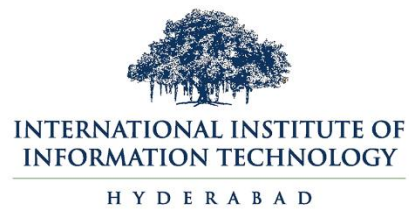


INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD



An Advanced Problem Solving Project on:

Implementation of Kruskal's Algorithm with vEB tree and its comparison with AVL Tree  
and Union Find

Rishab (2019201050)

Aman Dev Nautiyal (2019201095)

## 1 Introduction

Comparison of performance of Kruskal's Minimum Spanning Tree algorithm with its implementation through van Emde Boas Tree and AVL tree with Union Find. Kruskal's Algorithm uses Union Find operation. Instead of sorting an array for extracting the next smallest edge, we are using van Emde Boas Tree and AVL tree for performing extract min operation.

### 1.1 Van Emde Boas Tree

- Given a set **S** of elements such that the elements are taken from universe  $\{0, 1 \dots u-1\}$ .
- **Van Emde Boas tree (or vEB tree)** supports `insert()`, `delete`, `find()`, `successor()` and `predecessor()` operations in  $O(\log \log u)$  time, and `max()` and `min()` in  $O(1)$  time
- A thing to note about the vEB trees is that the time taken by it to perform the above operations is independent of the elements which are actually present in the tree. It depends on the universe size i.e 'u'.
- The operations supported by Van Emde Boas Tree are:
  1. `Insert(x)`: insert a value in the tree T
  2. `isEmpty()`: Returns true if T is empty else returns false
  3. `Delete(x)`: Deletes item x from the tree
  4. `Find(x)`: Returns true if value is present else returns false
  5. `Max(T)`: returns the maximum value present in the tree
  6. `Min(T)`: returns the minimum value present in the tree
  7. `Sucessor(x,T)`: returns the smallest value present in the tree  $\geq x$

Below are different solutions for the above problem.

- One solution to solve above problem is to use a **self-balancing Binary Search Tree** like **Red-Black Tree**, **AVL Tree**, etc. With this solution, we can perform all above operations in  $O(\log n)$  time.
- Another solution is to use **Binary Array (or Bit vector)**. We create an array of size  $u$  and mark presence and absence of an element as 1 or 0 respectively. This solution supports `insert()`, `delete()` and `find()` in  $O(1)$  time, but other operations may take  $O(u)$  time in worst case.
- **Van Emde Boas tree (or vEB tree)** supports `insert()`, `delete`, `find()`, `successor()` and `predecessor()` operations in  $O(\log \log u)$  time, and `max()` and `min()` in  $O(1)$  time.

Let us see the implementation details of van Emde Boas tree.

- Van Emde Boas Tree is recursively defined structure.
- In Van Emde Boas Tree, Minimum and Maximum queries works in  $O(1)$  time as Van Emde Boas Tree stores Minimum and Maximum keys present in the tree structure.

Advantages of adding Maximum and Minimum attributes, which help to decrease time complexity:

- If any of Minimum and Maximum value of VEB Tree is empty (NIL or -1 in code) then there is no element present in the Tree.
- If both Minimum and Maximum is equal then only one value is present in the structure.
- If both are present and distinct then two or more elements are present in the Tree. We can insert and delete keys by just setting maximum and minimum values as per conditions in constant time ( $O(1)$ ) which helps in decreasing recursive call chain: If only one key is present in the VEB then to delete that key we simply set min and

max to the nil value. Similarly, if no keys are present then we can insert by just setting min and max to the key we want to insert. These are  $O(1)$  operations.

- In successor and predecessor queries, we can take decisions from minimum and maximum values of VEB, which will make our work easier

