
97.                      x1=source[4*i+2];
98.                      y1=source[4*i+3];
99.              }
100.                 X[n++]=x1;
101.                 nodes.add(new Node(x1,x2,y1,1));
102.                 X[n++]=x2;
103.                 nodes.add(new Node(x1,x2,y2,-1));
104.             }
105.             sort(X);
106.             sort(nodes,h);
107.             int k=1;
108.             for(int i=1;i<n;i++)
109.                 if(X[i]!=X[i-1]) X[k++]=X[i];
110.             build(1,1,k-1);
111.             double ret=0.0;
112.             for(int i=0;i<n-1;i++)
113.             {
114.                 int l=binarySearch(nodes.get(i).l,k,X)+1;
115.                 int r=binarySearch(nodes.get(i).r,k,X);
116.                 if(l<=r){update(l,r,1,1,k-1);}
117.                 System.out.print(sum[1]+" ");System.out.println(nodes.get(i+1).h-nodes.get(i).h);
118.                 ret += sum[1]*(nodes.get(i+1).h-nodes.get(i).h);
119.             }
120.             System.out.println("area:"+ret);
121.         }
122.     }
123. }
```

In Class ScanLineFillingSegmentTree, line 2-8 contains all variables and arrays required for such algorithm. MAXN is a constant that represents the universe size of the coordinate system that is used to calculate areas. X[] is an array that stores the values of x_1 and x_2 of all scan-lines. List Nodes stores all scanlines. Instead of tree[], the segment tree here maintains cnt[] and sum[]. As explained in *Section II.1*, cnt[i] indicates whether the x-coordinate i should be calculates, and sum[] of a tree node stores the total length of segments that should be calculated in its children. Source[] and numOfMaps are inputs of this algorithm.

Line 14-64 is the methods of segment tree, which allow us to update the segment tree when scan-lines are inputted. Note that the “pushup” and “pushdown” procedures are created to help methods build() and update() to updates parent or child nodes conveniently. Line 81-122 is the main method of this algorithm. Line 86-104 read in inputs and store them into X[] and list nodes. Line 105-106 sorts X[] and list nodes in an increasing order. Line 107-109 delete repeating elements in X[]. Line 110 builds an empty segment tree. Integer ret, which is the final result, is created in line 111. Line 112 creates a loop that traverse through all scan-lines in list Nodes. Line 114-116 update x_1 through x_2 of a scan-line into the segment tree. Line 118 calculates the area. Sum[1], which is the sum element of the root of the segment tree that equals the total length of the x-coordinates that should be calculated as $\Delta \mathrm{x}$, and $\mathrm{nodes.get(i+1).h} - \mathrm{nodes.get(i).h} = \Delta \mathrm{h}$.

2. Realizing Scan-line Filling Algorithm by VEB tree

The following Classes realize scan-line filling algorithm by VEB tree. Class Scanline is same as the class Node in *Section III.1*. Class Node here is the structure of VEB tree node. Class ScanlineFillingVebTree is the body of this realization.

```
Class Scanline {
    double l,r,h;
    int flag;
    Scanline (double b, double c, double d, int a){
        flag=a;
        l=b;
        r=c;
        h=d;
    }
}
```

```
Class Node {
    int u;
    Integer min=null;
    Integer max=null;
    Node[] cluster;
    Node summary;
    Node(int u) {
        this.u = u;
        if (u != 2) {
            double pow = Math.log(u) / Math.log(2);
                    int high = (int) Math.pow(2, (int) (Math.ceil(pow / 2)));
                    int low = (int) Math.pow(2, (int) (Math.floor(pow / 2)));
                    this.summary = new Node(high);
                    this.cluster = new Node[low];
                    for (int i = 0; i < low; i++){this.cluster[i] = new Node(low);}
        }
    }
    int high(int x){
        return ((int)(x/Math.sqrt(u)));
    }
    int low(int x){
        return x % (int)Math.sqrt(u);
    }
}
```

```
Class ScanlineFillingVebTree {
    final int MAXN=256;
    double[] X=new double[MAXN];
    double[] Y=new double[2048];
    Node root=new Node(MAXN);
    int[] source;
    int numberOfMaps;
    LinkedList<Scanline> lines=new LinkedList<Scanline>();
    ScanlineFillingVebTree(int a, int[] b){
        numberOfMaps=a;
        source=b;
    }
    void emptyInsert(Node V,int x){
        V.max=x;
        V.min=x;
    }
    void insert(Node V, int x) {
        if (V.min == null) {
            emptyInsert(V,x);
        } else {
            if (x < V.min) {
                int temp = x;
                x=V.min;
                V.min=temp;
            }
            if (V.u > 2) {
                int highx = x / V.cluster.length;
                        int lowx = x % V.cluster.length;
                if (V.cluster[highx].min == null) {
