IIIT Hyderabad

computer science and engineering

Advanced Problem Solving

Study and implementation of van Emde Boas tree with application to Dijkstra and comparison wrt Binomial Heap

Prof. Dr.Avinash Sharma

TA: Dhawal Jain

Contributers:
Soumalya Bhanja - 2019201004
Sagnik Gupta - 2019201003

1 ABSTRACT

Dijkstra's Algorithm is used to calculate the shortest path between one node and every other node in a graph. In this project we aim to improve the time complexity of Dijkstra's single source shortest path algorithm using advanced data structures like van Emde Boas trees and Binomial Heaps. van Emde Boas trees support various operations of a Priority Queue in minimal worst case time. So, implementing Priority Queue in Dijkstra using van Emde Boas tree helps in improving the time complexity. Similarly Binomial heaps provide faster union or merge operations thus improving the time complexity. We present a comparison between the two approaches.

2 VAN EMDE BOAS TREE

Van Emde Boas trees support each of the operations INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR in O(lg lg n) time. Thereby giving an overall improvement in time complexity for single source shortest path algorithm. The keys to be stored in the tree are integers of range 0 to n-1, no duplicates are allowed. The functionality of priority queue can be implemented using Van Emde Boas trees.

Preliminary approaches

We call the set 0, 1, 2, ... n-1 the universe of values that can be stored and u the universe size.We assume throughout that u is an exact power of 2, i.e.,

$$u = 2^k$$

for some integer $k \geq 1$

Bit Vector

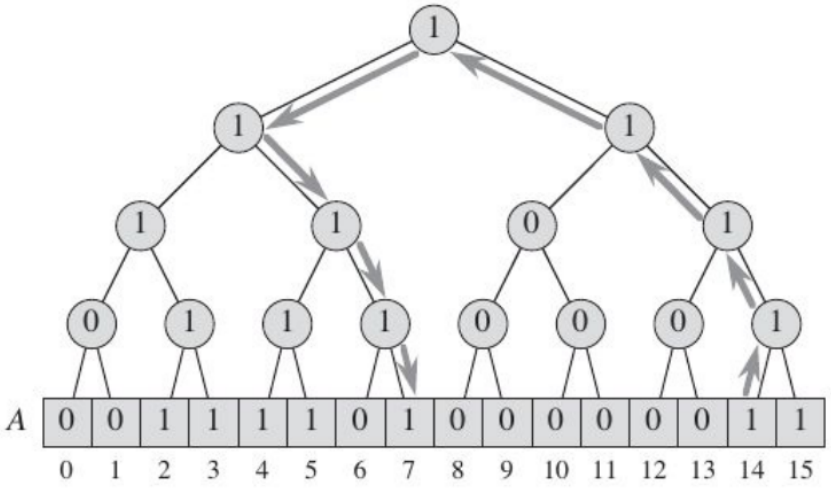
A bit vector is an array of bits of length U. Represents a set of elements with insertions, deletions, and look ups each taking time O(1) 1)To insert x, set the bit for x to 1 2)To delete x, set the bit for x to 0 3)To lookup x, check whether the bit for x is 1. Space usage is (U). The min, max, predecessor, and successor operations on bit vectors can be extremely slow. Run time will be (U) in the worst case.

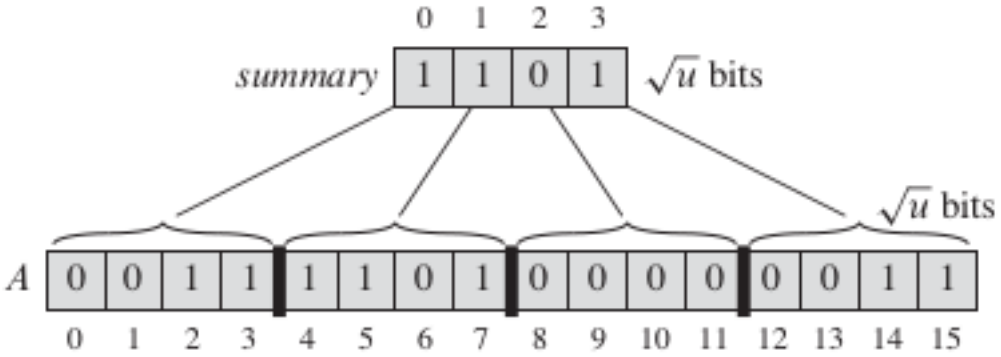
1	0	0	0	1	1	0	0
00100010	00000000	00000000	00000000	00000100	11110111	00000000	00000000

