International Institute of Information Technology
Hyderabad

Implementation of van Emde Boas tree with application to Kruskal and compare with resepect to AVL tree and union Find

Akash Kumar (2019201046)
akash.kumar@students.iiit.ac.in
Manik Gupta (2019202007)
manik.gupta@students.iiit.ac.in

4th November 2019

A project report submitted as a part of the course

Advanced Problem Solving

1 Objective

Kruskal's algorithm implementation using vEB tree, AVL tree and normal union find algorithm and their performance comparison

2 Introduction

2.1 Minimum spanning tree

A minimum spanning tree (MST) is a subset of the edges of a connected, weighted and undirected graph that connects all the vertices together with the minimum possible total edge weight and no cycles present .By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned by all its vertices. Minimum Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).

Common application of spanning trees are −

- Constructing trees for broadcasting computer networks.
- Taxonomy
- Cluster Analysis

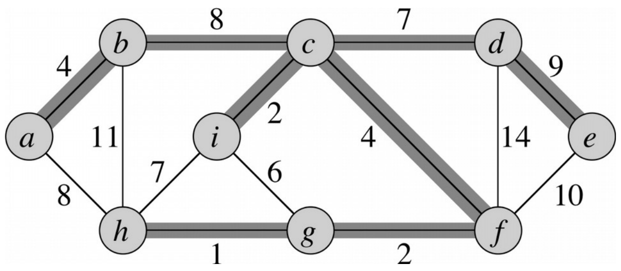

Figure 1: Minimum Spanning Tree

2.2 Kruskal Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approachThis algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties. 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 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

1. MakeSet: Makes a new set by creating a new element with a unique id, and initializing the parent to itself. Initially, size of each vertex is 1. The MakeSet operation has O(1) time complexity, so initializing n sets has O(n) time complexity

2. Find(x): Follows the chain of parent pointers from x till the root element, whose parent is itself and simultaneously updating the parent of every node in the path so that next time it will take less time. Returns the root element. Time Complexity: O(logn)

3. 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

4. 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). To prevent this union by rank or union by size is used by rank.

Union by rank: The above operations can be optimized to O(Log n) in worst case. The idea is to always attach smaller depth tree under the root of the deeper tree. This technique is called union by rank. The term rank is preferred instead of height because if path compression technique (we have discussed it below) is used, then rank is not always equal to height. Also, size (in place of height) of trees can also be used as rank. Using size as rank also yields worst case time complexity as O(Logn)

2.4 AVL Tree

AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree. Balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree. 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(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

2.4.1 The following operations of AVL Tree are used for the project:

1. Insert: Insert the weight of the edge of the tree.

   Time Complexity: O (log V)

2. Delete: Delete the weight of the edge from the tree .

   Time Complexity: O (log V)

3. Min/Max: Find the minimum/maximum value present in the tree. Time Complexity: O (log V)

2.5 vEB Tree

Van Emde Boas tree is a tree data structure which implements an associative array with m-bit integer keys. Van Emde Boas Tree supports search, successor, predecessor, insert and delete operations in O(log log u) time where u=2^m is the size of universe represnting maximum number of elements that can be stored in the tree. The ‘u’ is not to be confused with the actual number of elements stored in the tree, by which the performance of other tree data-structures is often measured. IvEB tree is faster than any of related data structures like priority queue, binary search tree, etc.