                    emptyInsert(V.cluster[highx],lowx);
                    insert(V.summary,highx);
                } else
                    insert(V.cluster[highx],lowx);
            }
            if (x > V.max){V.max = x;}
        }
    }
38.     public void delete(Node V, int x) {
39.         if (V.min == V.max){V.min = null;V.max = null;}
40.             else if (V.u == 2) {
41.             if (x == 0){V.min = 1;}
42.                 else {V.min = 0;}
43.                 V.max = V.min;
44.             } else {
45.             if (x == V.min) {
46.                     Integer firstCluster = minimum(V.summary);
47.                         if(firstCluster!=null&&minimum(V.cluster[firstCluster])!=null){
48.                         x = firstCluster * V.cluster.length + minimum(V.cluster[firstCluster]);
49.                         V.min = x;}
50.                 }
51.                 delete(V.cluster[V.high(x)],V.low(x));
52.                 if (minimum(V.cluster[V.high(x)]) == null) {
53.                         delete(V.summary,V.high(x));
54.                         if (x == V.max) {
55.                             Integer summary_max = maximum(V.summary);
56.                                 if (summary_max == null)
57.                                     V.max = V.min;
58.                                 else if (summary_max!=null && maximum(V.cluster[summary_max])!=null)
59.                                     V.max = summary_max * V.cluster.length + maximum(V.cluster[summary_max]);
60.                         }
61.                     } else if (x == V.max) {
62.                     V.max = V.high(x) * V.cluster.length + maximum(V.cluster[V.high(x)]);
63.                 }
64.             }
65.     }
66.
67.     Integer minimum(Node V){
68.         return V.min;
69.     }
70.
71.     Integer maximum(Node V){
72.         return V.max;
73.     }
74.
75.     boolean contain(Node V, int x) {
76.             if (V.min!=null&&V.max!=null&&(V.min == x || V.max == x))
77.                     return true;
78.             else if (V.u == 2)
79.                     return false;
80.             else
81.                     return contain(V.cluster[V.high(x)],V.low(x));
82.     }
83.     void main()
84.     {
85.         if(source[0]!=0){
86.             for(int i=0;i<numberOfMaps;i++){
87.                 double x1,y1,x2,y2;
88.                 if(source[4*i+1]<source[4*i+3]){
89.                     x1=source[4*i];
90.                     y1=source[4*i+1];
91.                     x2=source[4*i+2];
92.                     y2=source[4*i+3];
93.                 }
94.                 else{
95.                     x2=source[4*i];
96.                     y2=source[4*i+1];
97.                     x1=source[4*i+2];
98.                     y1=source[4*i+3];
99.                 }
100.                lines.add(new Scanline(x1,x2,y1,1));
101.                lines.add(new Scanline(x1,x2,y2,-1));
102.            }
103.            sort(lines,h);
104.            double ret=0.0;
105.            int sum = 0;
106.            for(int i=0;i<lines.size()-1;i++){
107.                if(lines.get(i).flag==1){
108.                    for(int a=(int)lines.get(i).l;a<=(int)lines.get(i).r;a++){insert(root,a);}
109.                }
110.                if(lines.get(i).flag==-1){
111.                    for(int c=(int)lines.get(i).l;c<=(int)lines.get(i).r;c++){delete(root,c);}
112.                }
113.                if(maximum(root)!=null&&minimum(root)!=null){
114.                    sum=0;
115.                    for(int j=0;j<MAXN;j++){if(contain(root,j)){sum++;}}
116.                    sum--;
117.                    ret+=sum*lines.get(i+1).h-lines.get(i).h;
118.                }
119.            }
120.            System.out.println("area:"+ret);
121.        }
122.    }
123. }
```

In the body of scan-line filling algorithm above, line 2-8 creates all variables and arrays needed, and line 13-82 are methods of VEB tree.

In the main method of this algorithm, the purpose of line 85-104 has the same purpose as the scan-line filling algorithm realized by segment tree, which is to read in input and sort data. Line 105 create a sum variable, which have the same purpose as the sum[1] in segment tree: to return the value of Δx. While considering the set cnt introduced in *Section II.1*, the method contain() serves as the same purpose, which stores the x-coordinates that should be calculated. Line 106-112 updates the VEB tree according to each scan-line. If the flag value of a scan-line is 1, we insert all coordinates from x_1 to x_2 into the tree. Else, we delete the coordinates. Line 114-116 update the value of sum according to the x-coordinates that are stored in the tree, and line 117 calculate the result.

