Advanced Problem Solving

Project Report on “Implementation of van Emde Boas Trees and comparision of with Fibonacci heaps and Binomial heaps on calculating shortest path in a graph using Dijkstra single source shortest path algorithm”.

Snehashis Pal - 2018201072
Anuj Bansal - 2018201096
M.Tech 2018 Semester I
International Institute of Information Technology, Hyderabad

1. Overview

A *graph* is defined mathematically as a tuple $G = (V, E)$, with V being the set of vertices and E the set of edges, which is a relation of ($V \times V \times Z$), with Z being the set of integers representing the weight of an edge. A *path* between two vertices is defined as a sequence of vertices connected by edges. The goal of a single source shortest path algorith is to find a path of minimum weight starting from a source vertex and ending at all other vertices in the graph. *Dijkstra's* algorithm takes a greedy approach in finding the minimum path shortest to all vertices from a source vertex path given a connected undirected graph with weighted edges. The algorithm is outlined in the following sections. The goal of this project is to design efficient data-structures that help us to compute the *DSP* and compare them based on execution times. The focus will be on *Binomial Heaps, Fibonacci Heaps*, and *van Emde Boas Trees.*

The output will be in the form of multiple tables and/or graphs and will be on datasets taken from a very generalized graph structure i.e. vertices and edges, with no notion of any prior application domain. This is done to ensure that thecomparison is not biased towards any system specific implementation. Data-sets used will be random in nature and thus the output will represent results in a very generalized model of a system using graphs.

2. Theory

The *Dijkstra Single Source shortest path* (*DSP*) algorithm primarily uses a set of vertices as its underlying data structure. As a result a data structure representing a set of values, possibly duplicates, is required to be maintained and updated. Namely the following operations must be done efficiently to achieve good time complexities.

- *Insert* : Inserts a new value in the set.
- *Update* : Update the contents of and element in the set.
- *Get-Minimum* : Gets the minimum element in the set.
- *Delete* : Removes the element from the set.

Note that sometimes *Update* operation can be replaced with a delete operation followed by an insertion operation.

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:             // Initialization
6          dist[v] ← INFINITY                  // Unknown distance from source to v
7          prev[v] ← UNDEFINED                 // Previous node in optimal path from source
8          add v to Q                          // All nodes initially in Q (unvisited nodes)
9
10     dist[source] ← 0                        // Distance from source to source
11
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]    // Node with the least distance
14                                             // will be selected first
15         remove u from Q
16
17         for each neighbor v of u:           // where v is still in Q.
18             alt ← dist[u] + length(u, v)
19             if alt < dist[v]:               // A shorter path to v has been found
20                 dist[v] ← alt
21                 prev[v] ← u
22
23     return dist[], prev[]
```

Illustration 1: Dijkstra single source shortest path algorithm

The asymtotic complexities of the data structures used are stated in a tabular form. The time complexities of fibonacci and binomial heaps are based on the number of nodes in the graph, denoted as *N,* i.e. the number of elements in the set. However vEB-Trees represent sets as a bit vector with a 1 at index *i* denoting the presence of the *i*-th element in the set. This means that complexities are in terms of the universe of possible values represented by the set, denoted as *U*. Furthermore, to simplify the vEB-Tree structure *U* is always taken as a power of 2 (2^k). These values are always integers. Thus if the maximum number in the set is *x* we chose *U* as 2^k such than $x \leqslant 2^k$. Also updates on vEB-Trees are taken as a combination of an deletion followed by an insertion.

Data Structure/ Complexity	Binary Heap	Fibonacci Heap (Amortized)	Binomial Heap (Amortized)	vEB-Tree
Insert	O(log*N*)	O(1)	O(1)	O(loglog*U*)
Delete	O(log*N*)	O(log*N*)	O(log*N*)	O(loglog*U)*
Get Minimum	O(1)	O(1)	O(1)	O(1)
Update	O(log*N*)	O(1)	O(log*N*)	O(loglog*U)*

We deffer the discussion on fibonacci heaps and binomial heaps to references since the primary focus is on implementation of vEB-Trees. As evidenced by *DSP* algorithm we need to

- store the distance from the source node to all other nodes in a set,
- get the minimum from the set,
- update the other distances in the set based on the distance from the node corresponding to the minimum element
- remove the minimum element from the set and repeat from step 2.