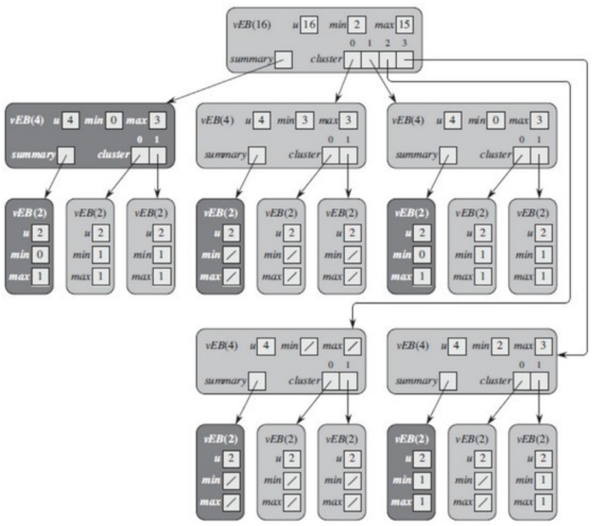

Figure 2: Van Emde Boas Tree

2.5.1 The following operations of vEB Tree are used for the project:

1. Insert: insert a m-bit value into the tree.

   Time Complexity: O(log log u)
2. Delete: delete a m-bit value from the tree.

   Time Complexity: O(log log u)
3. Min/Max: find the minimum/maximum value present in the tree.

   Time Complexity: O(1)

3 Kruskal Implementation

3.1 Simple implementation by sorting

Steps:

1. Initialize an empty set A
2. Create V trees, one containing each vertex
3. Sort the edges of the graph G(V,E) in non-descending order by weight
4. For each edge taken in non-descending order by weight, check whether the end points of the edge u, v belong to the same tree. If yes, the edge is discarded, else merge the 2 vertices into a single set and add the edge to A

Time Complexity: O(E Log E)

3.2 Using AVL Tree

Approach used to implement Kruskal's algorithm using AVL:

Used a hash table to store all the edges corresponding to the particular edge weight . Once the count of the edges goes down to zero , we delete that min weight from the hash.

If more than one edge with the same edge weight comes , then we add that edge in the list of that weight.

First we find the minimum edge weight using AVL Tree in O(logn) time and then we check in the map how many edges are present in the map corresponding to that edge weight . If there is only one edge present in the map corresponding to current min , then we simply delete that node from map as well as from AVL Tree.

If more than one edges are present corresponding to current min , then we take out one edge from the list corresponding to that current min and delete that edge from the list and update the list .

If the list corresponding to current min will become empty after taking out one edge(it means there is only one edge present corresponding to that edge weight) , then we delete that weight(current min) from list as well as from AVL Tree.

After finding out the current min , we find out the vertices of edges containing that minimum edge weight. Then we find out the parents of both the vertices and check if both are same .

If parents of both the vertices are not same , then we apply union on both the vertices and if they are same then we discard that edge otherwise it will make cycle .

Time Complexity: O(E log V)

3.3 Using vEB Tree

Used a hash table to store the multiple values of the same key. Once the count of the key goes down to zero, we delete the current min from the tree and update the min.

In vEB Tree , we are maintaining the weights of the edges as its elements and maintained a map to keep track of how many edges(pair of vertices) are present corresponding to edge weight .

First we find the minimum edge weight from vEB Tree in O(1) time and then we check in the map how many edges are present in the map corresponding to current min. If there is only one edge present in the map corresponding to current min, then we simply delete that node from map as well as from vEB Tree.

If more than one edges are present corresponding to current min , then we take out one edge from the list corresponding to that current min and delete that edge from the list and update the list .

If the list corresponding to current min will become empty after taking out one edge(it means there is only one edge present corresponding to that edge weight) , then we delete that weight(current min) from list as well as from vEB Tree.

After finding out the current min , we find out the vertices of edges containing that minimum edge weight. Then we find out the parents of both the vertices and check if both are same .

If parents of both the vertices is not same , then we apply union on both the vertices and if they are same then we discard that edge otherwise it will make cycle .

4 Implementation Challenges

1. Standard implementation of van Emde Boas tree doesn’t work when there are duplicate edge weights present in the tree. To handle duplicates, we used a hash table externally to store the edges of each weight.
2. Since vEB trees can find minimum element in log(log u) time while AVL Tree takes O(log n), so vEB trees should find MST in lesser time. However for smaller no of edge weights, the difference is not quite apparent. The difference only becomes apparent if no. of edges are quite large.

5 Results

The performance of finding MST using VEB and AVL is quite similar but as the no. of edges increase, vEB outperforms AVL. This is because (log log u) has a drastic effect only when u is very high.