IV. Proofs and Comparisons between the Theoretical Run Time Requirements of the Two Realizations

1. Scan-line Filling Realized by Segment Tree

Suppose $T_1(u)$ is the run time requirement of segment tree methods of universe size u. $T_2(u)$ is the run time requirement of scan-line filling algorithm realized by segment tree with universe size u. According to *Section II.3*, where we proved the run time requirement of segment trees, we have

$$T_1(u)\epsilon O(\log u)$$

According to *Section III.1*, where we realized scan-line filling algorithm by segment tree, we can see that update() is run once for each scan-line. Therefore, we have

$$T_2(u) = nT_1(u)$$

Where n is the total number of scan-lines. Thus, we have

$$T_2(u) \in O(u \log u)$$

as its run time requirement.

2. Scan-line Filling Realized by VEB Tree

Suppose $T_3(u)$ is the run time requirement of VEB tree methods of universe size u. $T_4(u)$ is the run time requirement of scan-line filling algorithm realized by VEB tree with universe size u. We have proved the run time requirement of VEB tree methods:

$$T_3(u)\epsilon O(\log \log u)$$

According to *Section III.2*, where we realized scan-line filling algorithm by VEB tree, we can see that for each scan-line, either method insert() or method delete() is run for at most u times, where u is the universe size. After VEB tree is updated, another loop is traversed to maintain element sum. So,

$$T_4(u) = n\big(uT_3(u) + uT_3(u)\big) = 2nuT_3(u)$$

Thus, we have

$$T_4(u) \in O(u^2 \log\log u)$$

as its run time requirement.

3. Comparison

From the section above, we have the run time requirement of both realizations. Scan-line filling realized by segment tree has run time requirement $O(u \log u)$, whereas the one realized by VEB tree has run time requirement $O(u^2 \log\log u)$.

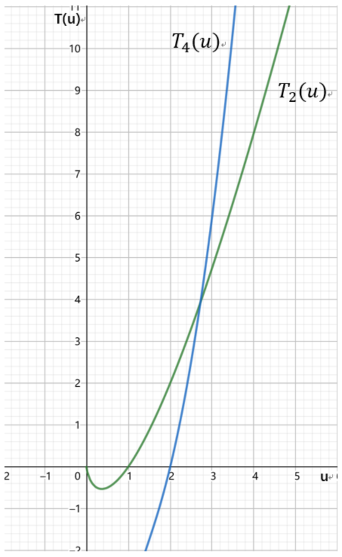


Figure 12: Comparison of Theoretical Run Time Efficiency

Figure 12 shows the functions of run time requirement of the two realizations. The blue curve represents $T_4(u)$, and the green curve represents $T_2(u)$. From the diagram, we can see that the two functions intersect at point (2.72, 3.93). As we can see, theoretically, VEB tree is faster when the universe size is small enough. If the universe size is large, segment tree is more efficient instead.

V. Testing Real Time Performances of Two Realizations

1. Testing Environment

The experiments are conducted under Java environment JRE 1.8.0_161. Data is processed by a 2-core CPU with processing speed 2.4GHz.

2. Testing Methods and Chosen Input

As an important algorithm that processes images, scan-line filling algorithm should be able to deal with different kinds of images efficiently. For the two realizations, we use randomly generated data to measure their normal run time requirement, as most images are random and different. We change the universe size and the number of regions to analyze how run times change with images of different complexity. After testing normal situations, we test two special situations. Firstly, we test the performance of both realizations with “neat input”: small regions that are not overlapping with each other, such as Figure 13.

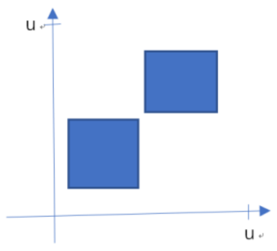


Figure 13: A Simple Example of “Neat Input”

Secondly, we test the performance of both realizations with “extreme input”: input regions covering the whole universe size, such as Figure 14.

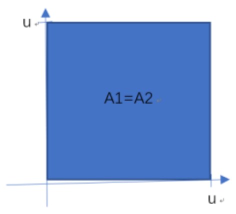


Figure 14: A Simple Example of "Extreme Input"

The chosen two special inputs will test how the two algorithms perform under special circumstances, such as an abstract painting with regular geometry shapes, or a black frame during a movie.

3. Results and Comparisons:

i. Random Inputs:

(1) Increasing Universe Size

We test the run time of both realizations with increasing universe size. Because of the structure of VEB tree, which recurse with size $\sqrt{u}$, the universe size is chosen as 16, 256, 65536 and 4294967296 to avoid error. The number of region is kept at 5. Run time is measured in unit nanosecond. Because of the systematic error caused by randomly generated data and the random error of Java, 5 trials are conducted at each universe size. Average run times are then calculated for more accurate results.

The following table shows the experiment results of scan-line filling realized by VEB tree.