Also, the distance of a multiple nodes from a source node can be same thus the data sturcture should have the ability to store and retrieve multiple values efficiently. We will first discuss a vEB – Tree structure for implementing sets and then extend it to include multisets very efficiently.

2.1 van Emde Boas(vEB) - Tree

As mentioned earlier vEB – Trees use a bit vector to represent sets from a universe of possible values *U*. The bit vector itself supports insertion, deletion, and hence update operation in O(1) time. Without any additional augmentation get min operation uses O(n) time in worst case. This is because a liner search in the bit vector may result in the element being found at the opposite end.

A vEB – Tree works by sumarizing information present in the bit-vector representation of sets. In, general we can summarize parts of the array, similar to that of a segment tree, which form the base of a tree (leaf nodes) using highier nodes. A summary in our context is an 'or' ing of the values present in the child nodes of the node which summarizes them. Intuitively, the summary is '1' when any of its child nodes are '1' meaning there is an element from the universe set, present in the part of the universe set which this node summarizes.

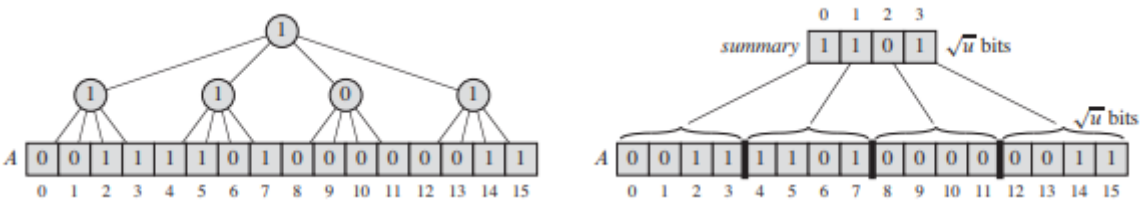


Illustration 2: The summary structure of a bit vector. A vEB - Tree uses a sqrt(U) bit vector as summary.

How does summarization help in decreasing the time to get minimum value from the set ? Eventhough we are not able to get exactly what elements are in a set, summarzed through a summary node we can determine if any elements exists in the summarized set. If no element exists we can skip the sublist entirely. This will help us in skipping over empty portions of the set, decreasing the time complexity of the system. We can recursively superimpose trees until we arrive at the root node which summarizes the entire set represented in the leaf nodes. This structure ca be then used to do get-min operations in reduced time. However nsertions and deletions will lead to changes in summary nodes and will have increased time complexity. Specifically, if a binary tree is created as a summary tree superimposed on a bit vector of size *U* we will require *logU* operations to get-min, insert and delete an element in the set. VEB – tree uses a tree of degree $\sqrt{u}$ to summarize a *U* size set.

2.1.1 vEB – Tree Structure

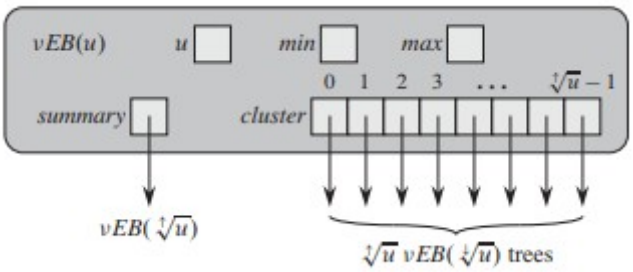


Illustration 3: Node structure

Each node in a vEB – tree structure summarizes a set of size 2^k represented as a bit vector using $\sqrt{(2^k)}$ summary vector. In general we divide the sub – vector into $\sqrt{(2^k)}$ parts, which is sumarized by $\sqrt{(2^k)}$ vector in the current node. These sub vectors are also represented as vEB – trees structures. Thus the leaf nodes of the tree together represents $U=2^k$, or the universe set. In the general case where *k* is not a power of 2 we divide the underlying vector into equal parts of the lower power of 2 and have summary vector of the highier power of 2. For example if *k = 3* i.e. $U = 2^3$

we divide the vector of *2* [3.] into 4(2^2) equal parts of 2 sub–nodes each, summarized by a 4 bit vector. We term the highier power of 2 as $\uparrow\sqrt{u}$ and the lower power as $\downarrow\sqrt{u}$. Thus, the vEB structure is a recursive one with top level nodes summarizing lower level nodes with a *cluster* of size $\uparrow\sqrt{u}$, each of which is also a vEB-node structure summarizing $\downarrow\sqrt{u}$ elements in the array. The contents of a node is summarized below

- *u* : universe size that this node encapsulates, any element found in this node or any child node must represent elements from the set 0 to *u – 1*. Thus, the root node covers the entire set, with child nodes covering parts of it.

