Implementation of van Emde Boas tree with application to Kruskal and compare wrt Union Find and AVL.

P V HARIDATTA 2019201010

KURUVA JAYA KRISHNA 2019201076

1.OBJECTIVE:

Kruskal’s algorithm Implementation using vEB tree,AVL,Union Find and their performance comparison.

2.INTRODUCTION :

2.1 MINIMUM SPANNING TREE :

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. *(MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

2.2 KRUSKAL 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 in graph theory as it finds a minimum spanning tree for a connected weighted graph adding non decreasing cost edges at each step. 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. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

2.3 DISJOINT DATA STRUCTURE :

A disjoint-set data structure is a data structure that tracks a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set. Disjoint-sets play a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph.

2.3.1 OPERATIONS ON DISJOINT DATA STRUCTURE :

- **MakeSet:** Makes a new set by creating a new element with a unique id, and initializing the parent to itself. The MakeSet operation has O(1) time complexity, so initializing n sets has O(n) time complexity
- **Find(x):** Follows the chain of parent pointers from x till the root element, whose parent is itself. Returns the root element. Time Complexity: O(logn)
- **Path compression:** Path compression flattens the structure of the tree by making every node point to the root whenever Find is used on it. This is valid, since each element visited on the way to a root is part of the same set. The resulting flatter tree speeds up future operations not only on these elements, but also on those referencing them.
- **Union(x,y):** Merges x and y into the same partition by attaching the root of one to the root of the other. If this is done naively, such as by always making x a child of y, the height of the trees can grow as O(n). Toprevent this, union by rank or union by size is used.

Union by rank: Union by rank always attaches the shorter tree to the root of the taller tree. Thus, the resulting tree is no taller than the originals unless they were of equal height, in which case the resulting tree is taller by one node. In union by rank, each element is associated with a rank. Initially a set has one element and a rank of zero. If we union two sets and if they have the same rank, the resulting set's rank is one larger. otherwise, if we union two sets and if they have different ranks, the resulting set's rank is the larger of the two. Ranks are used instead of height or depth because path compression will change the trees' heights over time.

Union by size: In Union by size always attaches the tree with fewer elements to the root of the tree having more elements. With neither path compression (or a variant), union by rank, nor union by size, the height of trees can grow unchecked as O(n), which implies that Find and Union operations will take O(n) time.

2.4 AVL :

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.

In an AVL Tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

- insert(x) : O(log n).
- is Empty() : O(1)
- find(x) : O(log n)
- delete(x) : O(log n).
- max() : O(log n).
- min() : O(log n).

2.5 vEB :

In order to gain insight for our problem we shall examine the following preliminary approaches for storing a dynamic set :

- Direct Addressing.

• Superimposing a binary tree structure.

• Superimposing a tree of constant height

2.5.1 DIRECT ADDRESSING

The direct-addressing stores the dynamic set as a bit vector. • To store a dynamic set of values from the universe { 0, 1, ..., u-1 } we maintain an array A[0 .. u-1] of u bits.

• Entry A[x] holds 1 if the value x is in the dynamic set and 0 otherwise.

• INSERT, DELETE and MEMBER operations take O(1)time with this bit vector.

• MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR can take O(u) in the worst case as we might have to scan through O(u) elements.

2.5.2 SUPERIMPOSING A BINARY TREE STRUCTURE :

• One can short-cut long scans in the bit vector by superimposing a binary tree on the top of it.

• Entries of the bit vector form the leaves of the binary tree.

• Each internal node contain 1 if and only if in its subtree contains 1.

• Bit stored in an internal node is the logical-or of its children.

2.5.3 MINIMUM:

• Algorithm: start at the root and head down toward the leaves, always taking the leftmost node containing 1.

• Time complexity: O(log u)

2.5.4 MAXIMUM:

• Algorithm: start at the root and head down toward the leaves, always taking the rightmost node containing 1.