u	Run Time (ns)				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
16	9047784	8868091	9907680	7297633	14601008
256	14619879	18863567	19718130	22177203	15554032
65536	156101142	129969891	149082073	100753517	183936184
4294967296	Stack Overflow				

Table 1: Run Time of Scan-line Filling Realized by VEB tree with different universe size (raw data)

After calculating the average run times, we have

u	Run Time (ns)
16	9944439.2
256	18186562.2
65536	143968561.4

Table 2: Relationship between VEB Tree Universe Size and Run Time (average)

The following table shows the experiment results of scan-line filling algorithm realized by segment tree.

u	Run Time (ns)				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
16	11224192	9375988	9604911	6477120	14594444
256	12500498	14153009	18281003	10663371	20474232
65536	Stack Overflow				

Table 3: Run Time of Scan-line Filling Realized by Segment Tree with Different Universe Size (raw data)

After calculating the average, we have

u	Run Time (ns)
16	10255331
256	15214422.6

Table 4: Relationship between Segment Tree Universe Size and Run Time (average)

Because universe size of segment tree is not limited with relationship $\sqrt{u}$, we take more trials, and results are shown in Table 5.

u	Run Time (ns)
16	10255331
256	15214422.6
512	18244818.6
1024	15918913.6
2048	16775692

Table 5: Relationship between Segment Tree Universe Size and Run Time

Plotting them in a coordinate system, we have the following functions where the orange curve represents VEB tree and the blue curve represents segment tree.

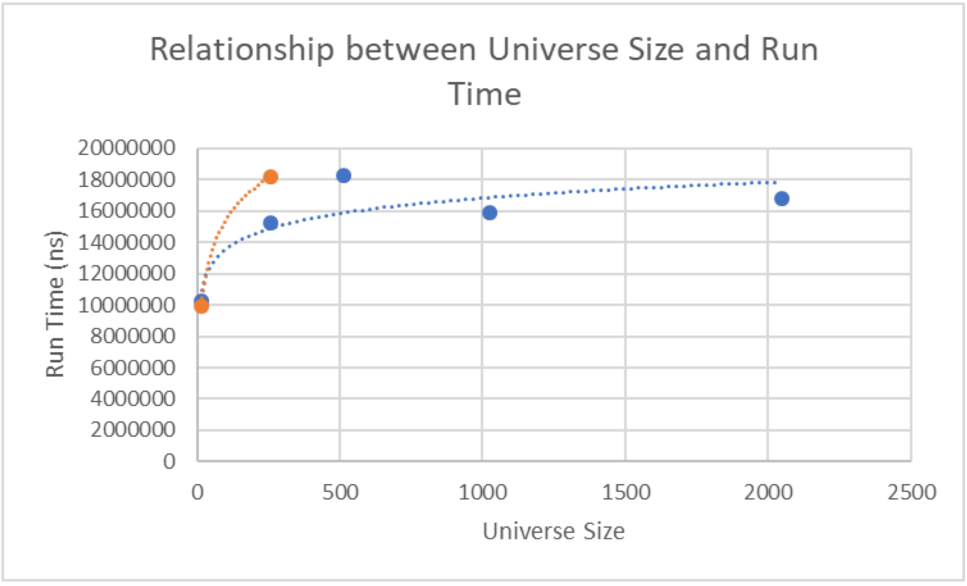


Figure 15: Relationship between Universe Size and Run Time for both Realizations (plotted)

As we can see from the plotted data and the trend lines that follow logarithmic relationship from Figure 15, we can see that when universe size is small, the run time of VEB tree is comparingly faster, though at such level of universe size, the difference in run time, which is less than 1ms, is insignificant. However, when universe size increases, the run time of VEB tree increases faster than the run time of segment tree, which makes VEB tree inefficient when universe size becomes large. Surprisingly, as VEB tree calls recursive methods on size $\sqrt{u}$ and segment tree calls recursive methods on size $\frac{1}{2}$u, VEB tree is less likely to throw StackOverFlow exception and is more practical when universe size is too large.

(2) Increasing Number of Regions

Now, we test the run time of the two realizations with different number of regions processed. The universe size is set at a constant of 256. The following table shows the experiment results of scan-line filling realized by VEB tree.

n	Run Time (ns)				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
10	39080568	35416982	27496171	19528181	18527977
20	28285503	33530625	27529812	35613905	49002196
30	74469241	48391735	32653498	36095956	32120986
40	85414048	34254317	54870907	42668666	45054304
50	50477066	53455934	60243208	60116028	51105168

Table 6: Run Times of Scan-line Filling Realized by VEB Tree with Different Number of Regions (raw data)

After calculating the average run times, we have

n	Run Times(ns)
10	28009975.8
20	34792408.2
30	44746283.2
40	52452448.4
50	55079480.8

Table 7: Relationship between Number of Regions and Run Time

We repeat the same experiment process with segment tree. The following table shows the experiment results of scan-line filling realized by segment tree.