- *min* : The minimum element in the set this node encapsulates or covers. Also, it is noted that this element does not exist in any of the child nodes of this node.

- *max* : The maximum element in the set this node covers. Unlike *min* this element is present in one of the sub - nodes unless it is same as min.

- *summary* : The summary of the clusters of this node. This, is also stored as a vEB tree with order equal to the number of summarized clusters.

- *cluster* : As explained a set of $\uparrow\sqrt{u}$ sub-nodes each a vEB tree.

Since in each node the set covered by it is represented as keys ranging from 0 to (*u* - 1) where *u* is the universe it covers we need to find a way to get the corresponding element in the overall set of elements. Consider an element in the *i* -th cluster of the root node. Then there are $(i-1)\downarrow\sqrt{u}$ possible elements with index less than it in the set. Also, if the position inside the *i*-th cluster is within the range 0 to $\downarrow\sqrt{u}-1$. So in order to find out which cluster a node belongs we implement the operation

$$high(x)=\lfloor x/\downarrow\sqrt{u}\rfloor$$

and to determine the corresponding position in the cluster we implement

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

Also, given the two previous function we can get the index of an element as

$$index(x,y)=x\downarrow\sqrt{u}+y$$

These three functions will help us in recursing down the vEB -tree from the root down to the desired node where the desired value will be stored. We first define the insertion and deletion algorithm considering singular values. The base condition is a vEB tree node of universe size 2. We chose this beacuse such a node only requires its *min* and *max* parameters to store which two element it stores.

2.1.2 *Algorithm* : Insertion into vEB Tree – insert (node,new_value)

1. *if current_node has no min value, assign min and max to new_value entered.*
2. *Else do*
    1. *if new_value < min of current_node*
        exchange new_value and min.
    2. *If u of current node > 2 i.e. not a base node*
        1. *get next_node = current_node.cluster(high(new_value))*
        2. *if next_node == NULL i.e first insertion*
            1. *update summary by insert(current_node.summary,high(new_value))*
        3. *call insert(current_node.cluster(high(new_ndoe)),low(new_value)) to insert into subcluster*
    3. *if entered value is > max set max = new_value*

