A report on

Implementation of the van Emde Boas tree with application to Kruskal and comparison with respect to Fibonacci and Binomial Heaps.

Submitted by-
Rajat Kumar Verma & Supriya Priyadarshani
2018201040 2018202009
International Institute of Information Technology, Hyderabad

TABLE OF CONTENTS

Introduction

In this project we are going to investigate the performance of binomial heaps, fibonacci heaps, and Van Emde Boas Tree to implement Kruskal's Minimum Spanning Tree Algorithm in the context of finding the minimum cost spanning trees. The main objective of this project is to understand how vEB tree performs in comparison with the other O(log n) time complexity algorithms.

Van Emde Boas tree

Van Emde Boas tree is a tree data structure which implements an associative array with m-bit integer keys. It performs all operations in O(log log M) time, where M is the maximum number of elements that can be stored in the tree. The M is not the actual number of elements stored in the tree, by which the performance of other tree data-structures is often measured but the maximum capacity of the tree. In this project we are going to investigate. The vEB tree has good space efficiency when it contains a large number of elements. vEB tree performs exponentially better than fibonacci heap and binomial heap.

History

The main interest was to design a data structure that supports the ordered dictionary operations on universe faster than a balanced BST. For serving this purpose, Bit Vectors came into existence. It's an array of bits of length U.

Bit Vectors

Many applications need dynamic set data structure which supports dictionary operations such as -search, insert and delete.To store a dynamic set of values from the universe {0, 1, 2.....u-1}, we maintain an array A {0 .... u-1}. The entry A[x] holds a 1 if the value of x is in the dynamic set, and it holds a 0 otherwise. Although we can perform each of the INSERT, DELETE, and MEMBER operations in O(1) time with a bit vector, but the remaining operations—MINIMUM, MAXIMUM, SUCCESSOR and

PREDECESSOR—each take Θ(u) time in the worst case. So, Bit vectors couldn't perform upto mark.

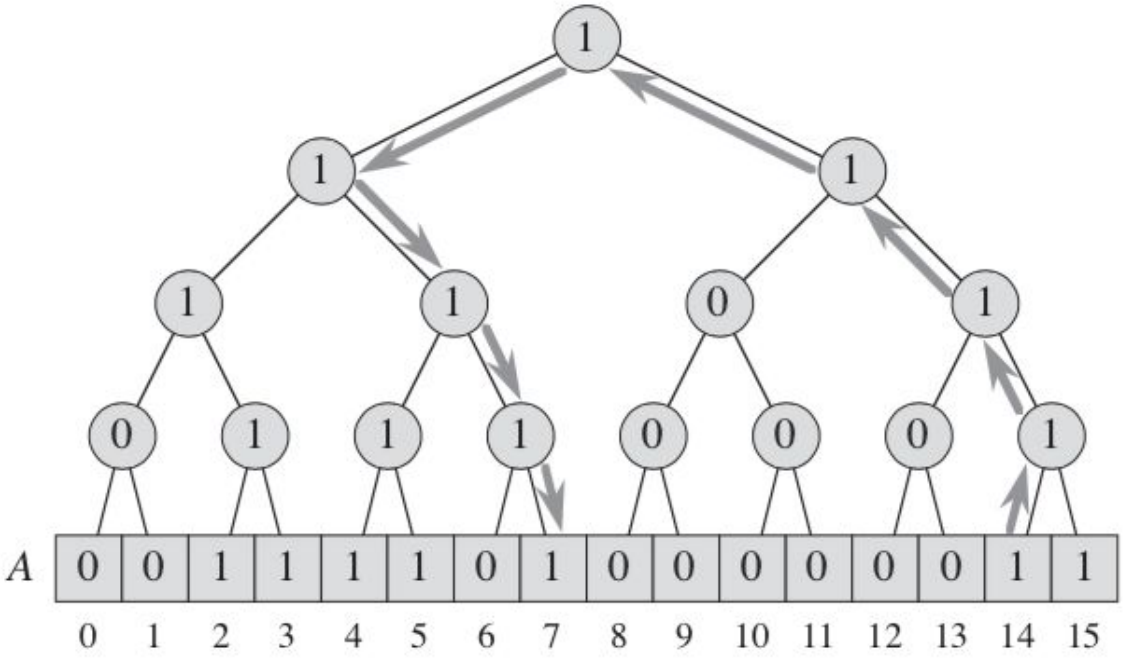


Figure 1.1 A binary tree of bits superimposed on top of a bit vector representing the set {2, 3, 4, 5, 7,14, 15} when u = 16. Each internal node contains a 1 if and only if some leaf in its subtree contains a 1. The arrows show the path followed to determine the predecessor of 14 in the set.