n	Run Time (ns)				
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
10	10527577	18219054	24436893	8974348	18914849
20	18782336	31403038	26438122	19582745	29165502
30	24597713	37598313	38031954	29314015	36640776
40	27757094	41003847	41234001	42645281	61724642
50	37210621	46554609	46529994	27937196	39234823

Table 8: Run Times of Scan-line Filling Realized by Segment Tree with Different Number of Regions (raw data)

After calculating the average run times, we have

n	Run Time (ns)
10	16214544.2
20	25074348.6
30	33236554.2
40	42872973
50	39493448.6

Table 9: Relationship between Number of Regions and Run Time

Plotting them in a coordinate system, we have the following functions where the orange curve represents the relationship between universe sizes and run times of VEB tree, and the blue curve represents segment tree.

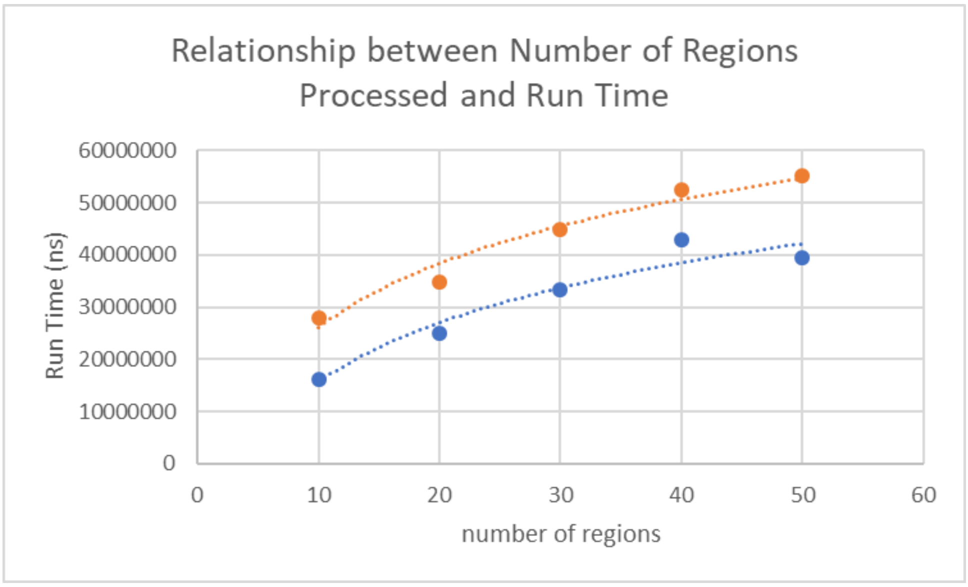


Figure 16: Relationship between Universe Size and Run Time for both Realizations (plotted)

According to the functions and the trend lines above, we can see that the orange curve is above the blue curve, meaning that for most cases regarding the number of regions processed, segment tree has a higher run time efficiency than VEB tree.

ii. Neat Input

For this special case, we generate several regions that are not intersected.

Universe size is kept at a constant value of 256.

The following table shows the experiment results.

Data Structure	Number of Regions	Run Time (ns)			
		Trial 1	Trial 2	Trial 3	Average Result
VEB Tree	5	14722854	15228700	15855981	15269178.33
Segment Tree	5	11688191	7503171	11473627	10221663
VEB Tree	10	23853099	21323871	14356495	19844488.33
Segment Tree	10	36679750	19139258	20638334	25485780.67

Table 10: Experiment Results of "Neat Input"

As we can see, in the first two trials, 5 small regions are processed, and then, 10 small regions are processed. When number of regions equals 5, segment tree processes faster, and when number of regions equals 10, VEB tree processes faster. However, the difference in processing time, which is about 5ms, is not significant. From the data, we can see that VEB tree is efficient when processing small regions at a large number. Considering the realization of VEB tree, when regions are small, less insert() and delete() methods will be called, which shorten run times significantly.

iii. Extreme Input

For this special case, we generate 2 regions that cover the whole universe.

Number of regions is kept at a constant value of 2.

The following table shows the experiment results.