Tiered Bit Vectors

We can put a summary structure on top of our bit vector. Break the universe u into O(u/k) blocks of size k. Summary structure is an auxiliary bit vector of size O(u/k) that stores which blocks are nonempty. Superimpose a binary tree structure

on top of the bit vector. Since the height is log u and each operation makes at most one pass up the tree and one pass down,the operation time is O(log u) To perform lookup(x) in this structure, check the bit vector to see if x mod $u^{1/2}$ is present





A recursive structure

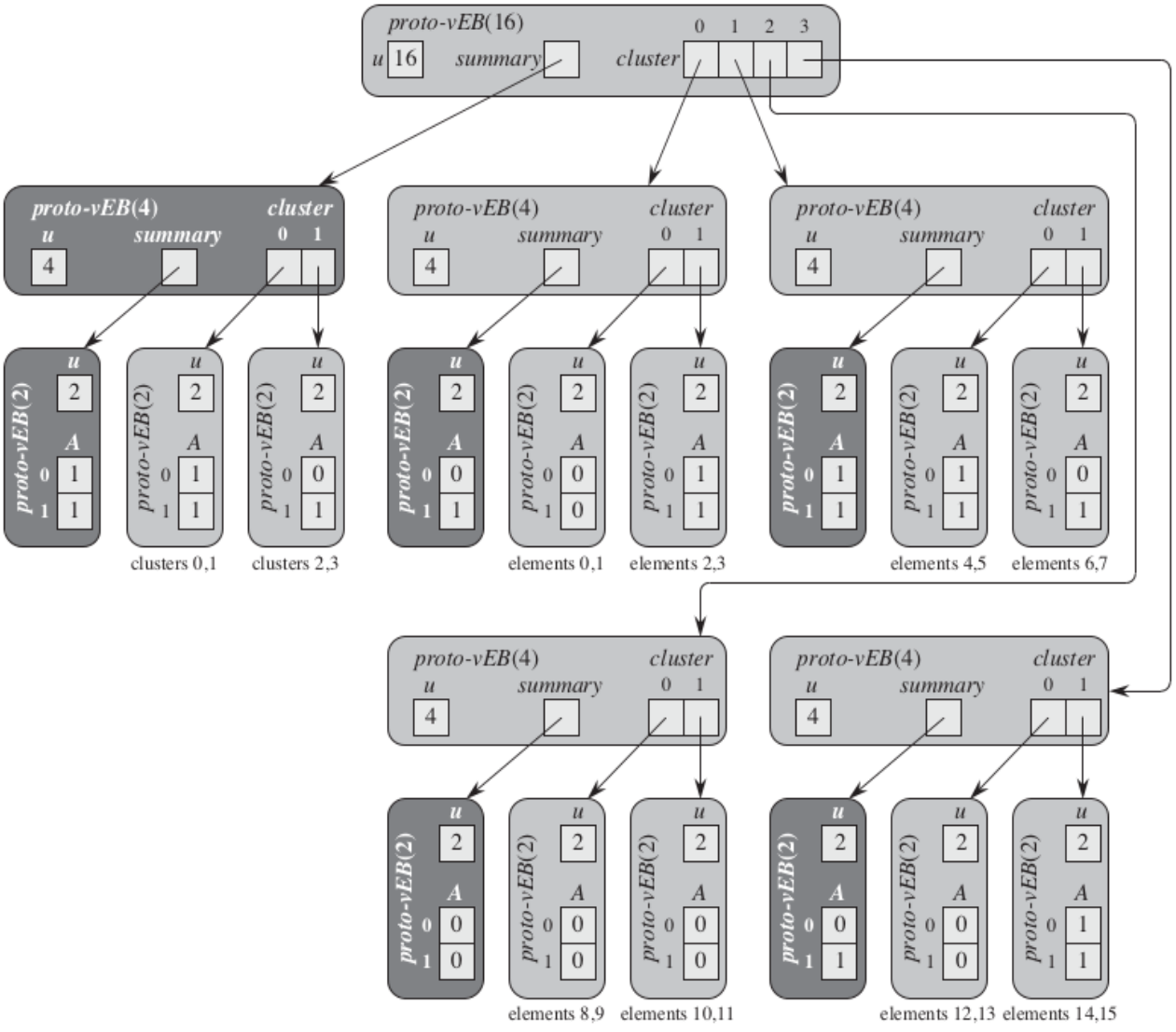
We will use the idea of superimposing a tree of degree $\sqrt[2]{u}$ on top of a bit vector, but shrink the universe size recursively by a square root at each tree level. The $\sqrt[2]{u}$ items on the first level each hold structures of $\sqrt[4]{u}$ items, which hold structures of $\sqrt[8]{u}$ items, and so on, down to size 2. Assume for simplicity now that $u = 2^{2k}$ for some integer k.

Our aim is to achieve time complexity:
T(u)=T($\sqrt{}$u)+O(1)=T($u^{1/2^k}$)+O(k)=O(log log u)

Since $u^{1/2^k}$=2 implies 2^k=log u, which gives k=log log u.On the top level of the tree, log u bits are needed to store the universe size, and each level needs half the bits of the previous level.
Define: high(x) = floor(x/$\sqrt{}$u), the most significant(log u)/2 bits of x gives the number of x's cluster. low(x) =x mod$\sqrt{}$u, the least significant(log u)/2bits of x gives x's position within its cluster. Index(x, y)=x$\sqrt{}$u+y, builds an element number, where x=index(high(x),low(x))

With above structure, we have optimized max(), min(), successor() and predecessor() to time complexity O(log u).

min() : Start with root and traverse to a leaf using following rules. While traversing, always choose the leftmost child, i.e., see if left child is 1, go to left child, else go to right child. The leaf node we reach this way is minimum. Since we travel across height of binary tree with u leaves, time complexity is reduced to O(log u).
max() : Similar to min(). Instead of left child, we prefer right child.
successor(x) : Start with leaf node indexed with x and travel towards the root until we reach a node z whose right child is 1 and not on the same path covered up until now. Stop at z and travel down to a leaf following the leftmost node with value 1.
predecessor() : This operation is similar to successor. Here we replace left with right and right with left in successor().
find() is still O(1) as we still have binary array as leaves. insert() and delete() are now O(Log u) as we need to update internal nodes. In case of insert, we mark the corresponding leaf as 1, we traverse up and keep updating ancestors to 1 if they were 0.


proto-vEB(16)
u 16
summary
cluster 0 1 2 3
proto-vEB(4)
u 4
summary
cluster 0 1
proto-vEB(2)
u 2
A 0 1 1
clusters 0,1
clusters 2,3
elements 0,1
elements 2,3
elements 4,5
elements 6,7
elements 8,9
elements 10,11
elements 12,13
elements 14,15

The van Emde Boas tree (vEB tree)

The proto-vEB structure of the previous section is close to what we need to achieve O(lg lg u) running times. It falls short because we have to recurse too many times in most of the operations.
A vEB tree stores its minimum element as min and its maximum as max, which help us as follows:
1.MINIMUM and MAXIMUM operations do not need to recurse.
2.SUCCESSOR can avoid a recursive call to determine if the successor of x lies within its high(x), because x's successor lies within its cluster iff x¡max of its cluster.
3.INSERT and DELETE will be easy if both min and max are NIL, or if they are equal.
4.If a vEB tree is empty, INSERT takes constant time just by updating its min and max. Similarly, if it has only one element DELETE takes constant time.
Time will be given by $T(u) \leq T(\sqrt{u}) + O(1)$, which also solves to O(log log u).

The minimum and maximum elements

Because we store the minimum and maximum in the attributes min and max, two of the operations are one-liners, taking constant time:

```
VEB-TREE-MINIMUM(V)
1  return V.min

VEB-TREE-MAXIMUM(V)
1  return V.max
```

The minimum and maximum elements

The procedure VEB-TREE-MEMBER(V, x) has a recursive case like that of PROTO-VEB-MEMBER , but the base case is a little different. We also check directly whether x equals the minimum or maximum element. Since a vEB tree doesn't store bits as a proto-vEB structure does, we design VEB-TREE-MEMBER to return TRUE or FALSE rather than 1 or 0.

```
VEB-TREE-MEMBER(V, x)
1  if x == V.min or x == V.max
2       return TRUE
3  elseif V.u == 2
4       return FALSE
5  else return VEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

The Successor

The procedure PROTO-VEB-SUCCESSOR(V, x) could make two recursive calls: one to determine whether x's successor resides in the same cluster as x and, if it does not, one to find the cluster containing x's successor.Because we can access the maximum value in a vEB tree quickly, we can avoid making two recursive calls, and instead make one recursive call on either a cluster or on the summary, but not on both.

```
VEB-TREE-SUCCESSOR(V, x)
 1  if V.u == 2
 2       if x == 0 and V.max == 1
 3            return 1
 4       else return NIL
 5  elseif V.min ≠ NIL and x < V.min
 6       return V.min
 7  else max-low = VEB-TREE-MAXIMUM(V.cluster[high(x)])
 8       if max-low ≠ NIL and low(x) < max-low
 9            offset = VEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10            return index(high(x), offset)
11       else succ-cluster = VEB-TREE-SUCCESSOR(V.summary, high(x))
12            if succ-cluster == NIL
13                 return NIL
14            else offset = VEB-TREE-MINIMUM(V.cluster[succ-cluster])
15                 return index(succ-cluster, offset)
```

The Predecessor

The VEB-TREE-PREDECESSOR procedure is symmetric to the VEB-TREE-SUCCESSOR procedure, but with one additional case: Lines 13–14 form the additional case. This case occurs when x's predecessor, if it exists, does not reside in x's cluster.

```
VEB-TREE-PREDECESSOR(V, x)
 1  if V.u == 2
 2      if x == 1 and V.min == 0
 3          return 0
 4      else return NIL
 5  elseif V.max ≠ NIL and x > V.max
 6      return V.max
 7  else min-low = VEB-TREE-MINIMUM(V.cluster[high(x)])
 8      if min-low ≠ NIL and low(x) > min-low
 9          offset = VEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
10          return index(high(x), offset)
11      else pred-cluster = VEB-TREE-PREDECESSOR(V.summary, high(x))
12          if pred-cluster == NIL
13              if V.min ≠ NIL and x > V.min
14                  return V.min
15              else return NIL
16          else offset = VEB-TREE-MAXIMUM(V.cluster[pred-cluster])
17              return index(pred-cluster, offset)
```

This extra case does not affect the asymptotic running time of VEB-TREE-PREDECESSOR when compared with VEB-TREE-SUCCESSOR , and so VEB-TREE-PREDECESSOR runs in O(lg lg u) worst-case time.

Insert

PROTO-VEB-INSERT made two recursive calls: one to insert the element and one to insert the element's cluster number into the summary. The VEB-TREE-INSERT procedure will make only one recursive call. When we insert an element, either the cluster that it goes into already has another element or it does not. If the cluster already has another element, then the cluster number is already in the summary, and so we do not need to make that recursive call. If the cluster does not already have another element, then the element being inserted becomes the only element in

the cluster, and we do not need to recurse to insert an element into an empty vEB tree:

Time is $T(u) \leq T(\sqrt{u}) + O(1) = O(\log \log u)$, since inserting into an empty tree is $O(1)$.

```
VEB-EMPTY-TREE-INSERT(V, x)
1  V.min = x
2  V.max = x

VEB-TREE-INSERT(V, x)
 1  if V.min == NIL
 2      VEB-EMPTY-TREE-INSERT(V, x)
 3  else if x < V.min
 4          exchange x with V.min
 5      if V.u > 2
 6          if VEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
 7              VEB-TREE-INSERT(V.summary, high(x))
 8              VEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
 9          else VEB-TREE-INSERT(V.cluster[high(x)], low(x))
10      if x > V.max
11          V.max = x
```

Binary Heap

A binary heap is a heap data structure that takes the form of a binary tree. It follows the following two properties:
Shape property: a Binary Heap is a complete binary tree.
Heap property: A Binary Heap is either Min Heap or Max Heap.
Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log n) time.

A Binary Heap is a Binary Tree with following properties.
1) It's a complete tree (All levels are completely filled except the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

Applications of Binary Heap

1) Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
2) Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.
4) Many problems can be efficiently solved using Heaps. See following for example.
a) K'th Largest Element in an array.
b) Sort an almost sorted array.
c) Merge K Sorted Arrays.

Binary Heap Operations

1) getMin() or getMax() : It returns the root element of Min Heap(or Max Heap). Time Complexity of this operation is O(1).
2) extractMin() or extractMax(): Removes the minimum(or max) element from

MinHeap(or maxheap). Time Complexity of this Operation is O(Logn) as this operation needs to maintain the heap property (by calling heapify()) after removing root.
3) decreaseKey(): Decreases value of key. The time complexity of this operation is O(log n). If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
4) insert(): Inserting a new key takes O(Logn) time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything(for mean heap, vice versa for max heap). Otherwise, we need to traverse up to fix the violated heap property.
5)delete(): Deleting a key also takes O(Logn) time. We replace the key to be deleted with minum infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove the key.

BINOMIAL HEAP

The main application of Binary Heap is as implement priority queue. Binomial Heap is an extension of Binary Heap that provides faster union or merge operation together with other operations provided by Binary Heap. A Binomial Heap is a collection of Binomial Trees A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one as leftmost child or other. A Binomial Tree of order k has following properties. a) It has exactly 2k nodes. b) It has depth as k. c) There are exactly kCi nodes at depth i for i = 0, 1, . . . , k. d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right. Binomial Heap: A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at most one Binomial Tree of any degree. Binary Representation of a number and Binomial Heaps

Binomial Heap Operations

1) BINOMIAL-HEAP-MINIMUM(H): The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. Since a binomial heap is heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number is at most lg n + 1, saving the current minimum in min and a pointer to the current

minimum in y. Because there are at most lg n + 1 roots to check, the running time of BINOMIAL-HEAP-MINIMUM is O(lg n).

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5      do if key[x] < min
6            then min ← key[x]
7                 y ← x
8         x ← sibling[x]
9  return y
```

2) BINOMIAL-HEAP-UNION(H1,H2): Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.
A) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

B) After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots. ——Case 1: Orders of x and next-x are not same, we simply move ahead. In following 3 cases orders of x and next-x are same. ——Case 2: If the order of next-next-x is also same, move ahead. ——Case 3: If the key of x is smaller than or equal to the key of next-x, then make next-x as a child of x by linking it with x. ——Case 4: If the key of x is greater, then make x as the child of next. The running time of BINOMIAL-HEAP-UNION is O(1g n), where n is the total number of nodes in binomial heaps H1 and H2.

```
BINOMIAL-LINK(y,z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z]+1
```

```
BINOMIAL-HEAP-UNION(H1, H2)
 1  H ← MAKE-BINOMIAL-HEAP()
 2  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
 3  free the objects H1 and H2 but not the lists they point to
 4  if head[H] = NIL
 5     then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x ≠ NIL
10        do if (degree[x] ≠ degree[next-x]) or
                    (sibling[next-x] ≠ NIL
                    and degree[sibling[next-x]] = degree[x])
11              then prev-x ← x                          ▷ Cases 1 and 2
12                   x ← next-x                          ▷ Cases 1 and 2
13              else if key[x] ≤ key[next-x]
14                      then sibling[x] ← sibling[next-x]        ▷ Case 3
15                           BINOMIAL-LINK(next-x, x)            ▷ Case 3
16                      else if prev-x = NIL                     ▷ Case 4
17                              then head[H] ← next-x            ▷ Case 4
18                              else sibling[prev-x] ← next-x    ▷ Case 4
19                           BINOMIAL-LINK(x, next-x)            ▷ Case 4
20                           x ← next-x                          ▷ Case 4
21           next-x ← sibling[x]
22  return H
```

4) BINOMIAL-HEAP-INSERT(H,x): The following procedure inserts node x into binomial heap H, assuming of course that node x has already been allocated and

key[x] has already been filled in. The procedure simply makes a one-node binomial heap H' in O(1) time and unites it with the n-node binomial heap H in O(1g n) time.

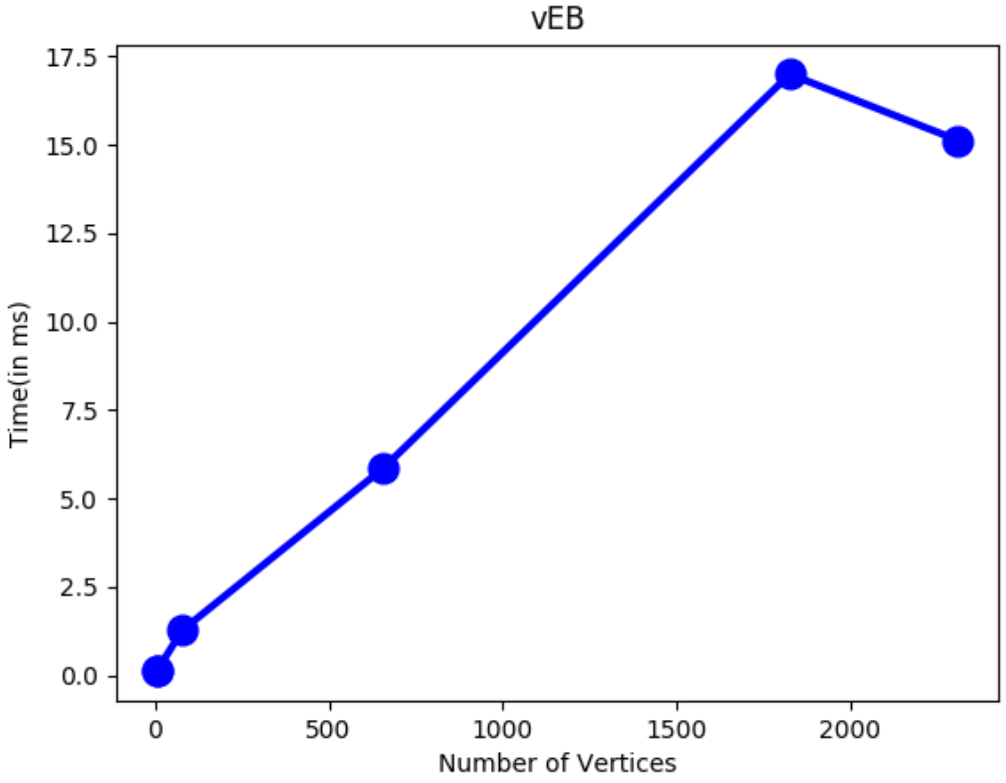
```
BINOMIAL-HEAP-INSERT(H,x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H,H')
```

5) BINOMIAL-HEAP-EXTRACT-MIN(H): The following procedure extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node. Since each of lines 1-4 int the following procedure takes O(lg n) time if H has n nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in O(lg n) time.

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  find the root x with the minimum key in the root list of H,
and remove x from the root list of H
2  H' ← MAKE-BINOMIAL-HEAP()
3  reverse the order of the linked list of x's children,
and set head[H'] to point to the head of the resulting list
4  H ← BINOMIAL-HEAP-UNION(H,H')
5  return x
```

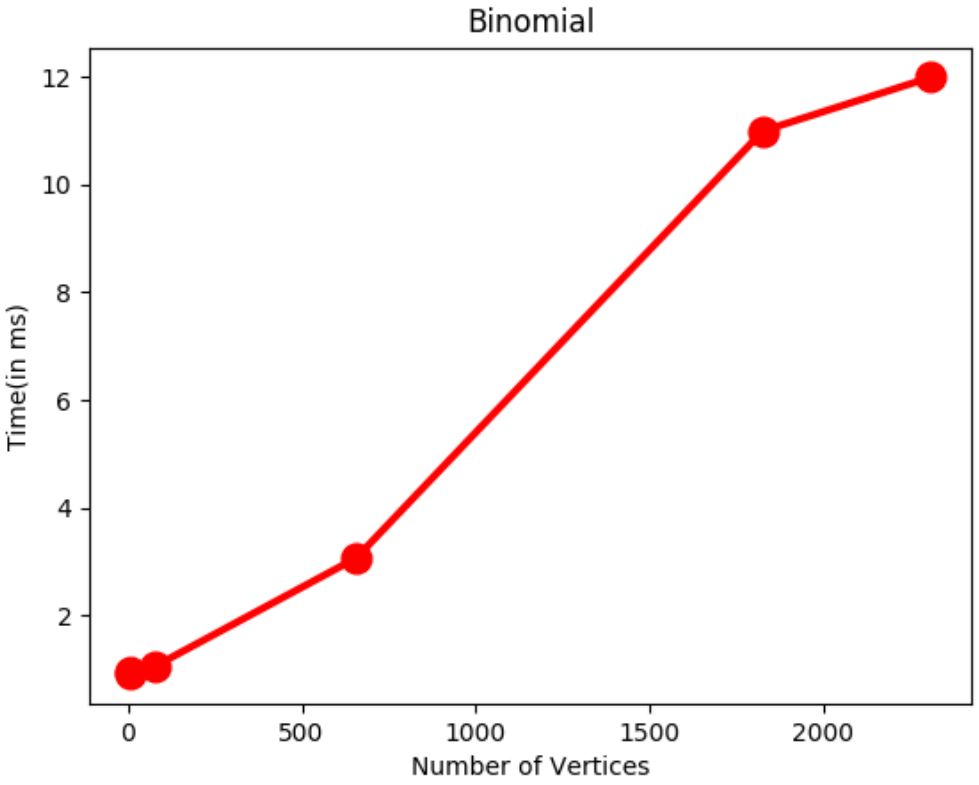
DJISKTRA USING VAN EMDE BOAS TREE

Priority queue has been replaced with the functionalities of vEB tree. Initially the source vertex is added to the empty tree. The Dijkstra implementation has been done using INSERT and SUCCESSOR operations. An unordered map has been maintained which stores the distance along with the list of vertices which are reachable within that distance at that point of time. The vEB structure stores the distance of vertices from the source. INSERT operations are used to add new distances into the vEB tree. After successful relaxation operations of vertices, the new distances are added into the tree. The SUCCESSOR operations is used to iterate over the various values of distances. During the iteration operation, the distances are relaxed until there are no more SUCCESSOR left within the vEB Tree. The nodes and the final values of the distances upto the nodes give the final values of shortest paths till that node.

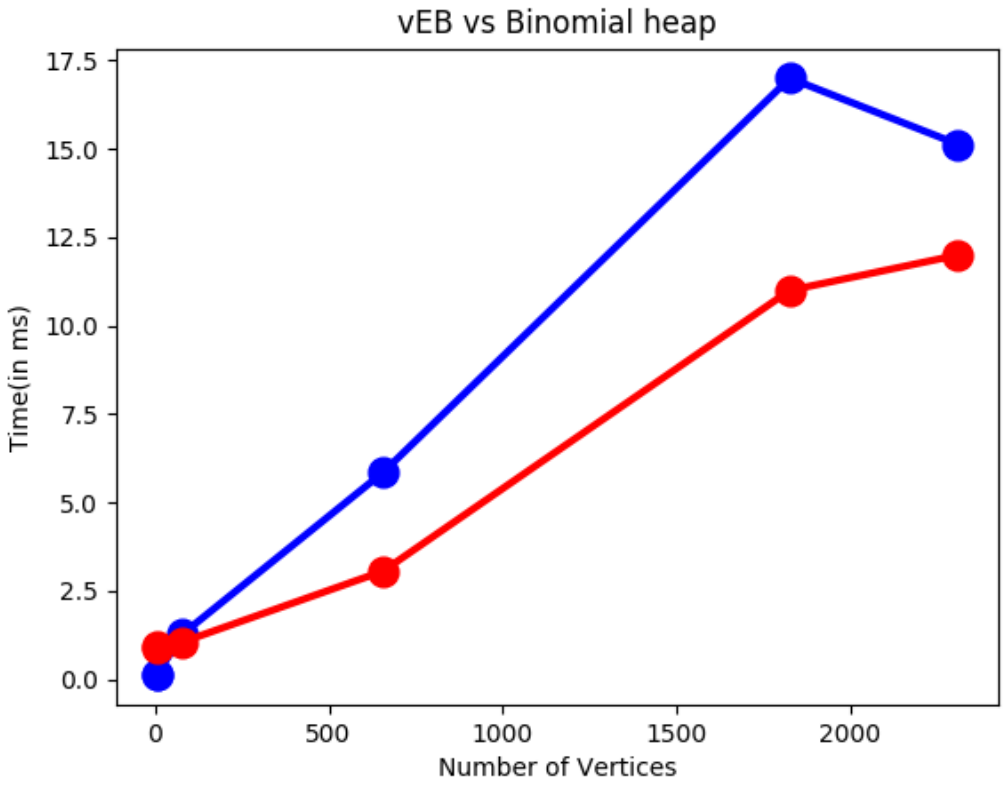


DJISKTRA USING BINOMIAL HEAP

Binomial heap has been used in place of priority queue for implementation of single source shortest path algorithm. The Dijkstra implementation has been done with INSERT, FIND-MIN and EXTRACT-MIN operations of the Binomial Heap. Initially the source vertex is added to the Binomial Heap. An unordered map has been maintained which stores the distance along with the list of vertices which are reachable within that distance at that point of time. The Binomial Heap stores the distance of vertices from the source. INSERT operations are used to add new distances into the Binomial Heap. After successful relaxation operations of vertices, the new distances are added into the Binomial Heap. The minimum distance is found out in each iteration using FIND-MIN, and the EXTRACT-MIN operation is used to remove minimum values from the Binomial Heap. The iteration continues upto the point when Heap becomes empty. In each iteration, relaxation operation is performed. The nodes and the final values of the distances upto the nodes give the final values of shortest paths till that node.



Comparison



Conclusion

The main advantage of vEB tree structure is the improvement in overall time complexity of priority queue. From O(E log V) to O(E log log V) for vEB trees in single source shortest path algorithm.
Binomial heap provides an overall time complexity of O(E log V) for single source shortest path algorithm.

REFERENCES

- **Introduction to Algorithms, CLRS**
- **MIT OpenCourseWare video** https://www.youtube.com/watch?v=hmReJCupbNU
- **Wikipedia** https://en.wikipedia.org/wiki/VanEmdeBoastree
- **Wikipedia** https://en.wikipedia.org/wiki/Binomialheap
- **Wikipedia** https://en.wikipedia.org/wiki/Dijkstra'salgorithm
- **Wikipedia** https://en.wikipedia.org/wiki/Binaryheap