Superimposing a tree on the top of the bit vector

We can cut short the time complexity of finding the predecessor and successor by superimposing a binary tree of bits on top of a bit vector representing the set. Figure 1.1 shows the same. The height of the tree is log n where n is the size of the bit vector. So all the operations specified above will take O(log n) in worst case, including finding predecessor and successor. Search is done in O(1) which is better than other height balanced trees like RB tree. But still, if the data is insufficient as compared to the size of universe, RB tree will perform better in all other conditions.

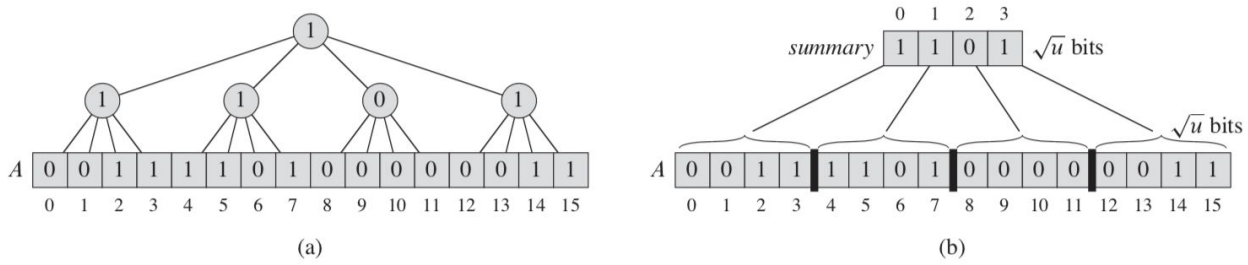


Figure 1.2 (a) A tree of degree √u superimposed on top of the same bit vector as in Figure 1.1 Each internal node stores the logical-r of the bits in its subtree. **(b)** A view of the same structure, but with the internal nodes at depth 1 treated as an array summary[0..√u-1], where summary[i] is the logical-or of the subarray A[i√u..(i+1)√u-1].

Superimposing a tree of constant height with greater degree

We need to check what happens if we superimpose the tree of greater degree. Here a tree of degree √u which leads to the height of tree to be constant always. As before, each internal node stores the logical-or of the bits within its sub-tree, so that the √u internal nodes at depth 1 summarize each group of √u values. Figure 1.2 shows the representation diagram for u = 16 with 4 clusters at depth 1 and a single root node at depth 0. Height of the tree is always 2 in this case i.e. constant.

Using this kind of structure make each of the operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR and DELETE in O(√u). Also to use this method we need to assume u = 2^{2k} ,so that $\sqrt{u}$ is an integer. Then superimpose a tree of degree $\sqrt{u}$ on top of the bit vector . the height of the tree is 2.
In each case we search through at most two clusters plus the summary array, so the time complexity is O($\sqrt{u}$).
This approach was a key idea of van Emde Boas Tree.
From figure one, the idea of recursion hits. Assume that u is 2^{2^k} so that it reduces down to 1 at the root node starting from the leaf level where u size is maximum. Implementing such kind of idea into recursive operation will generate a recursive equation given below.

$$T(n) = T(\sqrt{n}) + O(1)$$

$$T(u) = \Theta(\log \log u)$$

Solving the above equation will give T(u) = O(log log u). Where u is the universe size.
We got the running time as we needed but the story is still not over. The problem with the last structure is, if the size of universe is large and height of the tree is fixed to 2 searching for successor in the tree will take O($\sqrt{n}$) which is not as we needed and the other operations will suffer the same. We know the solution is not as per what we need but we have a correct recursive equation this concludes we are on the right track but not achieved the O(log log n) time complexity for our operations. For this a different structure was designed namely Proto-vEB structure.

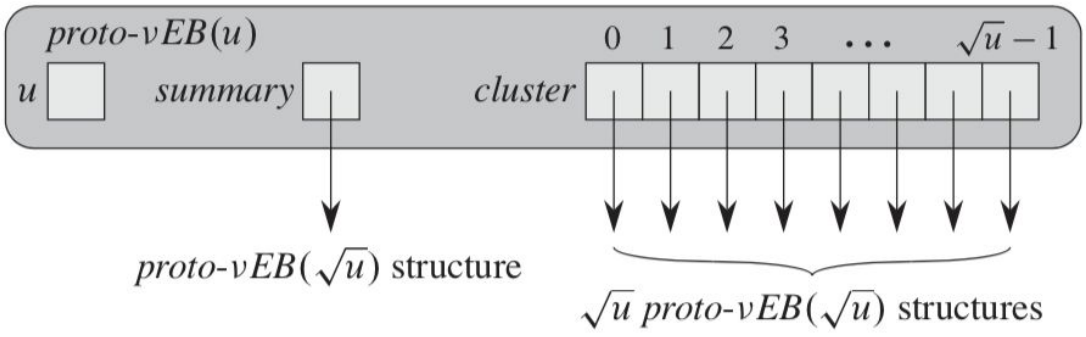


Figure 1.3: The information in a proto-vEB(u) structure when u >= 4. The structure contains the universe size u, a pointer summary to a proto-vEB(√ u) structure, and an array cluster[0 .. √u - 1] of u pointers to proto-vEB (√u) structures.