Data Structure	Universe Size	Run Time (ns)			
		Trial 1	Trial 2	Trial 3	Average Result
VEB Tree	256	20493103	19090028	19000182	19527771
Segment Tree	256	5571275	5083481	5024815	5226523.667
VEB Tree	16	8032811	5965531	6183787	6727376.333
Segment Tree	16	4943173	4473020	5089635	4835276

Table 11: Experiment Result of "Extreme Input"

As we can see, when universe size is small, difference in run time is not significant. However, when universe sizes increase, at the same time when the areas of regions increase, the processing time of VEB Tree becomes inefficient, with a difference of about 15ms in run times. VEB tree is less efficient and less useful when processing large regions with large areas. The realization of scan-line filling by VEB tree can explain this phenomenon. When large regions are processed, methods insert() and delete() are processed for more times, which significantly increase processing times. Also, considering Figure 12, theoretically, the function that represents run time of VEB tree has larger slope, which indicates that the run time of VEB tree increases more significantly with the increase in universe size.

We have explained why processing speed of VEB tree can fluctuate greatly with changes in universe sizes and region sizes. On the other hand, according to the experiment results and Figure 12, performance of segment tree is comparably more stable, because segment tree manages and maintains one and only one “segment” in method update() regardless of sizes for each inputted scan-line. So, run times are not significantly affected by special cases or properties of graphs.

VI. Conclusion

Truly, VEB tree is a highly efficient data structure comparing to segment tree. Through the recursive calls upon universe size of $\sqrt{u}$, this essay successfully proves that operations of VEB tree have run time requirement $O(\log\log n)$.

Because of the two loops called upon VEB tree methods when realizing scan-line filling algorithm, this essay proves that such realization has run time requirement $O(n^2 \log\log n)$, which is theoretically more efficient than $O(\mathrm{n} \log n)$ when universe size is small but less efficient if universe size is large.

In real-life practices, the advantage of VEB tree when universe size is small is not significant, because when universe size is small, both realizations already have very fast run times. When universe size gets larger, the disadvantage of VEB is enlarged. However, using VEB tree data structure solved the problem of StackOverflow that segment tree previously had. While the problem of stack overflow limits the universe size segment tree can maintain, such exception can be solved by using VEB tree data structure, which makes VEB tree a more practical realization when universe size is large.

Considering the natures of images processed by the algorithms, and according to the testing results of neat inputs and extreme inputs, we can see that VEB tree is more suitable to process small regions at a large number, while it is inefficient when processing large regions. The performance of segment tree, on the other hand, is more stable with different image complexity and different situations.

VII. Limitation and Evaluation

Because of the limitation of the use of equipment and tools, only personal laptops can be used during experiments. Therefore, large random errors occurred as JRE 1.8.0 presents uncertainties when system times are measured in a small unit of nanosecond. The fluctuations of CPU occupations of background software, system software and the operating system that is hard to control manually can lead to systematic errors. Though the accuracies and the precisions of experiment results can be improved with more powerful computers or environments that are specifically designed for testing, current experiment results are already presented clearly and resulted in clear conclusions.

VIII. Bibliography

1. "Understanding Segment Tree Step by Step." *Understanding Segment Tree Step by Step*, Cnblogs, 1 Dec. 2013, 22:30, www.cnblogs.com/TenosDoIt/p/3453089.html.
2. "Scanline Rendering." Wikipedia, Wikimedia Foundation, 28 June 2017, en.wikipedia.org/wiki/Scanline_rendering#Use_in_realtime_rendering.
3. Cormen, Thomas H., et al. *Introduction to Algorithms*. China Machine Press, 2016.
4. "Segment Tree Summary." *Cnblogs*, Cnblogs, 19 Aug. 2017, 15:17, www.cnblogs.com/yiyiyizqy/p/7396693.html.
5. Zhang, Kunwei. *The Power of Statistics*. Tsinghua University, 25 July 2013, wenku.baidu.com/view/5c6ca2dcce2f0066f5332277.html.
6. "Interval Tree and Segment Tree." *Interval Tree and Segment Tree*, CSDN, 21 Aug. 2014, 15:04, blog.csdn.net/yuzhiyuxia/article/details/38728997.
7. "Flexible Uses of Data Structures." *Flexible Uses of Data Structure*, CSDN, 29 Sept. 2015, 20:17, blog.csdn.net/zhoufenqin/article/details/47809551.
8. "HDU1542 Atlantis." *HDU1542 Atlantis*, CSDN, 17 Aug. 2015, 17:49, blog.csdn.net/mengxingyuanlove/article/details/47728405.
9. "Poligon Area Filling (Sorted Edge Table)." *Poligon Area Filling (Sorted Edge Table))*, CSDN, 19 Mar. 2012, 14:57, blog.csdn.net/orbit/article/details/7368996.
10. "Detailed Explanation of Segment Tree." *Detailed Explanation of Segment Tree*,

360doc, 26 July 2017,

www.360doc.com/content/17/0726/10/45548709_674210643.shtml.

11. “HDU 1255 Covered Area.” *HDU 1255 Covered Area*, CSDN, 14 Aug. 2015,

10:33, blog.csdn.net/qq_18661257/article/details/47658101.