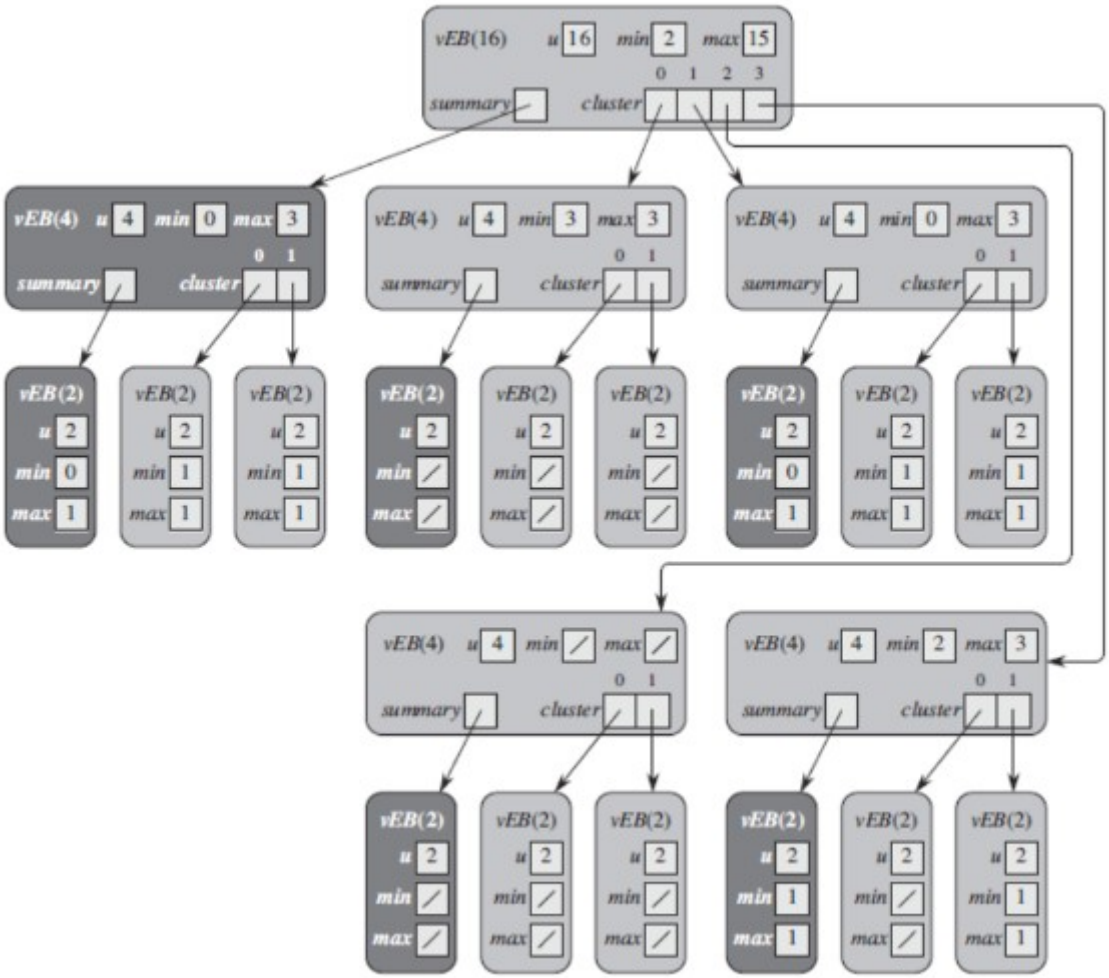


Illustration 4: vEB tree that stores the set {2, 3, 4, 5, 7, 14, 15}. Slashes indicate NULL values.

2.1.3 *Algorithm* : Deletion from vEB Tree – delete (node,old_value)

1. *if current_node only has min i.e 1 element delete set min = max = NULL*
2. *else if current_node.u = 2 i.e. base condition*
    1. *if old_value == min, delete min and set min to max*
    2. *else delete max and set max to min*
3. *else do*
    1. *if old_value == current_node.min*
        1. *find the cluster x where min next minimum is present using x = current_node.summary.min*
        2. *get the element w.r.t to the current universe set as next_min = index(x,cluster(x).min)*
        3. *set min, old_value = next_min, now we need to delete next min from the subclusters.*
    2. *Call delete(current_node.cluster(high(old_value)),low(old_value))*
    3. *if current_node.cluster(high(old_value)) is an empty cluster i.e. the last element in it was just deleted*
        1. *call delete(current_node.summary,high(old_value)) to delete summary entry*
        2. *if old_value == current.max i.e. we need to find new max*
            1. *get max summary cluster from current_node.summary.max*
            2. *if summary_max == NULL no items remain in any cluster*
                1. *set max = min as the only remaining element*
            3. *else get index of max w.r.t to current universe using index(summary_max,current_node.cluster(summary_max).max))*

4. *if old_value == max we need to find new max*
    1. *since old max was deleted and the corresponding cluster is not empty get max = index(high(old_value),current_node.cluster(high(old_value).max)*

Getting minimum is by simply accessing the minimum element of the root node. Updates are a combination of deletion and insertions.

2.2.4 Complexity Analysis

The recurrence relation corresponding to the insert and delete functions can be represented by

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

This follows from the fact that atmost one recursive call to the subcluster of size $\uparrow\sqrt{u}$ takes place. To solve this recurrence let $u=2^m$ thus,

$$T(2^m)\leqslant T(2^{\lceil m/2\rceil})+O(1)$$

$$\rightarrow T(2^m)\leqslant T(2^{(2m/3)})+O(1)$$

letting $S(m)=T(2^m)$ we get $S(m)\leqslant S(2m/3)+O(1)$ which by master method

$$S(m)=O(\log m)\rightarrow O(loglogu)$$

In addition the tree requres a *O(u)* time setup time an *O(u)* space to store all the values.