Proto van Emde Boas structures

The Figure 1.3 shows the Proto van Emde Boas structure. This will not give a time complexity of O(log log u) for all operations but it serves as a basis for the van Emde Boas tree. We define the Proto vEB structure as follows:

- If u = 2, then it is the base size, and it contains an array A[0..1] of two bits.
- Otherwise, u = 2^{2^k} for some integer k >= 1, so that u >= 4. In addition to the universe size u, the data structure proto-vEB(u) contains the following attributes, illustrated in Figure 1.3:
    - a pointer named summary to a proto-vEB(√u) structure and

- an array cluster[0 .. √u-1] of √u pointers, each to a proto-vEB(√u) structure.

The major problem with such a structure was that operations such as finding the minimum, insertion and deletion etc took Θ(log u) time instead of the O(log log u). Later the problem was solved by final vEB-Tree that could store the minimum and maximum at every node. Hence instead of fetching it from recursive call we could directly access it from the respective tree node. Recursive equation was as follows:

$$T(u) = 2T(\sqrt{u}) + O(1)$$

$$T(u) = \Theta(\log u)$$

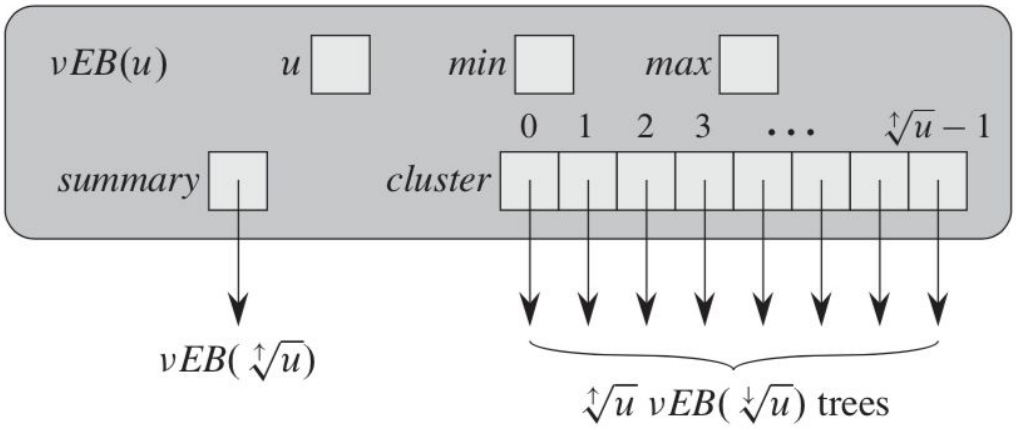


Figure 1.4: The information in a vEB(u) tree when u > 2. The structure contains the universe size u, elements min and max, a pointer summary to a vEB(⌈√u) tree, and an array cluster[0 .. ⌈√ u - 1] of ⌈√u pointers to vEB(⌊√u) trees.

van Emde Boas trees

Finally the van Emde Boas came with modified form of the proto-vEB structure call. The van Emde Boas tree is shown in Figure 1.4 that reduced the number of recursive calls hence reducing the time complexity to O(log log u). This was invented by Peter van Emde Boas in 1975. Data structures that support operations of priority queue like binary heaps, red-black trees, Fibonacci heaps. In each of these data structures, at least one important operation takes O(log n) time. vEB tree supports each of these operations in O(log log n).This was a remarkable

achievement in which the structure is recursively divided and min and the max serve for the actual data present recursively down the tree. At every level of tree min value does not appear in any of the subsequent tree nodes or cluster but the max value does.

Recursive equation for the vEB tree for most of the operation (except find_min or find_max, which is O(1)) came out to be as follows:

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

$$T(u) = \Theta(\log \log u)$$

Binomial Heap

Binomial heap is extension of Binary Heap that provide faster union and merge operation.

- A Binomial heap is a collection of Binomial trees.
- A Binomial Tree of order k can be constructed by taking two binomial trees.
- A Binomial Tree with order k has exactly 2k nodes and it has depth k.
- There are total kCi nodes at depth i.
- The root has degree k ,and the children of root are themselves Binomial trees with order k - 1 , k – 2, ... 0 from left to right
- Binomial heap with n nodes has the number of Binomial Trees equal to the number of set bits in the binary representation of n. Also the position of set bits in the binary representation tells the degree of all the nodes in the root list of the binomial heap. With this relation, **we can conclude that there are O(log n) Binomial Trees in a Binomial Heap with 'n' nodes, which makes all the operations take O(log n) in worst case.**

Fibonacci Heap

In terms of time complexity, fibonacci heap beats both binomial and binary heap because of its amortized cost. Below are some important properties about fibonacci heap:

- Maintains a pointer to the minimum value.