6 Time Analysis

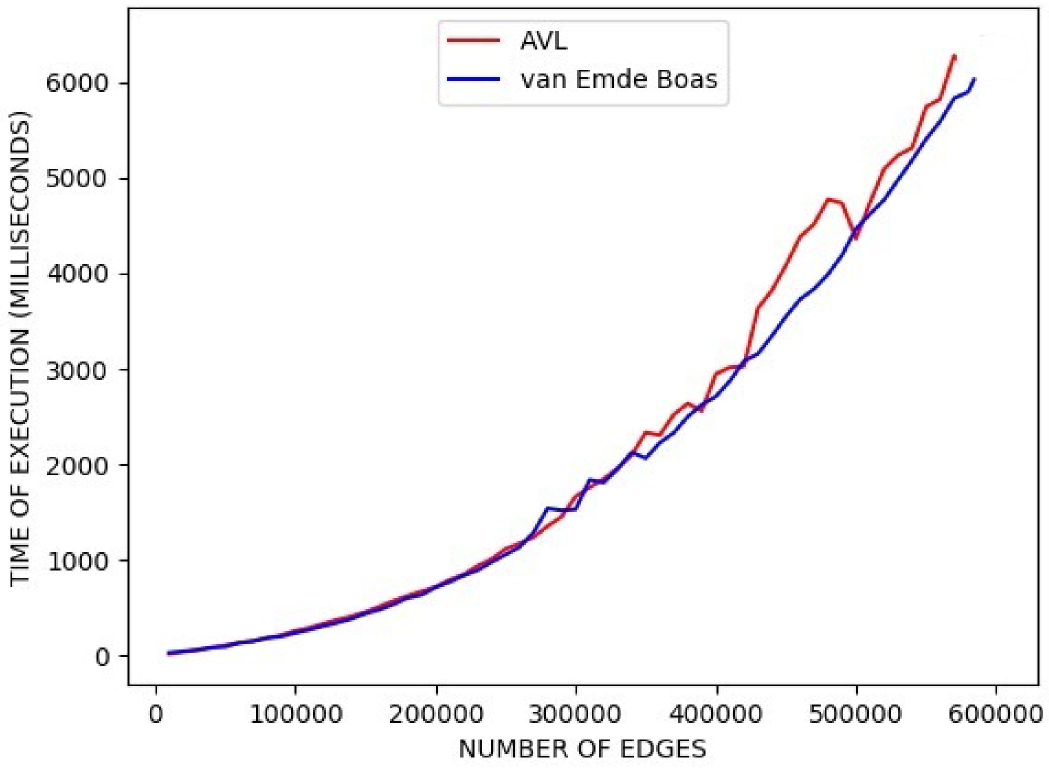

Figure 3: Comparison with random edge weights

7 End user documentation

7.1 To run kruskal algorithm using vEB:

1. Enter the directory
2. g++ kruskal_veb.cpp
3. ./a.out

Input format:

1. First line contains no. of vertices(V)

2. Second line contains no. of edges(E)
3. Next E lines contain edges in this format:
   <edge_weight vertex1 vertex2>

7.2 To run kruskal algorithm using AVL tree

1. Enter the directory
2. g++ kruskal_avl.cpp
3. ./a.out

Input format:

1. First line contains no. of vertices(V)
2. Second line contains no. of edges(E)
3. Next E lines contain edges in this format:
   <edge_weight vertex1 vertex2>

7.3 To run python script that generates random edge weights and compares time complexities of veb and avl

1. Enter the directory
2. python3 pygraph.py

8 References

1. MIT Lecture on van Emde Boas tree by Erik Demaine https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-videos/lecture-4-divide-conquer-van-emde-boas-trees/

2. Introduction to Algorithms by CLRS

   Chapter 20: van Emde Boas Trees

3. https://en.wikipedia.org/wiki/Van_Emde_Boas_tree

4. http://www.daimi.au.dk/gudmund/dynamicF04/vEB.pdf

5. http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/14/Slides14.pdf

6. http://www2.hawaii.edu/~nodari/teaching/s16/notes/notes10.pdf