2.2 Augmented vEB – Tree

A vEB tree in its original state cannot hold multiple values required to store distances from the source that are identical. In this part we provide an augmented vEB-Tree that can hold multiple values without compromising on the asymtotic complexties provided.

The idea is to maintain a list of eqal values instead of a sungular value at each node. More specifically we maintain a doubly linked list of values so that both insertions and deletions can occur in *O(1)* time. Maintaining this property wil ensure that insertion into the list itself does not interfere with the time complexities of the vEB – tree itself. A doubly linked list can be created by encapsulating the value along with next and previous pointers in a separate *key_structure.*

- For insertions if a value i.e. a *key_structure* llist alread exits in *min / max* pointers add the new *key_structure* to the doubly inked list, otherwise assign it to the *min, max* pointers.
- Deletions requires that we have a reference to the *key_structure* being deleted. If the referenced *key_struct* is part of a doubly linked list we can simply delete the *struct* from the list in *O(1).* The only time we require a tree deletion is when the final remaining *key_structure* is being deleted, thus requiring the *min/max* to be updated.

3. Testing methodology and Performance comparision

We test the three algorithms under a common system, coded using c++, keeping in mind the following points.

- It is of utmost importance to maintain consistency in the testing environment, and not necessarilysubjecting each data-structure to best case scenarios.
- All data-structures will be subject to the same set of randomly generated test cases.
- All tests will be performed on an Intel i5-8250u with a pre-specified core frequency.

- GCC optimization level 3 (-O3 flag) will be used to compile all code to eliminate coding inefficiencies.
- The results of each test case will be averaged over a set of runs. This is to ensure outliers do not affect the final results.
- Graphs generated will be as diverse and random as possible.

3.1 Implementation overview

In our implementation we define the *key_structure* as a node structure in the dijkstra graph itself. Thus the vEB – tree holds lists of graph nodes with distances from the source as the entry in set defined by the vEB tree. It does so by maintaining a next and previous pointer for implementing doubly linked lists. This same graph node structure has all the properties of a typical graph node including a set of pointers to other graph nodes connected to it, forming its adjecency list. The same node structure is used accross all the three data structures to maintain consistancy although no directly by itself.

The reference to the nodes are encapsulated in nodes of specific tree types i.e. fibonacci node, binomial node and vEBTree node each holding some additional information to pertaining to he structure at hand like parent,child pointers for fibonacci heaps and min/max for vEB-trees. Each different type of tree and all its functionality are modularized into separate classes and files. We leave the calling function, calculating time and averaging of multiple runs to be done in main based on parameters passed as command line arguments.

The code for this project can be found on https://github.com/snehashispal1995/APS_project_2018.

3.2 Testing

- To test execution time we use *std::chrono* which is C++ library built to measure execution time. The time is calculated in microseconds.

- We generate a set of random graphs each having atleast one hamiltonian path. While not necessary, this additional criteria helps us to easily genrate simple undirected connected graphs an in much less time.

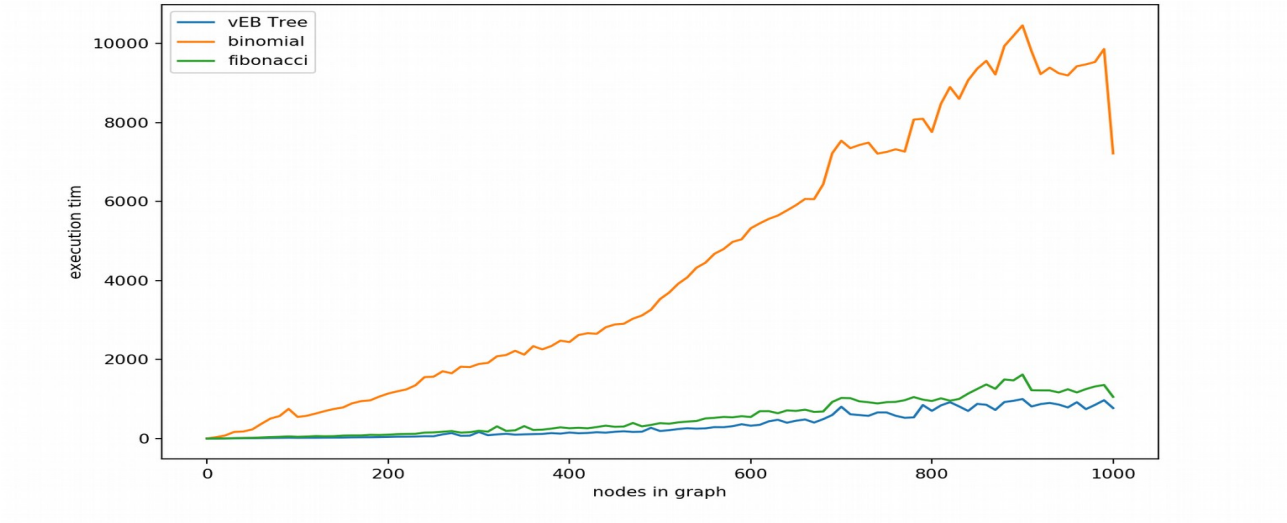
- To eliminate outliers we for each computation we perform an average of 20 runs for each tree data structure. This will hopefully level out the outliers due to fluctuations in core frequency.