IX. Appendix

1. Java Implementation of VEB Node Structure

Nodes of VEB structures includes indications of the information of their subtrees and pointers towards other VEB structures. The following Class represents a realization of a VEB structure:

```
1.  private static class Node {
2.      private int u;
3.      private Integer min;
4.      private Integer max;
5.      private Node[] cluster;
6.      private Node summary;
7.  
8.      public Node(int u) {
9.          this.u = u;
10.         if (u != 2) {
11.             double pow = Math.log(u) / Math.log(2);
12.             int ceil = (int) Math.pow(2, (int) (Math.ceil(pow / 2)));
13.             int floor = (int) Math.pow(2, (int) (Math.floor(pow / 2)));
14.             this.summary = new Node(ceil);
15.             this.cluster = new Node[floor];
16.             for (int i = 0; i < low; i++)
17.                 this.cluster[i] = new Node(floor);
18.         }
19.     }
20. }
```

In this code segment, line 2 to 6 defined the variables that are necessary in the VEB structure, including u, min, max, summary and cluster, as explained in the previous chapter.

Line 8 to 18 is the constructor of the Node class, which gives the variables defined proper values and addresses.

2. Java Implementation of VEB Tree Methods

i. Minimum and Maximum

As the values of the minimum and maximum data are directly indicated in the nodes, these values can be easily returned.

```java
public int minimum(Node V) {
        return V.min;
    }

public int maximum(Node V) {
        return V.max;
    }
```

Clearly, these methods have runtime efficiency O(1).

ii. Contain

To search for an element, a recursive structure should be implemented in order to search through each layers of the trees.

```java
1.  public boolean contain(Node V, int x) {
2.          if (V.min == x || V.max == x)
3.              return true;
4.          else if (V.u == 2)
5.              return false;
6.          else
7.              return contain(V.cluster[high(x)],low(x));
8.      }
```

Line 2 to 5 refer to the base cases of this method. Line 2 to 3 refer to the case where the element is found, and line 4 to 5 refer to the case where one of the leaves of the tree is reached and the element is not found.

iii. Insert

To insert, we not only need to recursively reach the leaves of the trees, but also update the summary nodes while inserting. The following methods represent insert().

```
1.  public void insert(Node V, int x) {
2.      if (V.min == null) {
3.          emptyInsert(V,x);
4.      } else {
5.          if (x < V.min) {
6.              int temp = V.min;
7.              V.min = x;
8.              insert(V,temp);
9.          }
10.         if (u > 2) {
11.             if (V.cluster[high(x)].min == null) {
12.                 emptyInsert(V.cluster[high(x)],low(x));
13.                 insert(V.summary,high(x));
14.             } else
15.                 insert(V.cluster[high(x)],low(x));
16.         }
17.         if (x > max)
18.             V.max = x;
19.     }
20. }
```

```
1.  public void emptyInsert(Node V,int x){
2.      V.max=x;
3.      V.min=x;
4.  }
```

Line 2-3 manage a simple base case, where a node is empty. In this case, we only need to insert the element into it by updating the min and max values. Line 5-8 manage the situation in which x is smaller than the original min. In this case, x becomes the new min, and the original min should be inserted into the tree instead. So, line 5-8 swap x with min. Line 10-18 manage the recursive situation when leaves are not reached. Line 11-13 is run if the cluster that x belongs to is empty. In this case, we can easily insert x into its empty cluster and update summary node. If the cluster that x belongs to is not empty, meaning that the base case is not yet reached, we should call the recursive method.

The running time requirement for insert is $O(\log\log u)$, because either the recursive method in line 13 or line 15 will be run, and both methods run on a universe size of $\sqrt{u}$.

iv. Delete

The following method will delete element x from node V. Here, we assume that element x is stored in V.

```
1.  public void delete(Node V, int x) {
2.        if (V.min == V.max) {
3.          V.min = null;
4.          V.max = null;}
5.        else if (V.u == 2) {
6.          if (x == 0)
7.            V.min = 1;
8.          else
9.            V.min = 0;
10.         V.max = V.min;
11.       } else {
12.         if (x == V.min) {
13.           int firstCluster = minimum(V.summary);
14.           x = firstCluster * V.cluster.length + minimum(V.cluster[firstCluster]);
15.           V.min = x;
16.         }
17.         delete(V.cluster[high(x)],low(x));
18.         if (minimum(V.cluster[high(x)]) == null) {
19.           delete(V.summary,high(x));
20.           if (x == V.max) {
21.             int summary_max = maximum(V.summary);
22.             if (summary_max == null)
23.               V.max = V.min;
24.             else
25.               V.max = summary_max * V.cluster.length + maximum(V.cluster[summary_max]);
26.           }
27.         } else if (x == V.max) {
28.           V.max = high(x) * V.cluster.length + maximum(V.cluster[high(x)]);
29.         }
30.       }
31.     }
32.   }
```

Line 2-4 manage the base case where only one element is included in V, where we directly set the max and min indicators to null. Line 5-10 manage another base case. When one of the leaves of the tree is reached, but there is more than one element stored in that leaf, there must be two elements. In this case, we either delete element 0 or element 1 of that leaf, depending on the value of x.