- All roots are connected using circular doubly linked linked which was a singly linked list in case of binomial heap.
- The main idea in fibonacci heap is to operate in lazy way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. But the extract operation iato has to do more work & so is more complicated. And hence delete operation will also be more complicated.
- In practical, Fibonacci heap is found little slower than expected because of it's hidden high constants. We will observe it here.

Comparison

Fibonacci heap vs binomial heap

Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a lazy manner, postponing the work for later operations.
The comparison will be based on the delete, insert and extract_min operations. So we will see the difference in the working and time complexities of the 3 algorithms.
Each tree in a Fibonacci heap is allowed to lose at most two children before that tree needs to be "reprocessed" at a later step. The marking step in the Fibonacci heap allows the data structure to count how many children have been lost so far. An unmarked node has lost no children, and a marked node has lost one child. Once a marked node loses another child, it has lost two children and thus needs to be moved back to the root list for reprocessing.

Binomial Heap

Insertion in Binomial Heap - O (log n):

➔ Create a Binomial heap with degree 'k'

➔ Call Union on existing heap 'H' and the new Binomial Heap.

getMin(H) - O(log n):

- ➔ To get the minimum, we need to traverse the complete root list, which is a singly linked list. But because of the above specified property of binomial heap (there are maximum log n trees in a binomial heap), it takes O(log n).
- ➔ We can also set a pointer to minimum node, and obtain the minimum in O(1) time, but then, after every extract_min operation, we need to update that, which consequently takes O(log n) time complexity.

extractMin(H) - O(log n):

- ➔ Call getMin(H)
- ➔ Remove the node with minimum pinter.
- ➔ Make new Binomial heap of the subtrees of the removed node.
- ➔ Call Union to merge both the heaps.

Union:

- ➔ Simply merge the two heaps in non-decreasing order of degrees.
- ➔ After merge, we need to make sure then there is at most one binomial tree of any one order. So any two binomial trees with same order are to be combined. We traverse the list of the merged roots, and keep track of tree-pointers previous and next.

Fibonacci Heap

Insertion in Fibonacci Heap:

Insertion in fibonacci heap takes O(1) time, as it simply adds the node to the root list and updates the min pointer.
Actual cost ci = O(1)

Change in Potential = $\Delta\theta = \theta(Hi) - \theta(Hi-1) = +1$ (tree with one node added)

Amortized Cost = $ci + \Delta\theta = O(1)$
So, insertion in fibonacci heap takes O(1) time complexity.

Trade-Off with Insertion in Fibonacci Heap:

Main trade-off here is when we insert k-nodes, the fibonacci heap would consist of just a root list of k nodes, so trade-off is when we perform extract-min operation on

heap, after removing the node that H.min points to, we would have to look through each of the remaining k-1 nodes in the root list during the extract_min operation, to find new minimum node & while doing this we also consolidate nodes into min-heap to reduce the size of the root list based on the same degree nodes in the root list.

Extract min in fibonacci heap:

Just remove the minimum and return that, now meld all the children nodes to the root list, update minimum.
Consolidate the tree so that no two nodes has same rank. This operation is little difficult and it takes O(log n) time.

```
Fib_Heap_Extract_min(H)
{
        z = H.min
        if ( z != NIL)
        {
                for each child x of z
                {
                        add x to the root list of H
                        x.p = NIL
                }
                remove z from the root list of H
                if (heap is empty)
                        H.min = NULL
                else{
                        H.min = z.right
                        consolidate(H)
                }
        }
         return z                        //which was the current minimum
}
```

The most complex step here is consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a *distinct degree* value.

1) First find root x & y in the root list with same degree. Let's say x.key <= y.key then,

2) Remove y from the list & make y a child of x by calling the Fib_Heap_Link procedure. This procedure increements the attribute degree & clears the mark on y.

Union:

The union operation takes O(1) as it only needs to concatenate the new root of the new heap in the existing root list. And then just updating the new min pointer by comparing the 2 current minimum of the 2 heaps.

vEB tree

The data structure is a tree with high degree and is defined recursively. Each van-Emde-Boas node v stores a set S belonging to U universe of items over a shrinking universe U with size of U = u. It has the following attributes:

Universe: universe is the maximum rounded of range of weight , rounded to nearest 2^{2^k} *value* .

clusters: There are √u clusters ,where each of them are Van-Emde-Boas nodes. Each cluster stores a set S' that is a subset of √u over the reduced universe √u. All the clusters will be managed by hash tables. Let x = c√u + i, then x tells the position to insert the element.

Summary: There is a summary structure that keeps track of which clusters are nonempty.