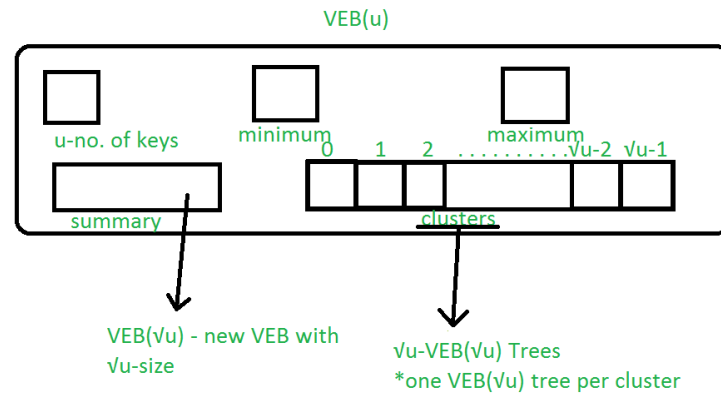


Figure: Sample of an auxillary van emde boas structure

A  $vEB$  tree  $T$  over the universe  $\{0, \dots, M-1\}$  has a root node that stores an array  $T.children$  of length  $\sqrt{M}$ .  $T.children[i]$  is a pointer to a  $vEB$  tree that is responsible for the values  $\{i\sqrt{M}, \dots, (i+1)\sqrt{M}-1\}$ . Additionally,  $T$  stores two values  $T.min$  and  $T.max$

Data storage in a VEB structure is as follows:

The smallest value currently in the tree is stored in  $T.min$  and largest value is stored in  $T.max$ . Note that  $T.min$  is not stored anywhere else in the  $vEB$  tree, while  $T.max$  is. If  $T$  is empty then we use the convention that  $T.max=-1$  and  $T.min=M$ . Any other value  $x$  is stored in the subtree  $T.children[i]$  where  $i = \lfloor x/\sqrt{M} \rfloor$ . The auxiliary tree  $T.aux$  keeps track of which children are non-empty, so  $T.aux$  contains the value  $j$  if and only if  $T.children[j]$  is non-empty.

- MEMBER operation can be performed in  $O(1)$  time, what is better than  $O(\log n)$  for BST.
- If the number of elements  $n$  is much smaller than size of the universe  $u$  than BST would be faster for all the other operations.

1. Pseudocode for finding successor of an element  $x$  in the tree:

```

FindSucessor(T,x)
if  $x < T.min$  then
    return  $T.min$ 
if  $x \geq T.max$  then // no next element
    return  $M$ 
 $i = \text{floor}(x/\sqrt{M})$ 
 $lo = x \bmod \sqrt{M}$ 
 $hi = \sqrt{M} \text{ FindSucessor}(T.aux, i)$ 
if  $lo < T.children[i].max$  then
    return  $hi + \text{FindSucessor}(T.children[i], lo)$ 
return  $hi + T.children[\text{FindSucessor}(T.aux, i)].min$ 

```

2. Pseudocode for inserting an element  $x$  into the tree

```

function  $\text{Insert}(T, x)$ 
if  $T.min > T.max$  then // T is empty
     $T.min = T.max = x;$ 
    return
if  $x < T.min$  then
     $\text{swap}(x, T.min)$ 

```

```

if x > T.max then
    T.max = x
    i = floor(x /  $\sqrt{M}$ )
    lo = x mod  $\sqrt{M}$ 
    Insert(T.children[i], lo)
if T.children[i].min == T.children[i].max then
    Insert(T.aux, i)
end

```

3. Pseudocode for deletion of an element x from the tree.

Deletion from VEB tree is a tricky task.

1. We check if only one key is present, then assign the maximum and minimum of the tree to null value to delete the key.

Base Case:

2. If the universe size of the tree is two then, after the above condition of only one key is present is false, exactly two key is present in the tree (after the above condition turns out to false), So delete the query key by assigning maximum and minimum of the tree to another key present in the tree.

Recursive Case:

3. If the key is the minimum of the tree then find the next minimum of the tree and assign it as the minimum of the tree and delete query key.
4. Now the query key is not present in the tree. We will have to change the rest of the structure in the tree to eliminate the key completely:
  - a. If the minimum of the cluster of the query key is null then we will delete it from summary as well. Also, if the key is the maximum of the

tree then we will find new maximum and assign it as the maximum of the tree.

- b. Otherwise, if the key is maximum of the tree then find the new maximum and assign it as the maximum of the tree.

## 1.2 AVL (ADELSON-VELSKY AND LANDIS) TREE

An AVL Tree is a self-balancing binary search tree. It was the first such data structure to be invented. AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

If the above condition is violated after any operation then rebalancing is done to restore this property. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take  $O(h)$  time where  $h$  is the height of the BST. The cost of these operations may become  $O(n)$  for a skewed Binary tree. If we make sure that height of the tree remains  $O(\log n)$  after every insertion and deletion, then we can guarantee an upper bound of  $O(\log n)$  for all these operations. The height of an AVL tree is always  $O(\log n)$  where  $n$  is the number of nodes in the tree.

So operations like Insertion, Deletion, Find(x), FindMin, FindMax all are carried out in a worst case time of  $O(\log n)$ .

### 1. Pseudocode for Insertion into an AVL tree.

Let the newly inserted node be  $w$

- Perform standard BST insert for w.
- Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

Rotation is done accordingly for each of the cases such that both the AVL height balancing property and the BST property is maintained.

## 2. Pseudocode for Deletion in an AVL Tree

- 1)** Perform the normal BST deletion.
- 2)** The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- 3)** Get the balance factor (left subtree height – right subtree height) of the current node.
- 4)** If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left



Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.

**5)** If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

## **2 Implementation**

### **2.1 Kruskal's Algorithm**

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy Algorithm.

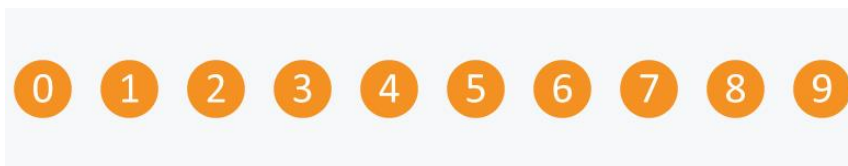
This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

### **2.2 Disjoint set Union**

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data structure consists of only three operations:


- **make\_set(v)** - creates a new set consisting of the new element **v**
- **union\_sets(a, b)** - merges the two specified sets (the set in which the element **a** is located, and the set in which the element **b** is located)
- **find\_set(v)** - returns the representative (also called leader) of the set that contains the element **v**. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after **union\_sets** calls). This representative can be used to check if two elements are part of the same set or not. If **find\_set(a) == find\_set(b)**, then they are in the same set, Otherwise they are in different sets. This data structure allows us to do each of these operations in almost  $O(1)$  time on average.
- Figure to show the working of union-find:



|     |   |   |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|---|
| Arr | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

The initial set is shown above, after performing few union operations and find operations the figures are shown below:

- Further after performing  $\text{Union}(4, 3)$ ,  $\text{Union}(8, 4)$ ,  $\text{Union}(9, 3)$  and  $\text{Union}(6, 5)$



- Arr looks as follows:
- Find (0, 7) - as 0 and 7 are disconnected, this will give false result
- Find (8, 9) - though 8 and 9 are not connected directly, but there exists a path connecting 8 and 9, so it will give us true result.

|     |   |   |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|---|
| Arr | 0 | 1 | 1 | 3 | 3 | 5 | 5 | 7 | 3 | 3 |
|     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

## 2.1 Implementation of Kruskal's through Van Emde Boas Tree:

We are using the following methods while Implementing Kruskal's Algorithm using VEB Tree.

- VEB tree is initialized for a given universe size(here 65536) with the above mentioned VEB structure. Here 65536 is the maximum edge weight that can occur in the graph.
- Following are the methods we are using while implementing VEB for Kruskal's:
  - Insert( $x, T$ ) : To add an element(weight)  $x$  to VEB structure  $T$ .  $O(\log \log u)$ .
  - GetMin( $T$ ): To get the minimum edge weight present in the tree.  $O(1)$ .
  - Delete( $x, T$ ): To delete an edge weight. Complexity =  $O(\log \log u)$ .
- During the whole process, we maintain an unordered\_map provided by STL to deal with duplicate values.
- The key for indexing into the map is the edge weight and the corresponding value mapped to it is a forward\_list(STL) which contains the corresponding edges for that weight.
- Only when all the edges of a weight are exhausted; we remove the corresponding weight entry from the VEB tree .
- We create forests using Union operation described earlier. We perform this operation in non-decreasing order of edge weights.
- For each extracted edge we perform a Find operation to ensure that the participating nodes do not form an a cycle(by ensuring they don't have the same parent)
- If there is no cycle we perform the Union operation for the two nodes of the edge.

- This is performed for  $(n-1)$  edges, and hence the Minimum Spanning Tree is obtained.
- For this implementation we took the universe size of the form  $(2^{(2^k)})$ . This is done to ensure that we can calculate  $\text{upsqrt}(U)$  and  $\text{lowsqrt}(U)$  by a simple  $\text{sqrt}(U)$  operation. This was done to lower the calculation time overheads.

## 2.2 Implementation using AVL

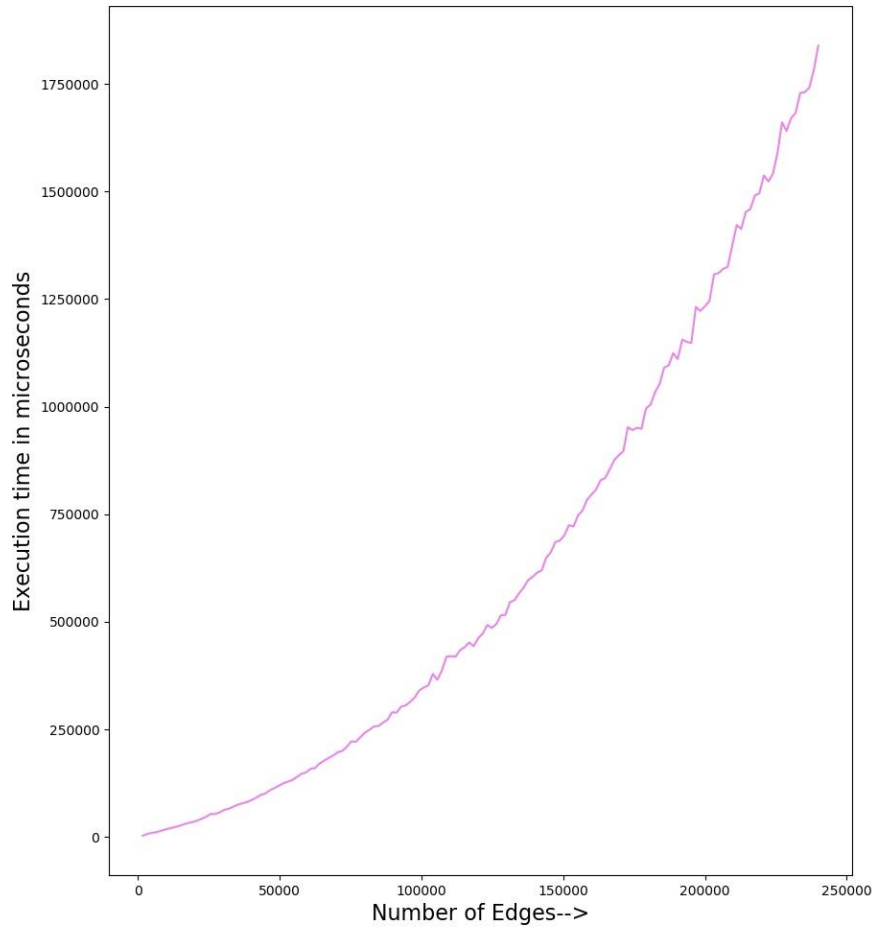
We are using the following methods while Implementing Kruskal's Algorithm using AVL Tree.

- Following are the methods we are using while implementing AVL Tree for Kruskal's:
  - $\text{Insert}(x)$  : Insert an element(edge weight) into the tree.  $O(\log n)$ .
  - $\text{Min\_val}()$  : Extracting the minimum weight value from the AVL Tree. Complexity =  $O(\log n)$ .
  - $\text{Delete}(x)$  : Deletes the node with value  $x$  from the AVL Tree. Complexity =  $O(\log n)$ .
- During the whole process, we maintain an `unordered_map` provided by STL to deal with duplicate values.
- The key for indexing into the map is the edge weight and the corresponding value mapped to it is a `forward_list(STL)` which contains the corresponding edges for that weight.

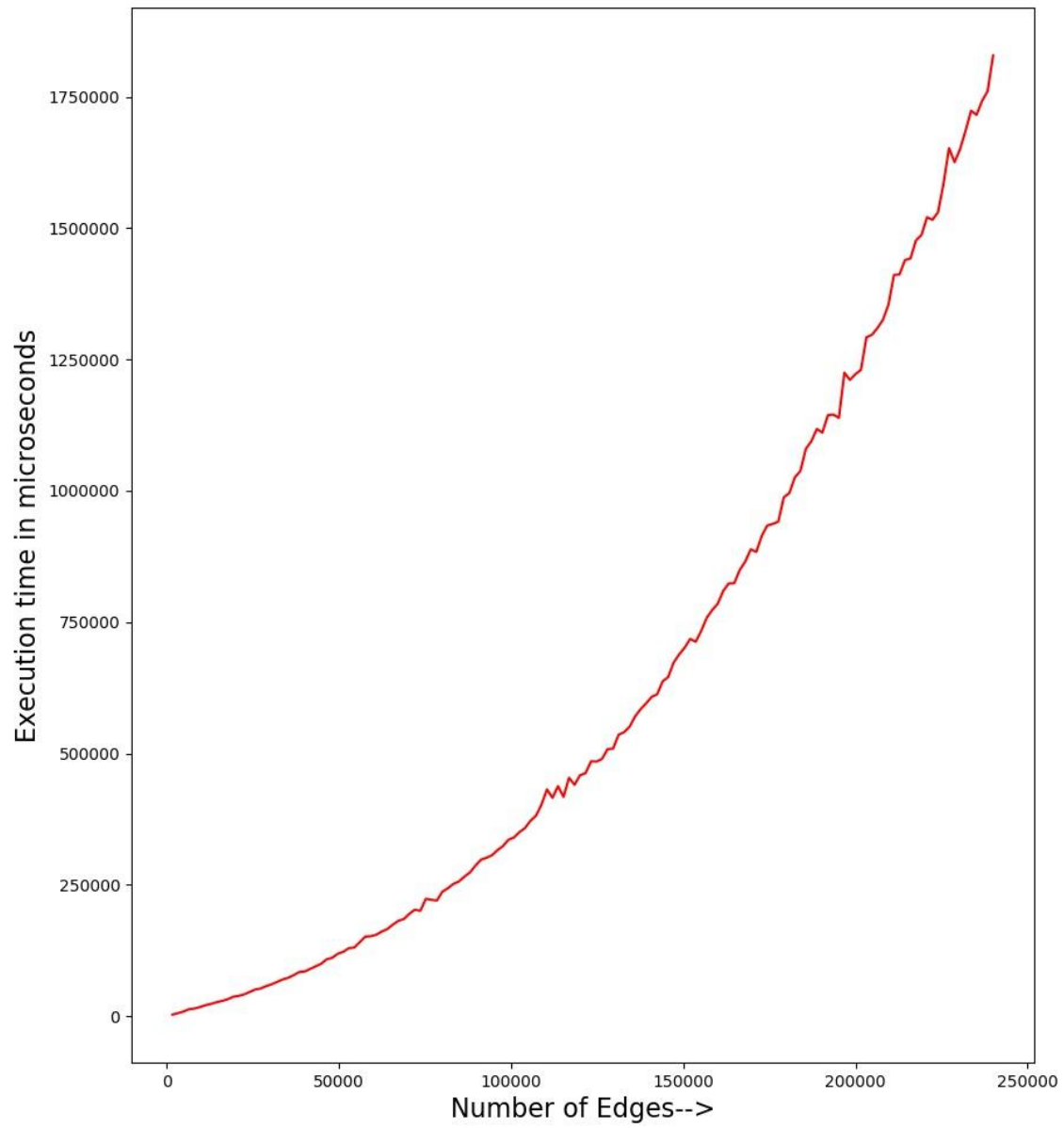
- Only when all the edges of a weight are exhausted; we remove the corresponding weight entry from the AVL tree .
- We create forests using Union operation described earlier. We perform this operation in non-decreasing order of edge weights.
- For each extracted edge we perform a Find operation to ensure that the participating nodes do not form an a cycle(by ensuring they don't have the same parent)
- If there is no cycle we perform the Union operation for the two nodes of the edge.
- This is performed for  $(n-1)$  edges, and hence the Minimum Spanning Tree is obtained.

### 3 Individual and Comparison of performances of VEB and AVL Tree:

#### 3.1 AVL TREE

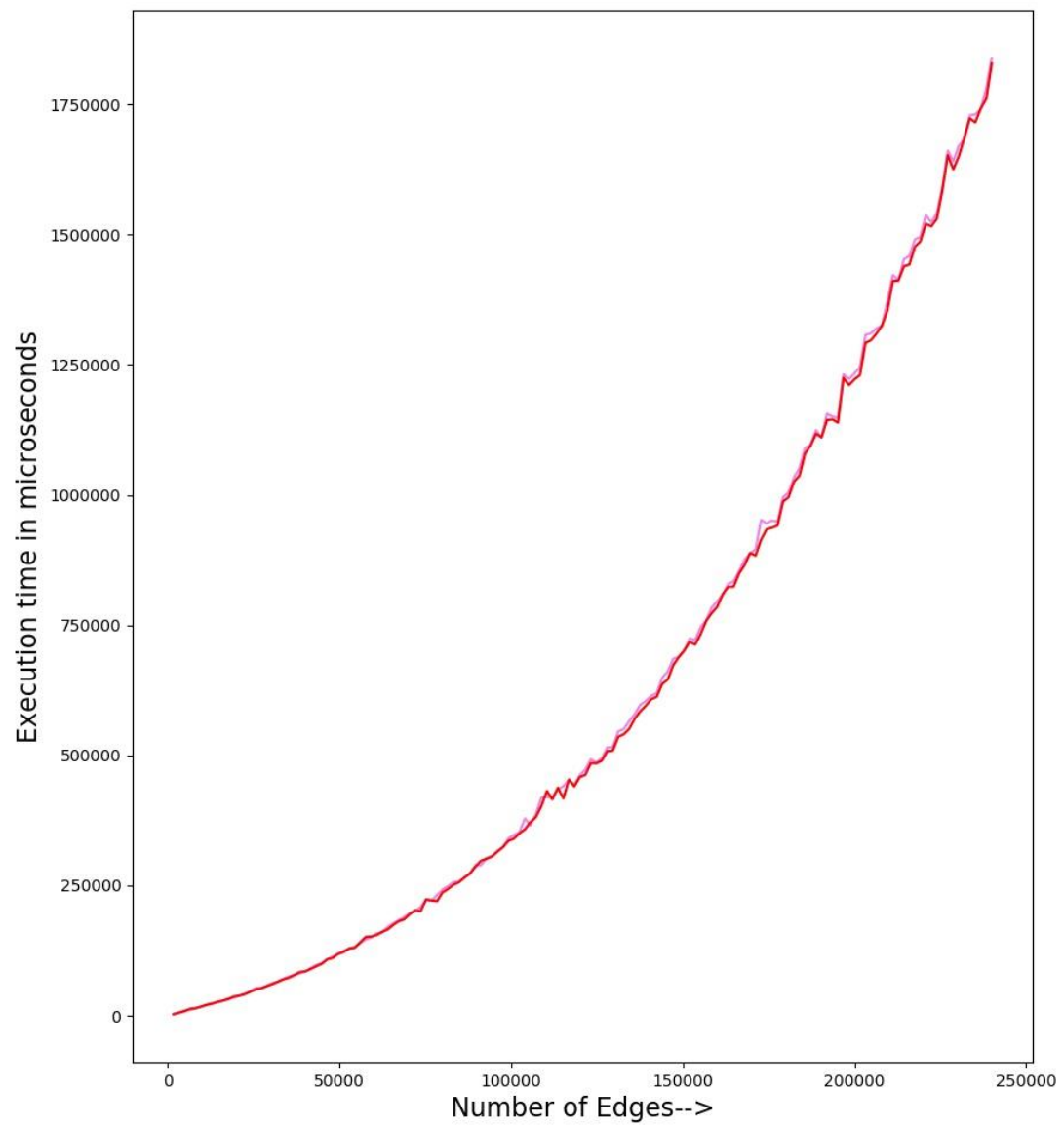


### 3.2 VAN EMDE BOAS TREE





### 3.3 VEB vs AVL Tree



- The weights lied between the range (1,65536], consequently we fixed the universe size of VEB tree to be 65536 .
- We can see that for the initial values, the AVL tree outperforms the VEB tree. The reason for this are the constants that are associated with the VEB tree, which has a fixed structure, unlike the dynamically changing AVL tree.
- For higher values, the VEB tree starts to outperform the AVL tree as the  $O(\log \log u)$  (u being universe size) factor of vEB tree becomes significantly smaller than the  $O(\log n)$  (n being the number of edges in tree) of the AVL tree.
- The increase in performance is not very pronounced at the given range. However following the same trend, the performance of vEB tree should be considerably better than the AVL tree at higher number of edges and the weight range.

For implementation details visit:  
[https://www.github.com/AmanNautiyal/Kruskal\\_vEB\\_AVL](https://www.github.com/AmanNautiyal/Kruskal_vEB_AVL)