If base case is not reached, meaning that V contains 2 or more than 2 elements, and $u \geq 4$, line 11-29 is run. In this case, we are deleting element x in one of V's cluster. Line 13-16 manage the case where we are deleting the min element in V. When deleting min, we should update min to another value and then delete the original min. So, line 14 create an integer that stores the cluster of the smallest element other than min, and in line 15-16, this element will become the new min.

Line 18 is the recursive method that delete x from its cluster. Line 19-20 update summary node if x's cluster becomes empty after the deletion. Line 21-29 will update element max if deleted element is the maximum of V.

The running time requirement of delete() is still $O(\log\log u)$. From the method above, we can see that there are two recursive methods in line 18 and line 20. Both line 18 and line 20 run on a universe size of $\sqrt{u}$. If both line 18 and 20 are called, the run time requirement will exceed $O(\log\log u)$. However, line 20 is called only if the cluster that x belongs to is empty. So, line 20 is called only when the cluster contains a single element x before calling line 18. In this case, the recursive call of line 18 has running time requirement $O(1)$ due to the base case according to line 2-4. So, there are only 2 cases: line 18 is run without running line 20, or line 20 is run with line 18 having efficiency of $O(1)$. Therefore, the running time requirement of delete() is still $O(\log\log u)$.

3. Java Implementation of Segment Tree Structure

```java
public class Node {
    public int left;
    public int right;
    public int max;
    public int sum;
    public Node(){
        left=right=max=sum=0;
    }
}
```

Integers left and right are the left and right end points of the part of array a node maintains. For example, for root [1,10] in Figure 10, left will be 1 and right will be 10.

4. Java Implementation Segment Tree Methods

i. Build

```java
1.  public void build(int i, int l, int r){
2.      tree[i].left=l;tree[i].right=r;
3.      if(l==r){
4.          tree[i].sum=0;
5.          tree[i].max=0;
6.      }
7.      else{
8.          int mid=(l+r)/2;
9.          build(i*2,l,mid);
10.         build(i*2+1,mid+1,r);
11.         tree[i].sum=tree[i*2].sum+tree[i*2+1].sum;
12.         tree[i].max=((tree[i*2].max>=tree[i*2+1].max)? tree[i*2].max:tree[i*2+1].max);
13.     }
14. }
```

The code segment above is a method that create an empty tree. Lines 3-5 handle the base case of this recursive method. When integer left equals right, one of the leaves of the tree is reached. Because we are creating an empty tree, we give integer max and sum of that leaf the value of 0. Line 8-10 calls build() recursively. Line 9 creates left subtrees, and line 10 creates right subtree. Line 11-12 performs “push-up”: after the recursive calls, as we instantiated all leaves, the “push-up” procedure instantiates parent nodes according to their children.

ii. Update

Suppose the segment tree maintain data of B[] by tree[], the following method update() will update data in B[pos] into val by traversing tree[i].

```
1.  public void update(int i, int pos, int val){
2.      if(tree[i].left==tree[i].right){
3.        tree[i].sum=tree[i].max=val;
4.      }
5.      else{
6.        int mid=(tree[i].left+tree[i].right)/2;
7.        if(pos<=mid){
8.          update(i*2,pos,val);
9.        }
10.       else{
11.         update(i*2+1,pos,val);
12.       }
13.       tree[i].sum=tree[i*2].sum+tree[i*2+1].sum;
14.       tree[i].max=((tree[i*2].max>=tree[i*2+1].max)? tree[i*2].max:tree[i*2+1].max);
15.     }
16. }
```

Line 2-3 manages the base case, in which tree leaves are reached. In this case, we need to update the value of this node into val. Line 6-7 decides whether pos belongs to left subtree or right subtree. After such decision, line 8 and line 11 will call update() recursively to update one of its subtree. Line 13-14 performs “push-up”. (explained in *Section III.4.i*)

Such method has running time requirement of $O(\log n)$. As explained in *Section II.3*, segment tree is a balanced binary tree. Suppose a tree with universe size u, the height of the tree is $log_2 u$. As only one branch of the tree is traversed in update(), the running time shall not exceed $log_2 u$, and therefore its running time requirement is $O(\log u)$.