• Time complexity: O(log u)

2.5.5 PREDECESSOR (A, x) :

• Start at the leaf indexed by x and head up toward the root until we enter a node from the right and this node has a 1 in its left child z.

• Head down through node z, always taking the rightmost node containing a 1. Time complexity: O(log u).

2.5.6 SUCCESSOR (A, x)

• Start at the leaf indexed by x and head up toward the root until we enter a node from the left and this node has a 1 in its right child z.

• Head down through node z, always taking the leftmost node containing a 1. Time complexity: O(log u).

2.5.7 STRUCTURE OF vEB NODE :

Van Emde Boas Tree supports search, successor, predecessor, insert and delete operations in O(lglgN) time which is faster than any of related data structures like priority queue, binary search tree, etc. Van Emde Boas Tree works with O(1) time-complexity for minimum and maximum query.

2.5.8 Van Emde Boas Tree is a recursively defined structure

1. **u**: Number of keys present in the VEB Tree.
2. **Minimum:** Contains the minimum key present in the VEB Tree.
3. **Maximum:** Contains the maximum key present in the VEB Tree.
4. **Summary:** Points to new VEB(sqrt(u) Tree which contains overview of keys present in clusters array.

5. **Clusters:** An array of size each place in the array points to new VEB(sqrt(u) Tree

In Proto Van Emde Boas Tree the size of universe size is restricted to be of type 2^{2k} but in Van Emde Boas Tree, it allows the universe size to be exact power of two. So we need to modify High(x), low(x), generate_index() helper functions used in Proto Van Emde Boas Tree as below.

High(x): It will return floor(x/ceil(srqt(u)), which is basically the cluster index in which the key x is present.

Low(x): It will return x mod ceil(sqrt(u)) which is its position in the cluster

generate_index(a, b) : It will return position of key from its position in cluster b and its cluster index a.

3 IMPLEMENTATION :

3.1 IMPLEMENTATION USING DISJOINT DATA STRUCTURE :

The structure of Node consists of following items:

- from,to : ends of Edges

- weight : weight of Edges

Methods used while Implementing .

- unio : for performing union operation on two vertices.

- find : to find representative elements of two vertices.

3.2 IMPLEMENTATION USING AVL TREE :

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

• The structure of AVL Tree node consists of following items:

– data : Weight of Edge

– *left : left subtree pointer

– *right : right subtree pointer

– height : height of that node

- from,to : ends of Edge

- weight : weight of Edge

• Methods used while implementing AVL Tree for Kruskal

– insert : Insert the node in the AVL Tree.

Complexity = O(log E).

For E insert operations Complexity =O(ElogE)

– Inorder Traversal: To get Edge weights in sorted order

Complexity = O(E).

- unio : for performing union operation on two vertices.

- find : to find representative elements of two vertices.

3.3 IMPLEMENTATION USING VAN EMDE BOAS TREE

We are using the following methods while Implementing Kruskal's Algorithm using Van Emde Boas Tree.

• The structure of Binomial heap node consists of following items:

– U : Universe Size

– min : minimum of each cluster

– max : maximum of each cluster

– *summary : Summary of entire cluster

– **cluster : maintains the pointers to the nodes of vEB Tree.

• Following are the methods we are using while implementing vEB Tree for Kruskal's:

– veb_minimum() : To get the minimum weight edge from the vEB Tree. Complexity = O(1).

– delete_element() : Deletes the node from the vEB Tree. Complexity = O(log log u).

– insert_node(): Insert the node in the vEB Tree.

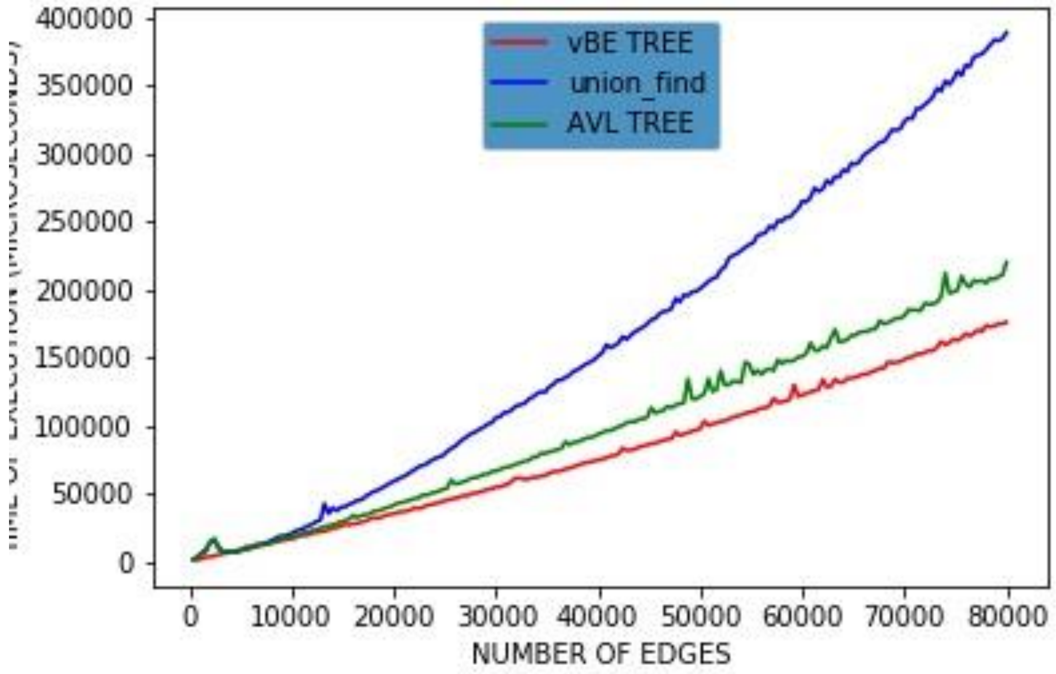
Complexity = O(log log u).

4 COMPARISON OF PERFORMANCE OF DATA STRUCTURES :

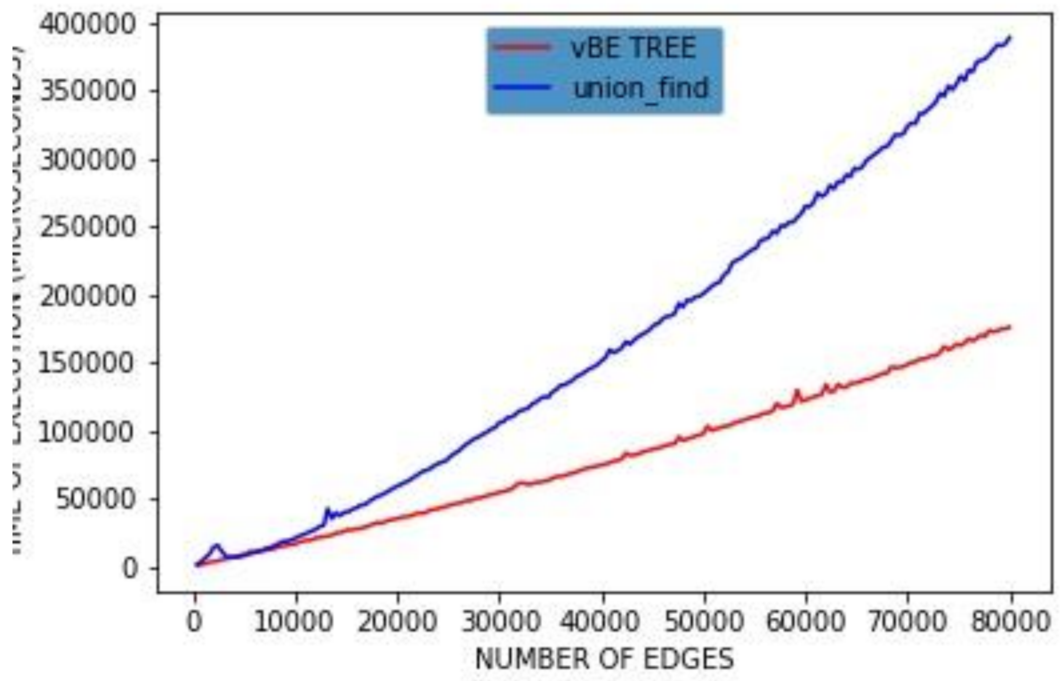
4.1.PERFORMANCE COMPARISON OF AVL,vBE,Disjoint_union :

y-axis : Time taken in Microseconds

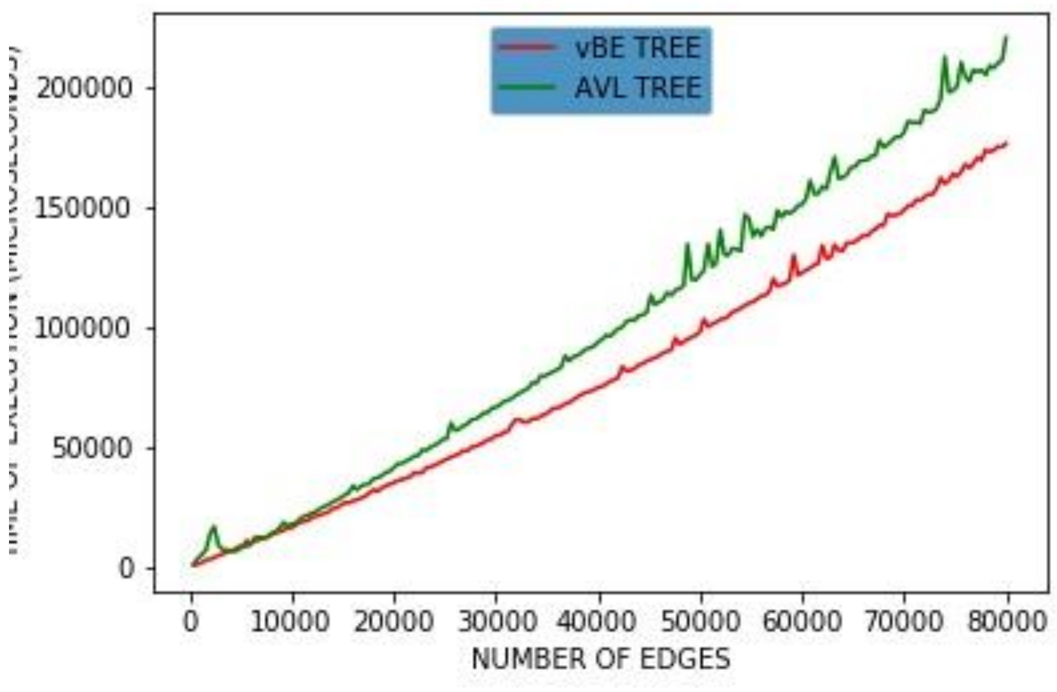
x-axis : Number of edges


vBE TREE
union_find
AVL TREE
400000
350000
300000
250000
200000
150000
100000
50000
0
0
10000
20000
30000
40000
50000
60000
70000
80000
NUMBER OF EDGES

4.2.PERFORMANCE COMPARISON OF vEB tree,Disjoint Union :



4.3.PERFORMANCE COMPARISON OF vEB tree,AVL tree



5.MANUAL :

5.1.Generating input :

1. Run generate.py to generate 200 files containing random number of edges with corressponding random edges weights of range 0- 100000

2.Run avl_kruskals.cpp,veb_kruskals.cpp,union_find.cpp which stores result in files avl.txt,union_file.txt,veb.txt.

3.Run plot_graph.py to generate comparison graphs.