- We test the three algorithm on graphs with various levels of *density* by varying the nodes to 10 to 1000 in steps of 10. We define *density* of a graph as the percentage of edges present of the total possible edges in a graph. For example if a graph has 100 nodes and 100 edges then the *density* of the graph is 100/4950 or 2 %. For our purposes we generate graphs with nodes ranging from 10 to 1000 with density at 10%, 25%, 50%, 75% and 90%.

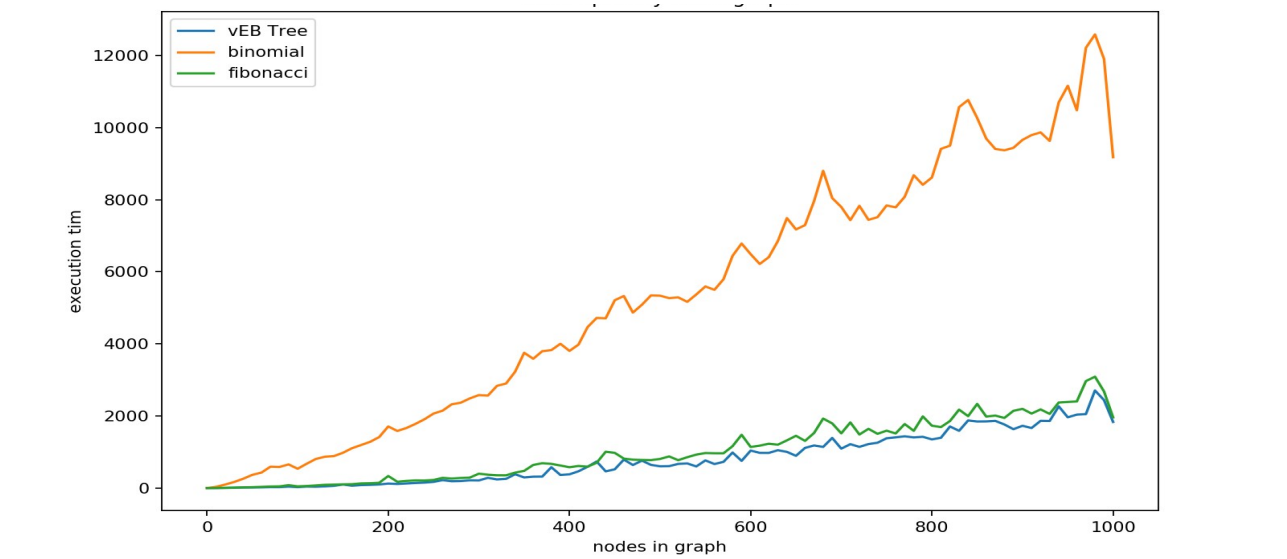
- In all tests we have taken the *u* value of vEB-Tree to be the most optimal i.e. the least power of 2 capable of storing the maximum possible value of the distance from the source node. For example if *w* is the max edge weight in a graph with *N* nodes we choose a power of 2, say *k*, such that $2^k > w(N-1)$ i.e. the max possible distance.