Below are the algorithms for finding minimum, finding maximum, insertion, deletion and extract minimum :

```
vEB-TREE-MINIMUM(V)
        return V.min

vEB-TREE-MAXIMUM(V)
        return V.max
```

```
vEB-EMPTY-TREE-INSERT (V, x)
        V.min = x
        V.max = x


vEB-TREE-INSERT (V, x)
        if V == NULL
                V =   vEB-CREATE-TREE(sqrt(V.u)) //creating new  memory for tree V of size
V.u
        If V.min == NULL
                vEB-EMPTY-TREE-INSERT (V,x)
        else if x < V.min
                exchange x with V.min
             if  V.u > 2
                if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NULL
                        vEB-TREE-INSERT(V.summary,high(x))
                vEB-TREE-INSERT(V.cluster[high(x)],low(x))
             if V.max < x
                V.max = x

vEB-TREE-DELETE(V,x)
        if x == V.min
                if vEB-TREE-MINIMUM(V.summary) == NULL
                        if V.min != V.max
                                V.min = V.max
                        else
                                V.min = NULL
                                V.max = NULL
                                V = NULL
                i = vEB-TREE-MINIMUM(V.summary)
                x = V.min = index(i,  vEB-TREE-MINIMUM(V.cluster[i]))
        if V.u > 2
                vEB-TREE-DELETE(V.cluster[high(x)], low(x))
                if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NULL
                        vEB-TREE-DELETE(V.summary, high(x))
        if x == V.max
                if vEB-TREE-MAXIMUM(V.summary) == NULL
                        V.max = V.min
                else
                        i = vEB-TREE-MAXIMUM(V.summary)
                        V.max = index(i, vEB-TREE-MAXIMUM(V.cluster[i]))
```

```
vEB-EXTRACT-MIN(V)
    X = vEB-TREE-MINIMUM(V)
    vEB-TREE-DELETE(V,x)
    return x
```

Time Complexity of the three Algorithms for Kruskal Implementation

- ➢ vEB tree :
  O(n log log n)
  Since, the extract min operation in vEB tree takes O(log log n), for n such operations, it will take O(n log log n).
- ➢ Fibonacci Heap:
  O(n log n)
  Since for extract min operation it takes O(log n), for n such operations, it will take O(n log n).
- ➢ Binomial Heap:
  O(n log n)
  Since for extract min operation it takes O(log n), for n such operations, it will take O(n log n).

Testing And Analysis

- ➢ The **green line** represents the nodes vs time plotting of fibonacci heap.
- ➢ The **blue line** represents the nodes vs time plotting of Binomial heap.
- ➢ The **Red line** represents the nodes vs time plotting of Van Emde Boas tree.
- ➢ The x-axis represents Number of Nodes.
- ➢ The y-axis represents Time complexity corresponding to the number of nodes.

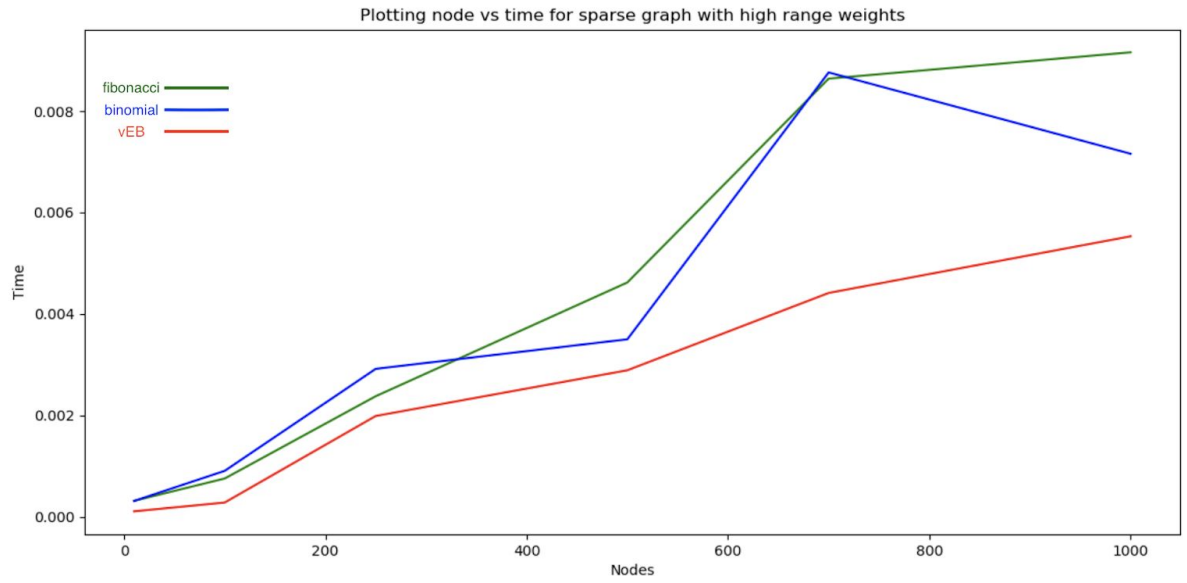
Case 1: Keeping the maximum weight of the edges constant and varying the the number of nodes, we plotted the increasing number of nodes with the time taken by the three algorithms. This plotting is for the sparse graphs.

➢ Below is the table showing the data of the plotted graph:

Keeping the maximum edge length constant, which is max_edge_length = 100000

(The reason for which we are considering the maximum edge weight is that vEB tree is constructed on the basis of maximum edge weight)

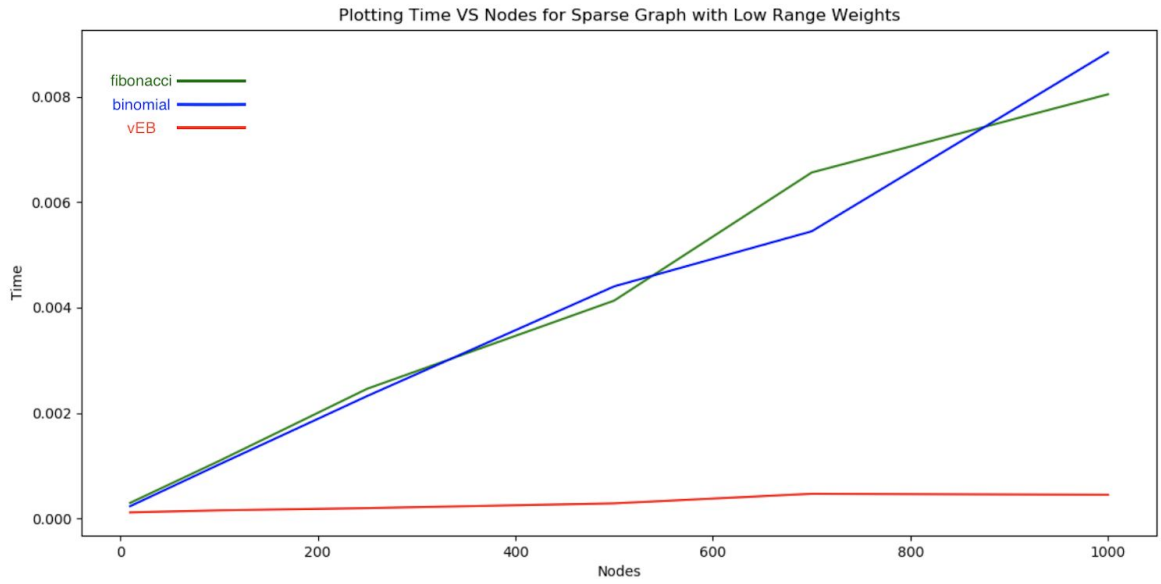
No. of Nodes	No. of Edges	vEB Tree	Fibonacci Heap	Binomial Heap
10	10	0.000105	0.000313	0.000310
100	100	0.000279	0.000753	0.000904
250	250	0.001986	0.002376	0.002916
500	500	0.002888	0.004620	0.003499
700	700	0.004413	0.008641	0.008763
1000	1000	0.005531	0.009160	0.007159



- ➢ We can clearly observe that with increase in number of nodes, vEB tree is outperforming fibonacci heap and Binomial heap.

- ➢ Initially fibonacci heap is performing better than binomial heap, because of its better amortized time complexity, but on increasing the number of nodes further binomial heap is outperforming fibonacci heap. The reason behind this is that fibonacci heap takes lesser time complexity for the similar operations in binomial heap, but with very high constant values, which are not ignorable, and hence we can see the effect of its hidden high constants.

Case 2: Plotting for the sparse graphs but for low maximum range of weight.



Maximum edge weight = 10

No. of Nodes	No. of Edges	vEB Tree	Fibonacci Heap	Binomial Heap
10	10	0.000116	0.000297	0.000234
100	100	0.000156	0.001091	0.001024
250	250	0.000198	0.002461	0.002323

500	**500**	**0.000288**	**0.004132**	**0.004402**
700	**700**	**0.000470**	**0.006561**	**0.005446**
1000	**1000**	**0.000451**	**0.008044**	**0.008838**

- With less range of maximum weight, vEB tree performs far better than fibonacci heap and binomial heap. The reason being, for all the operations vEB tree takes O(log log u) time, so irrespective of how many edges are there, and how many nodes are there, the time complexity of vEB which turns out to be exponentially lower than other O(log n) algorithms anyway becomes extremely low with O(log log 10) approximately, which gives almost constant time complexity even with increasing number of nodes.

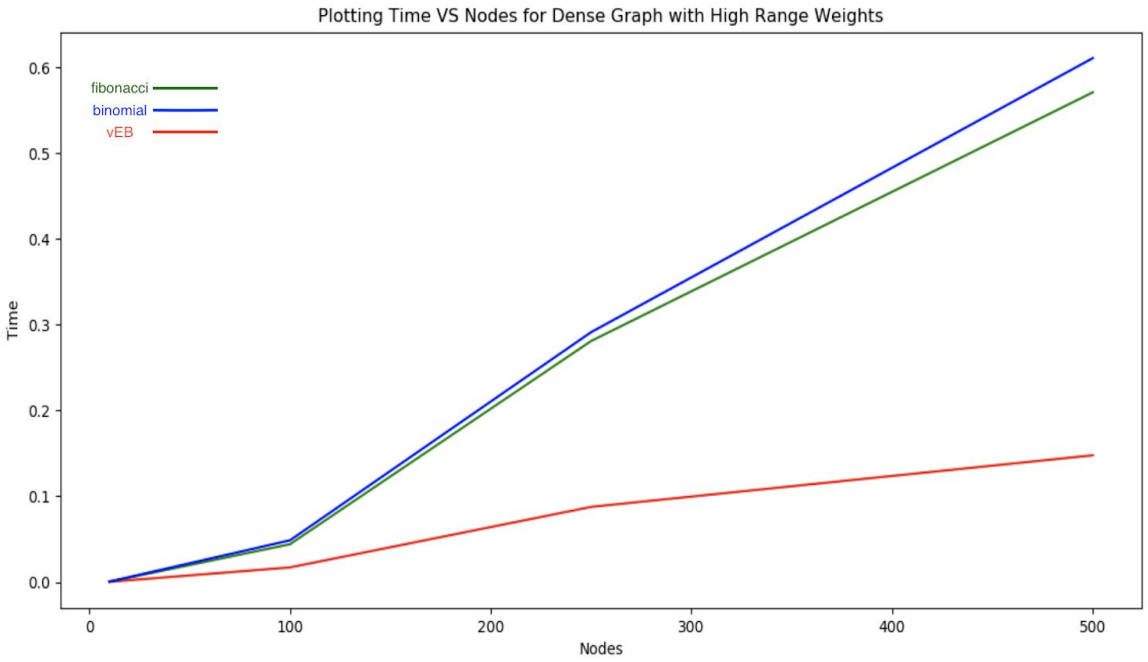
- Binomial and Fibonacci heaps though, keep competing with each other in terms of time complexity (This can be clearly observed from the graph). Sometimes, fibonacci outperforms binomial because of its amortized time complexity, while other times binomial heap outperforms fibonacci because of the hidden large constants in the time complexity of fibonacci heap.

Case 3: Plotting for the dense graphs for high maximum range of weight.

- Below is the table showing the data of the plotted graph:

Keeping the maximum edge length constant, which is Maximum edge weight = 100000

No. of Nodes	No. of Edges	vEB Tree	Fibonacci Heap	Binomial Heap
10	**45**	**0.000252**	**0.000380**	**0.000448**
100	**4950**	**0.017007**	**0.044044**	**0.048602**
250	**31125**	**0.087578**	**0.280988**	**0.291302**
500	**62375**	**0.147749**	**0.571062**	**0.610964**



- In an **amortized analysis,** the time required to perform a sequence of data-structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved. An amortized analysis guarantees the average performance of each operation in the worst case. And hence, in this case, where we have taken dense graph, the concept of amortized cost is utilized perfectly making the effect of hidden high constant negligible. So fibonacci heap makes sure that the time complexity is always better than binomial heap no matter to what extent the number of nodes are increased.
- If we observe the vEB tree performance in this graph, the result is so obvious. The exponentially better complexity of vEB tree makes it perform far better than the other 2 algorithms.
- If we compare this graph with the first graph, that is same range of edge weight but with sparse graph, we can see that, the difference between the performance of vEB tree and other 2 algorithms is increased to a great extent. The reason is that fibonacci and binomial heaps seem to be

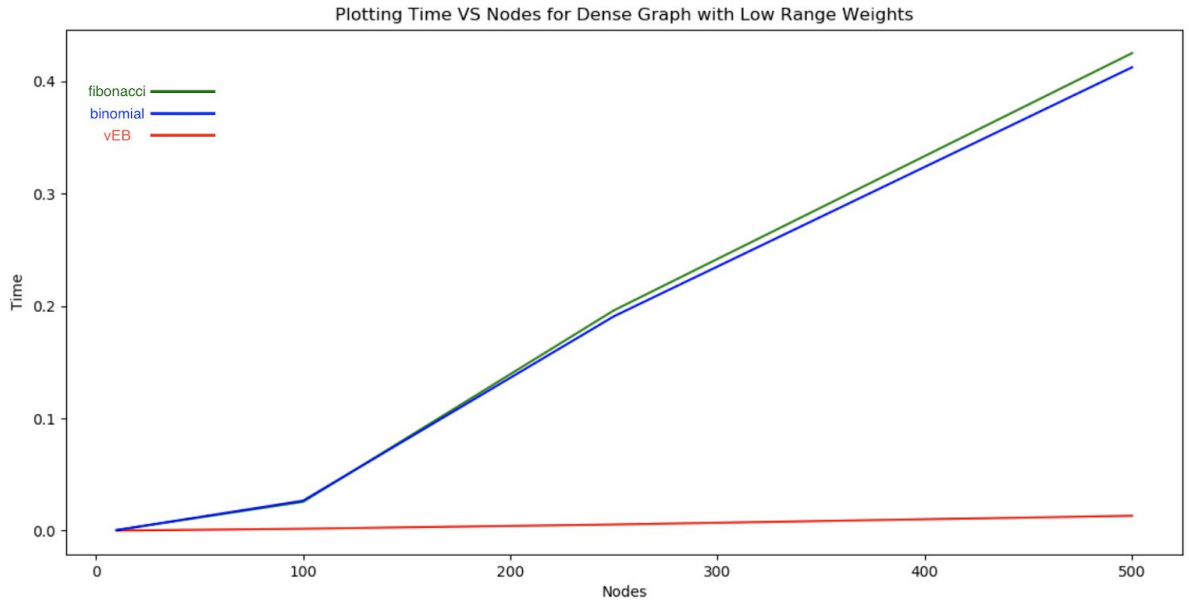
deteriorating their performances as the edges increase, while for vEB tree we all know, that vEB tree has good space efficiency when it contains a large number of elements, which is not the case with other algorithms. For fibonacci and binomial the performance deteriorated with increase in space.

Case 4 : Plotting for the **dense graphs** for low maximum range of weight.

➢ Below is the table showing the data of the plotted graph:

Keeping the maximum edge length constant,
Maximum edge length = 45

No. of Nodes	No. of Edges	vEB Tree	Fibonacci Heap	Binomial Heap
10	45	0.000122	0.000584	0.000565
100	4950	0.001752	0.025730	0.026713
250	31125	0.005545	0.196074	0.190784
500	62375	0.013283	0.424915	0.412317



- ➢ With dense graph, and low range of weight, vEB performs in very obvious and expected way. The complexity remains almost constant even with the increase in nodes. This behavior is similar to its behaviour in case of sparse tree with low edge weight range(case 2).
- ➢ Unlike case 2, where the behaviour of binomial and fibonacci heap were switching for different ranges of number of nodes, here, in this case it seems certain for binomial to perform uniformly fairly well as compared to fibonacci heap with increase in nodes. It seems, that in dense graph, with less range of edge weight, the hidden constants have more effect on the time complexity. So we can assume that with dense graph and less edge weight range the amortized cost for fibonacci heap turns out to be higher, because of its hidden high constants.

Conclusion

From the above analysis we analysed that in all the above cases vEB outperformed the other two algorithms, but we can conclude it more precisely as follows:

- When the edge weight range is smaller
  - In this case vEB performs fairly well and the performance is far better than any other algorithm.
  - Nothing certain can be concluded for fibonacci and binomial heap in this case.
  - However, we can still say that Binomial is preferable in case of dense graph.

- When the graph is dense, and the range of edge weight is quite big.
  - Fibonacci is preferable over Binomial Heap in this case.
  - vEB tree outperforms both of them.

- When graph is sparse and the range of edge weight is quite big.
  - Binomial and Fibonacci outperform each other for different ranges of number of nodes.
  - vEB tree outperforms both of them.

Percentage gain

CASE	FIBONACCI HEAP(%)	BINOMIAL HEAP(%)
1	45.283	42.809
2	86.475	84.39
3	59.50625	63.626
4	91.584	91.429

Remark : The above analysis is based on implementation of kruskal's algorithm, which takes into consideration mainly the extract_min operation. If we will compare the other operations like tree creation, then vEB tree performs little less better than it performs now, and in some cases worse than Fibonacci and Binomial Heap. The reason is the hidden high constants. For each step in each operations in vEB tree, it needs to perform quite large mathematical operations, and because of its extremely complicated structure, the operations on each node, will give higher constant values as compared to other algorithms, which makes vEB tree perform bad and less preferable in some cases.

Applications and Scope

- In practice, the van Emde Boas layout (based on van Emde Boas tree) is used in cache-oblivious data structures meaning faster data access .In computing, a cache-oblivious algorithm (or cache-transcendent algorithm) is an algorithm designed to take advantage of a CPU cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter. A common cache-oblivious data structure is cache-oblivious search tree.
- Priority queues are essential for various network processing applications, including per-flow queueing with quality-of-service (QoS), guarantees management of large fast packet buffers, and management of statistics counters. So, the high-performance priority queues can be implemented based on van Emde Boas Tree to achieve this algorithm in O(log log u) time

complexity where u is the universe size which actually represents the priority keys, which is done in O(log n) using other efficient algorithms.
- Pipelined version of van Emde Boas Tree can be used to get an amortized time complexity of O(1).

Git Repository

- *https://github.com/sup19-19/vEB-tree-to-Implement-Kruskal-s-Algorithm.git*

References

- *https://www.slideserve.com/amayeta/pipelined-van-emde-boas-tree-algorithms-analysis-and-applications*

- *http://fileadmin.cs.lth.se/cs/Personal/Rolf_Karlsson/lect12.pdf*