3.3 Results and Analysis

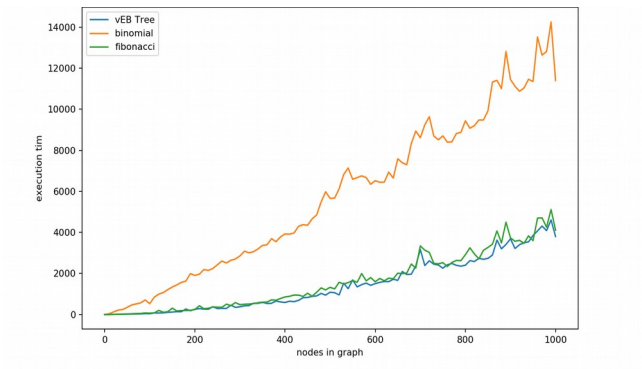
We plot the running time obtained in microseconds into a graph with x-axis as nodes and y-axis as execution time. We take graphs of different densities and compare the effectiveness of the three data structures on each one of them. The graphs themselves range from 10 to 5000 nodes. This will give is reasonable data through which we can estimate the performance of each data structure and compare them with one another. We avoid exact measurements as they are system dependent and a more consistent, standardized system will give more measurable, repliable results than the one used by us for testing.



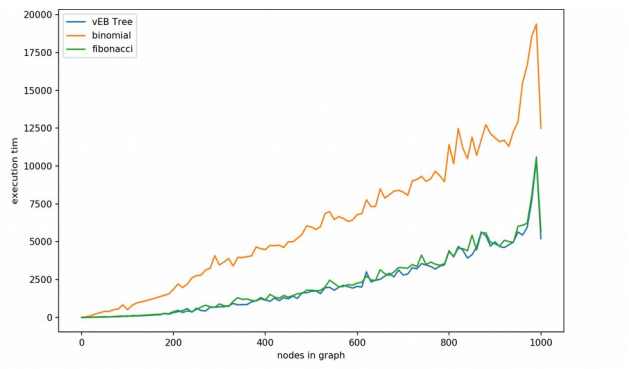
Max weight of edge = 10 with density of 10%



Max weight of edge = 20 with density of 25%



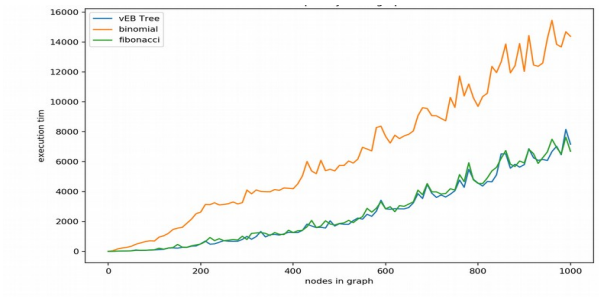
Max weight of edge = 25 density 50 %



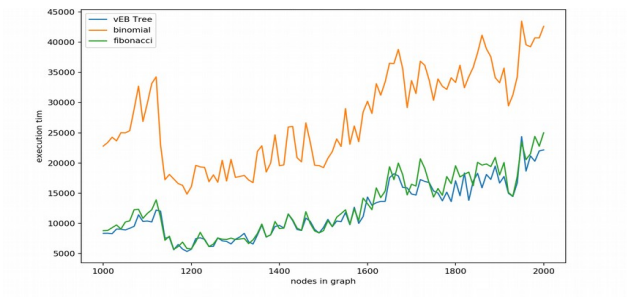
Max weight of edge = 30 density 50%

From the graphs both we can conclude that veb trees performance is similar to that of fibonacci heaps. Also is seems that running times obtained seems to depend very much on the type of graphs that are randomly being generated, as graph with highier node count show larger variations in

execution times. As we go to highier density number the time taken by fibonacci heaps ad vEB trees are more close together. As expected binomial heap gives the worst execution time owing to its highier update complexity.



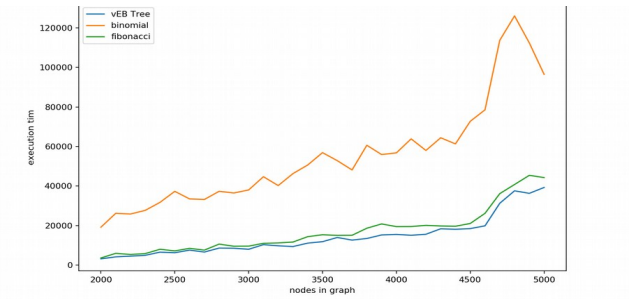
Max edge weight = 50 density = 75%



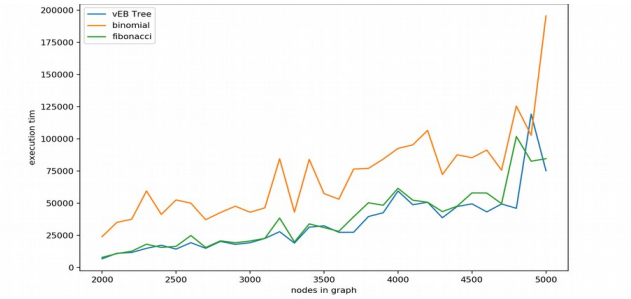
Max edge weight = 50 density = 90%

The 90% density graph has a high number of variations in the execution time as excpected due to larger variations in the randomness of the graph. Some of the variations may be attributed to core frequency fluctuations. It seems as though in all three data structures a change of a few nodes is causing a large shift in the runtimes with a sharp increase followed by a sharp decrease. Also by this time we have increased the max wight of an edge thereby increasing the universe space requirements of the vEB-Tree. This has also increased the housekeeping summary information requirements of vEB-Tree thereby increasing execution time. For the final graph we take a larger span of nodes i.e. 1000 to 2000 and find similar fluctuations in executio times.

Moving over to even highier order graphs. Like previous two graphs all subsequent graphs have max edge weight = 50.

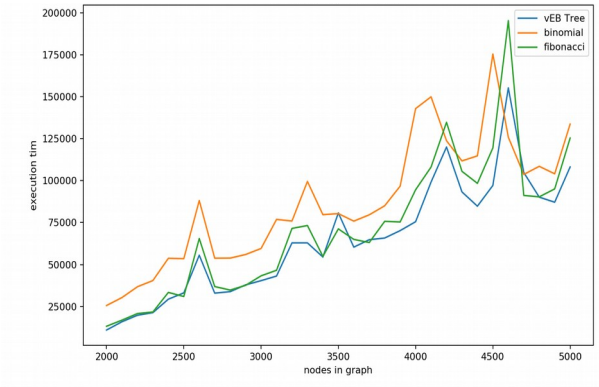


density 10%

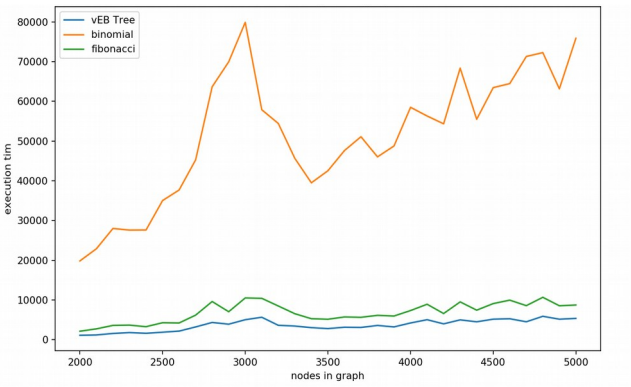


density 25%

From the graphs above it seems that as we increase the density of the graph the execution times get closer together. The following graphs seems to confirm this theory.



density 50%



density 1%

4. Conclusions

We see that the vEB-Tree provide an time efficient solution to sets and multiset problem. They beat out both binomial heaps and fibonacci heaps in must of the tests done here. They are pirtucularly good with sparse graphs. Even at highier density numbers the can still be preferred over the other data structures if the extra space requirements is not much of an issue. Fibonacci heaps provides a excellent alternative with runtimes that are extremely close to vEB-Tree while simultaneously not being restricted by the universe space of keys. In applications with a large universe space of keys or with non-discrete keys fibonacci heaps provide excellent runtimes while being space efficent. Binomial heaps perform the worst of the three data structures showcased, although an increase in density does not seem to affect its performance to the same extent as it affect the other two data structure. Hence graphs with high degree can avoid the extra complexity of a fibonacci heap or a vEB-Tree if binomial heaps are sufficient.

5. References

- Fibonacci Heaps, Chapter 19, Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- van Emde Boas Trees, Chapter 20, Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- Binomial Heap. Geeksforgeeks https://www.geeksforgeeks.org/binomial-heap-2
- Djkistra. Wikipedia.org https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm