Dijkstra using vEB Tree

Prof. Venkatesh Chopella

Prof. Vikram Pudi

TA: Rajat Agarwal

Aishwary Dewangan (2018202016)

Vishal Chugh (2018202021)

GIT Repository: https://github.com/aishwr/Dijkstra-using-van-Emde-Boas-Tree

Abstract

van Emde Boas trees support various operations of a Priority Queue such as SEARCH , INSERT , DELETE , MINIMUM, MAXIMUM , SUCCESSOR, and PREDECESSOR in O(lg lg n) time worst case time. The hitch is that the keys must be integers in the range 0 to n -1, with no duplicates allowed. So, implementing Priority Queue in Dijkstra using vEB tree helps in improving the time complexity from O(E log V) to O(E log log V).

Preliminary Approaches for vEB Tree

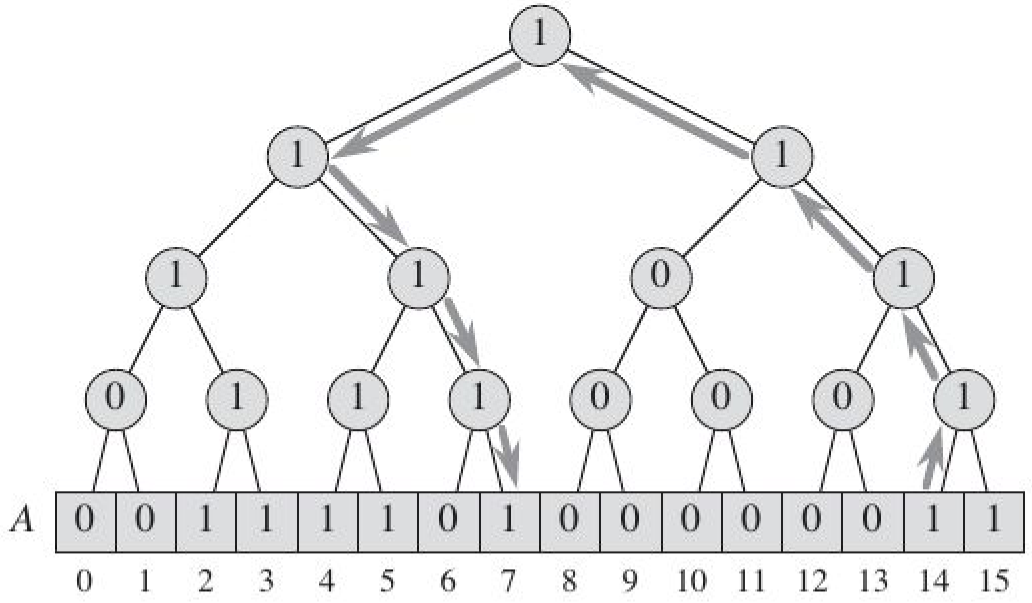
n denotes the number of elements currently in the set and **u** denotes the range of possible values. We call the set {0, 1, 2, … ,u -1} the **universe** of values that can be stored and u the universe size.

We assume throughout that u is an exact power of 2, i.e., u = 2^k for some integer k >= 1.

- **Using Bit Vector:** To store a dynamic set of values from the universe {0, 1, 2, …, u- 1} we maintain an array A[0, …, u-1] of u bits. The entry A[x] holds a 1 if the value x is in the dynamic set, and it holds a 0 otherwise. Although we can perform each of the INSERT, DELETE, and MEMBER operations in O(1) time with a bit vector, the remaining operations—MINIMUM , MAXIMUM, SUCCESSOR, and PREDECESSOR, each take $\Theta($ u) time in the worst case.
- **Superimposing a Binary Tree structure:** The entries of the bit vector form the leaves of the binary tree, and each internal node contains a 1 if and only

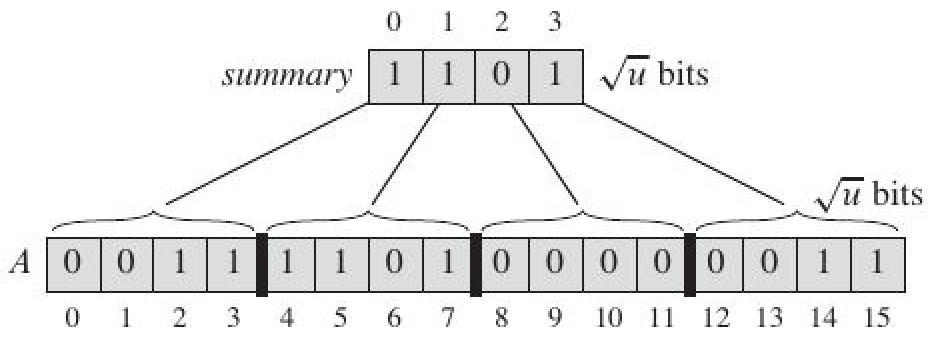
if any leaf in its subtree contains a 1. In other words, the bit stored in an internal node is the logical-or of its two children.

- To find the **minimum** value in the set, start at the root and head down toward the leaves, always taking the leftmost node containing a 1.
- To find the **successor** of 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**. Then head down through node z, always taking the leftmost node containing a 1 (i.e., find the minimum value in the subtree rooted at the right child z).



The above figure represents a binary tree of bits superimposed on top of a bit vector representing the set {2, 3, 4, 5, 7, 14, 15} when u=16. Each internal node contains a 1 if and only if some leaf in its subtree contains a 1. The arrows show the path followed to determine the predecessor of 14 in the set.

- **Superimposing a tree of constant height:** Instead of superimposing a binary tree on top of the bit vector, we superimpose a tree of degree $\sqrt{u}$. The height of the resulting tree is always 2. Each internal node stores the logical-or of the bits within its sub-tree, so that the $\sqrt{u}$ internal nodes at depth 1 summarize each group of $\sqrt{u}$ values. An array **summary**[0... $\sqrt{u}$-1], where summary[i] contains a 1 if and only if the subarray A[i $\sqrt{u}$...(i+1) $\sqrt{u}$-1] contains a 1. We call this $\sqrt{u}$-bit subarray of A the ith **cluster**. For a given value of x, the bit A[x] appears in cluster number floor(x/ $\sqrt{u}$).



- To find the minimum (maximum) value, find the leftmost (rightmost) entry in summary that contains a 1, say summary[i], and then do a linear search within the ith cluster for the leftmost (rightmost) 1.
- To find the successor (predecessor) of x, first search to the right (left) within its pcluster. If we find a 1, that position gives the result. Otherwise, let i floor(x/ $\sqrt{u}$) and search to the right (left) within the summary array from index i. The first position that holds a 1 gives the index of a cluster. Search within that cluster for the leftmost (rightmost) 1. That position holds the successor (predecessor).

In each of the above operations, we search through at most two clusters of $\sqrt{u}$ bits plus the summary array, and so each operation takes O ($\sqrt{u}$) time.

- **A recursive structure:** we make the structure recursive, shrinking the universe size by the square root at each level of recursion. Starting with a universe of size u, we make structures holding $\sqrt{u}$ = $u^{1/2}$ items, which themselves hold structures of $u^{1/4}$ items, which hold structures of $u^{1/8}$ items, and so on, down to a base size of 2. This recurrence relation can be represented as:

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

  Let m = lg u, so that u = 2^m

  and we have

  $$T(2^m) = T(2^{m/2}) + O(1):$$

Now we rename S(m) = T(2^m), giving the new recurrence

$$S(m) = S(m/2) + O(1)$$

By case 2 of the master method, this recurrence has the solution S(m) = O(lg m). We change back from S(m) to T(u), giving T(u) = T(2^m) = S(m) = O(lg m) = O(lg lg u).

Now a given value **x** resides in cluster number floor(x/ $\sqrt{u}$). If we view x as a lg u-bit binary integer, that cluster number, $floor($ x/ $\sqrt{u}$), is given by the most significant (lg u)/2 bits of x. Within its cluster, x appears in position x mod $\sqrt{u}$, which is given by the least significant (lg u)/2 bits of x. We will need to index in this way, and so we define some functions that will help us do so:

high(x) = $floor$ (x/ $\sqrt{u}$)

low(x) = x mod $\sqrt{u}$

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

The function high(x) gives the most significant (lg u)/2 bits of x, producing the number of x's cluster. The function low(x) gives the least significant (lg u)/2 bits of x and provides x's position within its cluster. The function index(x, y) builds an element number from x and y, treating x as the most significant (lg u/2) bits of the element number and y as the least significant (lg u/2) bits.

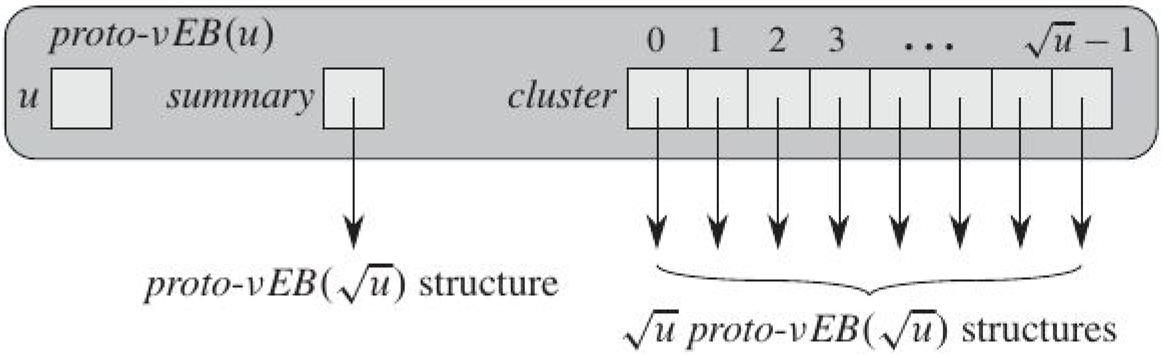
- **Proto van Emde Boas structures:** It uses a recursive data structure to support the operations.

  For the universe {0, 1, 2, ..., u-1}, we define a proto van Emde Boas structure, or proto-vEB structure, which we denote as proto-vEB(u), recursively as follows. Each proto-vEB(u) structure contains an attribute u giving its universe size. In addition, it contains the following:

  - If u = 2, then it is the base size, and it contains an array A[0...u-1] of two bits.
  - Otherwise, u = 2^{2^k} for some integer k >= 1, so that u >= 4. In addition to the universe size u, the data structure proto-vEB(u) contains the following attributes:
    - a pointer named summary to a proto-vEB($\sqrt{u}$) structure and

- an array cluster[0… $\sqrt{u}$-1] of $\sqrt{u}$ pointers, each to a proto-vEB($\sqrt{u}$) structure.

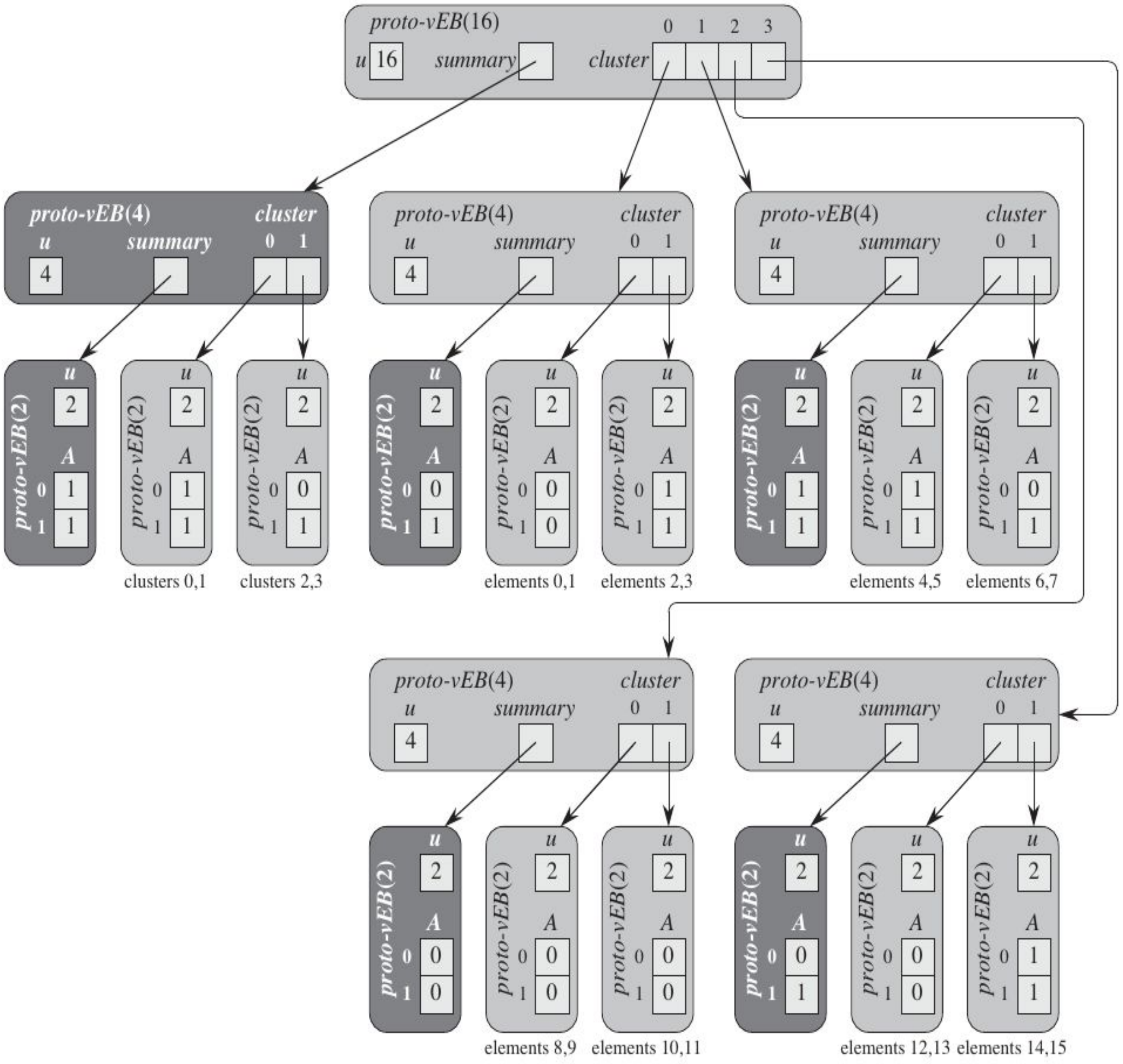
The element x, where 0 <= x < u, is recursively stored in the cluster numbered high(x) as element low(x) within that cluster.



In the proto-vEB structure, we use explicit pointers rather than index calculations. The array summary contains the summary bits store recursively in a proto-vEB structure, and the array cluster contains $\sqrt{u}$ pointers.

If the value i is in the proto-vEB structure pointed to by summary, then the ith cluster contains some value in the set being represented. As in the tree of constant height, cluster[i] represents the values i $\sqrt{u}$ through (i+1) $\sqrt{u}$-1, which form the ith cluster. Like the clusters, each summary is just a dynamic set with universe size $\sqrt{u}$, and so we represent each summary as a proto-vEB($\sqrt{u}$) structure.

The figure below shows a fully expanded proto-vEB(16) structure representing the set {2, 3, 4, 5, 7, 14, 15}.



The figure above shows a fully expanded proto-vEB(16) structure representing the set {2, 3, 4, 5, 7, 14, 15}.

proto van Emde Boas Operations

We shall now describe how to perform operations required in the implementation of priority queue for Dijkstra on a proto-vEB structure.

- **Finding the minimum element:** The procedure PROTO-vEB-MINIMUM(V) returns the minimum element in the proto-vEB structure V , or NIL if V represents an empty set.

```
PROTO-vEB-MINIMUM(V)
 1  if V.u == 2
 2      if V.A[0] == 1
 3          return 0
 4      elseif V.A[1] == 1
 5          return 1
 6      else return NIL
 7  else min-cluster = PROTO-vEB-MINIMUM(V.summary)
 8      if min-cluster == NIL
 9          return NIL
10      else offset = PROTO-vEB-MINIMUM(V.cluster[min-cluster])
11          return index(min-cluster, offset)
```

Line 1 tests for the base case, which lines 2–6 handle by brute force. Lines 7–11 handle the recursive case. First, line 7 finds the number of the first cluster that contains an element of the set. It does so by recursively calling PROTO-vEB-MINIMUM on V.summary, which is a proto-vEB($\sqrt{u}$) structure. Line 7 assigns this cluster number to the variable min-cluster. If the set is empty, then the recursive call returned NIL, and line 9 returns NIL. Otherwise, the minimum element of the set is somewhere in cluster number min-cluster. The recursive call in line 10 finds the offset within the cluster of the minimum element in this cluster. Finally, line 11 constructs the value of the minimum element from the cluster number and offset, and it returns this value.

Time Complexity of PROTO-vEB-MINIMUM

Let T(u) denote the worst-case time for PROTO-vEB-MINIMUM on a proto-vEB(u) structure, we have the recurrence:

$$T(u) = 2T(\sqrt{u}) + O(1)$$

Let m = lg u,

$$T(2^{m}) = 2T(2^{m/2}) + O(1)$$

Renaming S(m) = T(2^{m}) gives

$$S(m) = 2S(m/2) + O(1)$$

which, by case 1 of the master method, has the solution S(m) = Θ(m). By changing back from S(m) to T(u), we have that T(u) = T(2^{m}) = S(m) = Θ(m) = Θ(lg u).

Because this procedure makes two recursive calls on proto-vEB($\sqrt{u}$) structures, it does not run in O(lg lg u) time in the worst case.

- **Finding the successor:** The procedure PROTO-vEB-SUCCESSOR(V, x) returns the smallest element in the proto-vEB structure **V** that is greater than x, or NIL if no element in V is greater than x. It does not require x to be a member of the set, but it does assume that 0 $\leq$x < V.u.

```
PROTO-vEB-SUCCESSOR(V, x)
1   if V.u == 2
2       if x == 0 and V.A[1] == 1
3           return 1
4       else return NIL
5   else offset = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
6       if offset ≠ NIL
7           return index(high(x), offset)
8       else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
9           if succ-cluster == NIL
10              return NIL
11          else offset = PROTO-vEB-MINIMUM(V.cluster[succ-cluster])
12              return index(succ-cluster, offset)
```

The PROTO-vEB-SUCCESSOR procedure works as follows. Line 1 tests for the base case, which lines 2–4 handle by brute force: the only way that x can have a successor within a proto-vEB(2) structure is when x=0 and A[1] is 1. Lines 5–12 handle the recursive case. Line 5 searches for a successor to x within x's cluster, assigning the result to offset. Line 6 determines whether x has a successor within its cluster; if it does, then line 7 computes and returns the value of this successor. Otherwise, we have to search in other clusters. Line 8 assigns to succ-cluster the number of the next non empty cluster, using the summary information to find it. Line 9 tests whether succ-cluster is NIL, with line 10 returning NIL if all succeeding clusters are empty. If succ-cluster is non-NIL , line 11 assigns the first element within that cluster to offset, and line 12 computes and returns the minimum element in that cluster.

Time Complexity of PROTO-vEB-SUCCESSOR

The SUCCESSOR operation is even worse. In the worst case, it makes two recursive calls, along with a call to PROTO-vEB-MINIMUM. Thus,

$$T(u) = 2T(\sqrt{u}) + \Theta(\lg \sqrt{u})$$
$$= 2T(\sqrt{u}) + \Theta(\lg u)$$

This recurrence has the solution T(u) = Θ(lg u lg lg u).

- **Inserting an element:** To insert an element, we need to insert it into the appropriate cluster and also set the summary bit for that cluster to 1. The procedure PROTO-vEB-INSERT(V, x) inserts the value x into the proto-vEB structure V.

```
PROTO-vEB-INSERT(V, x)
1   if V.u == 2
2       V.A[x] = 1
3   else PROTO-vEB-INSERT(V.cluster[high(x)], low(x))
4       PROTO-vEB-INSERT(V.summary, high(x))
```

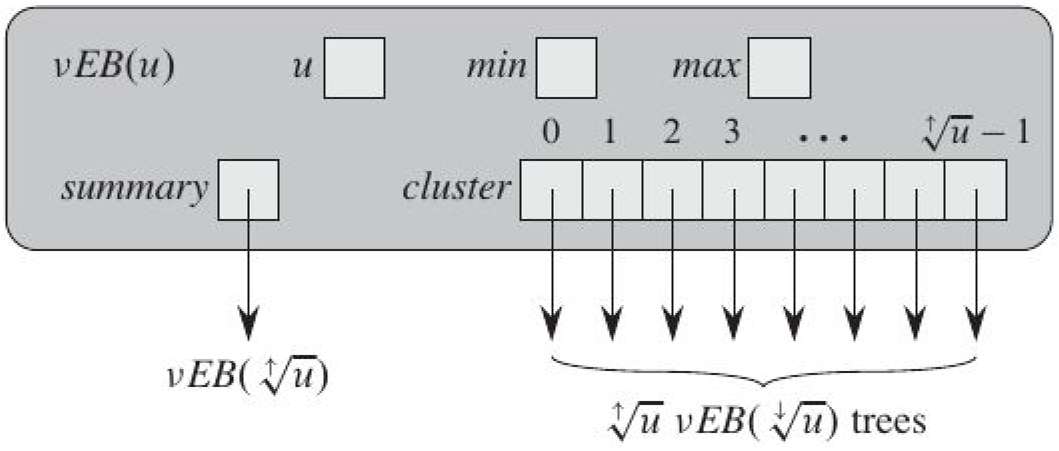
In the base case, line 2 sets the appropriate bit in the array A to 1. In the recursive case, the recursive call in line 3 inserts x into the appropriate cluster, and line 4 sets the summary bit for that cluster to 1.

van Emde Boas Trees

The van Emde Boas tree, or vEB tree, modifies the proto-vEB structure. We denote a vEB tree with a universe size of u as vEB(u) and, unless u equals the base size of 2, the attribute summary points to a vEB(uppersqrt(u)) tree and the array cluster [0...uppersqrt(u) - 1] points to uppersqrt(u) vEB(lowersqrt(u)) trees. A vEB tree contains two attributes not found in a proto-vEB structure:

- min stores the minimum element in the vEB tree, and
- max stores the maximum element in the vEB tree.

Furthermore, the element stored in min does not appear in any of the recursive vEB(lowersqrt(u)) trees that the cluster array points to. The elements stored in a vEB(u) tree V , therefore, are V.min plus all the elements recursively stored in the vEB(lowersqrt(u)) trees pointed to by V.cluster[0...u-1].


$vEB(u)$
u
min
max
0 1 2 3 ... $\sqrt[\uparrow]{u}-1$
$summary$
$cluster$
$vEB(\sqrt[\uparrow]{u})$
$\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$ trees

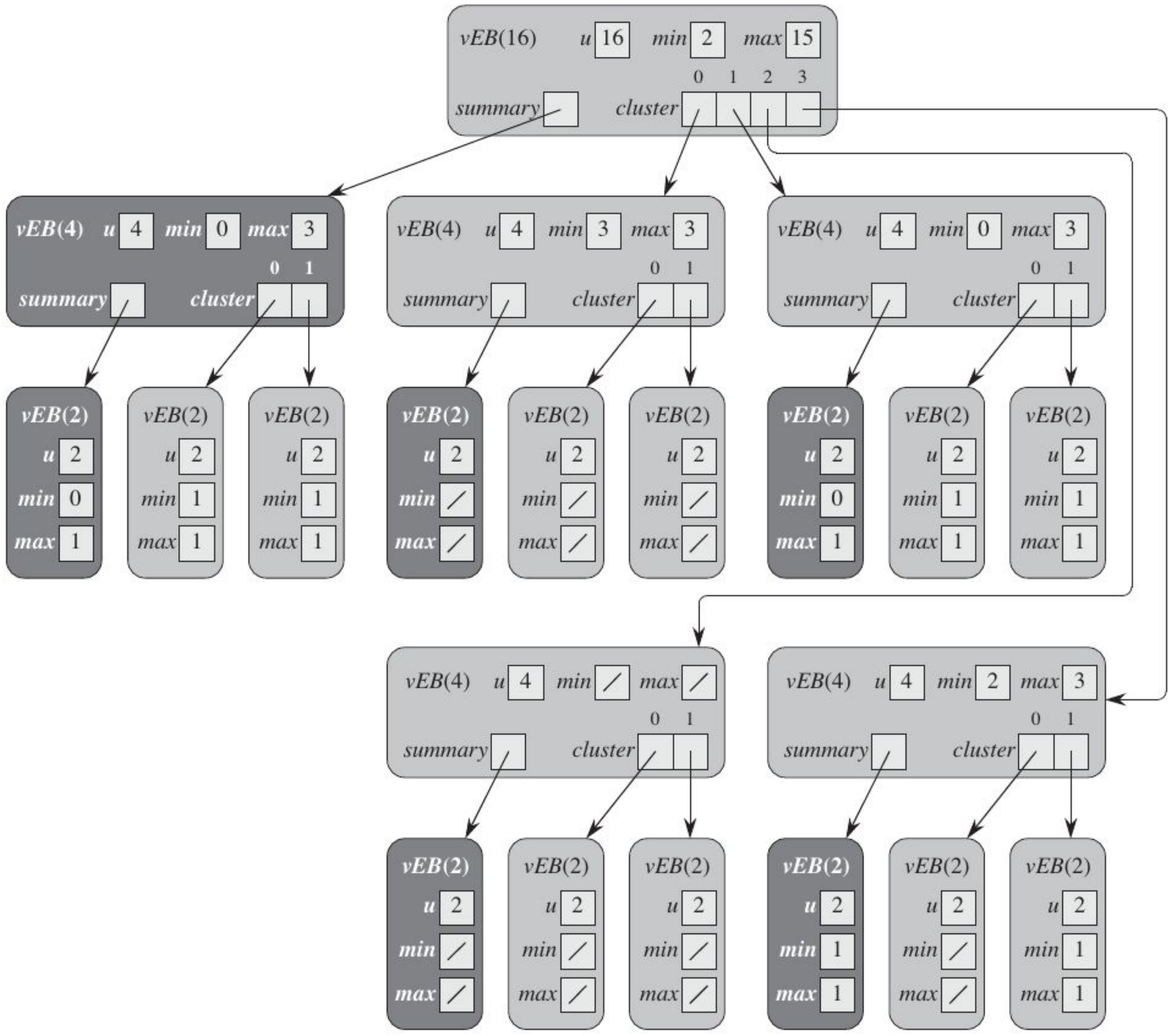
Since the base size is 2, a vEB(2) tree does not need the array A that the corresponding proto-vEB(2) structure has. Instead, we can determine its elements from its min and max attributes. In a vEB tree with no elements, regardless of its universe size u, both min and max are NIL.

The **min and max** attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:

1. The SUCCESSOR operation can avoid making a recursive call to determine whether the successor of a value x lies within high.x/. That is because x’s successor lies within its cluster if and only if x is strictly less than the max attribute of its cluster. A symmetric argument holds for PREDECESSOR and min.
2. The MINIMUM and MAXIMUM operations do not even need to recurse, for they can just return the values of min or max.
3. We can tell whether a vEB tree has no elements, exactly one element, or at least two elements in constant time from its min and max values. This ability will help in the INSERT and DELETE operations. If min and max are both NIL, then the vEB tree has no elements. If min and max are non- NIL but are equal to each other, then the vEB tree has exactly one element. Otherwise, both min and max are non-NIL but are unequal, and the vEB tree has two or more elements.
4. If we know that a vEB tree is empty, we can insert an element into it by updating only its min and max attributes. Hence, we can insert into an empty vEB tree in constant time. Similarly, if we know that a vEB tree has only one element, we can delete that element in constant time by updating only min and max. These properties will allow us to cut short the chain of recursive calls.

The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence:

$$T(u) \leq T(\text{uppersqrt}(u)) + O(1)$$



A vEB(16) tree V holding the set {2, 3, 4, 5, 7, 14, 15}. Because the smallest element is 2, V.min equals 2, and even though high(2) = 0, the element 2 does not appear in the vEB(4) tree pointed to by V.cluster[0]: notice that V.cluster[0].min equals 3, and so 2 is not in this vEB tree. Similarly, since V.cluster[0].min equals 3, and 2 and 3 are the only elements in V.cluster[0], the vEB(2) clusters within V.cluster[0] are empty.

van Emde Boas Trees Operations

- **Finding the minimum and maximum elements:** Finding minimum and maximum element in vEB tree takes constant time.

```
vEB-TREE-MINIMUM(V)
return V.min
```

```
vEB-TREE-MAXIMUM(V)
return V.max
```

- **Finding the successor and predecessor:** In a vEB tree because we can access the maximum value 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)
if V.u == 2
    if x == 0 and V.max == 1
        return 1
    else return NIL
elseif V.min ≠ NIL and x < V.min
    return V.min
else max-low = vEB-TREE-MAXIMUM(V.cluster[high(x)])
    if max-low ≠ NIL and low(x) < max-low
        offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
        return index(high(x), offset)
    else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
        if succ-cluster == NIL
            return NIL
        else offset = vEB-TREE-MINIMUM(V.cluster[succ-cluster])
            return index(succ-cluster, offset)
```

- **Inserting an element:** 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:

```
vEB-EMPTY-TREE-INSERT(V, x)
V.min = x
V.max = x
```

```
vEB-TREE-INSERT(V, x)
if V.min == NIL
    vEB-EMPTY-TREE-INSERT(V, x)
else if x < V.min
        exchange x with V.min
    if V.u > 2
        if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
            vEB-TREE-INSERT(V.summary, high(x))
            vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
        else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
    if x > V.max
        V.max = x
```

  - **Time Complexity of vEB Tree Operations:**

    The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence:

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

    Letting m = lg u, we rewrite it as:

$$T(2^{m}) \leq T(2^{m/2}) + O(1)$$

    Noting that $m/2 \leq 2m=3$ for all $m \geq 2$, we have

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

Let S(m) = T(2^m), we rewrite this last recurrence as

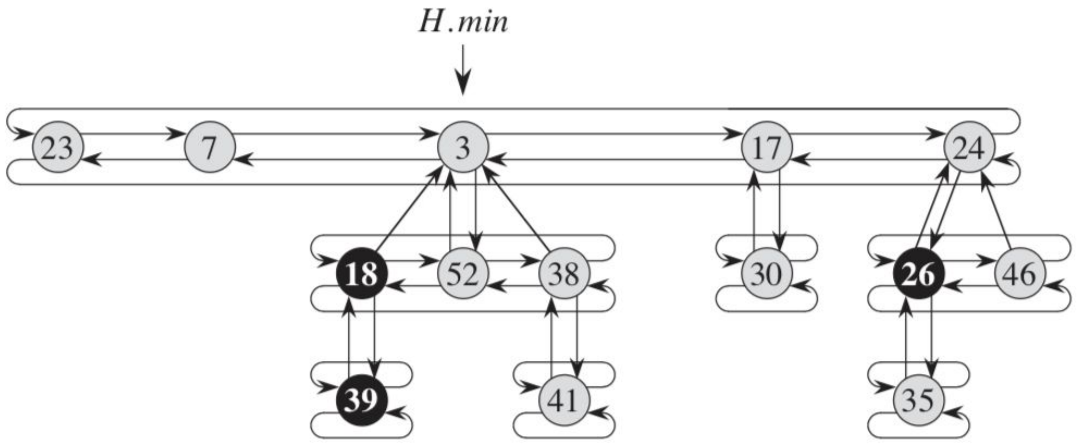
$$S(m) = S(2m/3) + O(1)$$

which, by case 2 of the master method, has the solution S(m) = O(lg m). (In terms of the asymptotic solution, the fraction 2/3 does not make any difference compared with the fraction 1/2, because when we apply the master method, we find that $\log_{3/2} 1 = \log_2 1 = 0$) Thus, we have T(u) = T(2^m) = S(m) = O(lg m) = O(lg lg u).

Fibonacci Heap

A ***Fibonacci heap*** is a collection of rooted trees that are ***min-heap ordered***. That is, each tree obeys the ***min-heap property***: the key of a node is greater than or equal to the key of its parent. Fibonacci heaps are based on rooted trees. We represent each element by a node within a tree, and each node has a *key* attribute.

Each node in Fibonacci Heap contains a pointer to its parent and a pointer to any one of its children. The children of a node are linked together in a circular, doubly linked list, which we call the ***child list*** of that node. Each child in a child list has pointers *left* and *right* that point to left and right siblings, respectively. Siblings may appear in a child list in any order. Circular, doubly linked lists have two advantages for use in Fibonacci heaps. First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in O(1) time. Second, given two such lists, we can concatenate them (or “splice” them together) into one circular, doubly linked list in O(1) time.



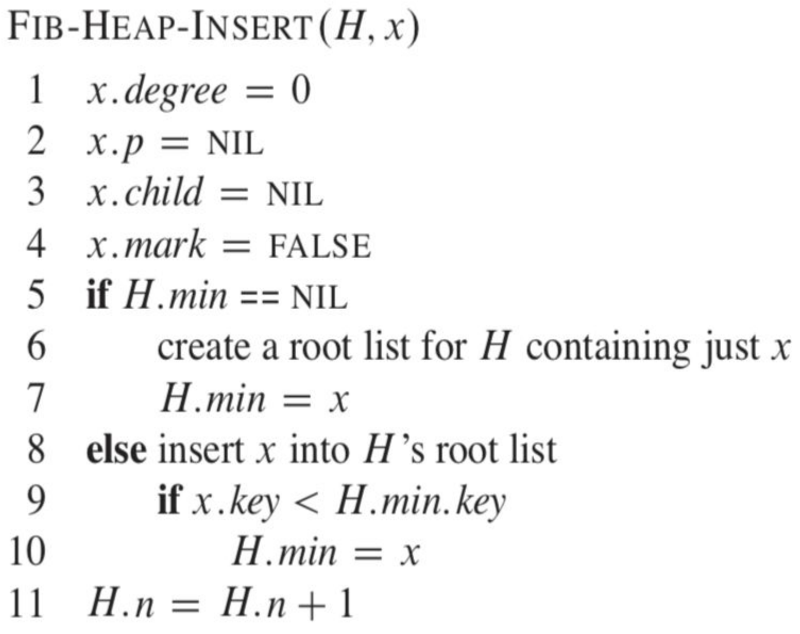
We access a given Fibonacci heap H by a pointer H.*min* to the root of a tree containing the minimum key; we call this node the ***minimum node*** of the Fibonacci heap. If more than one root has a key with the minimum value, then any such root may serve as the minimum node. When a Fibonacci heap H is empty, H.*min* is NIL. The roots of all the trees in a Fibonacci heap are linked together using their *left* and *right* pointers into a circular, doubly linked list called the ***root list*** of the Fibonacci heap. The pointer H.*min* thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list. We rely on one other attribute for a Fibonacci heap H.*n*, the number of nodes currently in H.

Creating a new Fibonacci Heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where H:*n* D 0 and H:*min* D NIL; there are no trees in H. The amortized cost of MAKE-FIB-HEAP is equal to its O(1) actual cost.

Inserting a node

The following procedure inserts node x into Fibonacci heap H , assuming that the node has already been allocated and that x:*key* has already been filled in.



```
FIB-HEAP-INSERT$(H, x)$
 1  $x.degree = 0$
 2  $x.p =$ NIL
 3  $x.child =$ NIL
 4  $x.mark =$ FALSE
 5  if $H.min ==$ NIL
 6      create a root list for $H$ containing just $x$
 7      $H.min = x$
 8  else insert $x$ into $H$'s root list
 9      if $x.key < H.min.key$
10          $H.min = x$
11  $H.n = H.n + 1$
```

Lines 1–4 initialize some of the structural attributes of node x. Line 5 tests to see whether Fibonacci heap H is empty. If it is, then lines 6–7 make x be the only node in H’s root list and set H:*min* to point to x. Otherwise, lines 8–10 insert x into H’s root list and update H:*min* if necessary. Finally, line 11 increments H:*n* to reflect the addition of the new node.

Finding the Minimum Node

The minimum node of a Fibonacci heap H is given by the pointer H.*min*, so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

Binomial Heap

A binomial heap is a collection of binomial trees where each Binomial Tree follows Min Heap property. And there can be at most one Binomial Tree of any degree.

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 2^k nodes.
(b) The height of tree is k.
(c) It has exactly ^kC_i nodes at the depth i for i = 0, 1, 2,.....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.

Creating a new Binomial Heap

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H , where head[H] = NIL. The running time is Θ(1).

Finding the Minimum Key

The following procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞.

```
BINOMIAL-HEAP-MINIMUM(H)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
    do if key[x] < min
          then min ← key[x]
               y ← x
       x ← sibling[x]
return y
```

Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number 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).

Inserting a node

The following procedure inserts node x into binomial heap H , assuming that x has already been allocated and key[x] has already been filled in.

```
BINOMIAL-HEAP-INSERT($H$, $x$)
    $H'$ ← MAKE-BINOMIAL-HEAP()
    $p[x]$ ← NIL
    $child[x]$ ← NIL
    $sibling[x]$ ← NIL
    $degree[x]$ ← 0
    $head[H']$ ← $x$
    $H$ ← BINOMIAL-HEAP-UNION($H$, $H'$)
```

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(lg n) time. The call to BINOMIAL-HEAP-UNION takes care of freeing the temporary binomial heap H'.

Dijkstra’s Single Source Shortest Path Algorithm

Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V,E) for the case in which all edge weights are nonnegative.

Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u ∈ V-S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

Line 1 initializes the d and *Π* values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that Q = V-S at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V ; since S = ∅ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from Q = V-S and line 6 adds it to set S , thereby maintaining the invariant. (The first time through this loop, *u* = *s*.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in V-S. Then, lines 7–8 relax each edge *(u,v)* leaving *u*, thus updating the estimate *v.d* and the predecessor *v.Π* if we can improve the shortest path to *v* found so far by going through *u*. Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q and added to S exactly once, so that the **while** loop of lines 4–8 iterates exactly |V| times.

Analysis

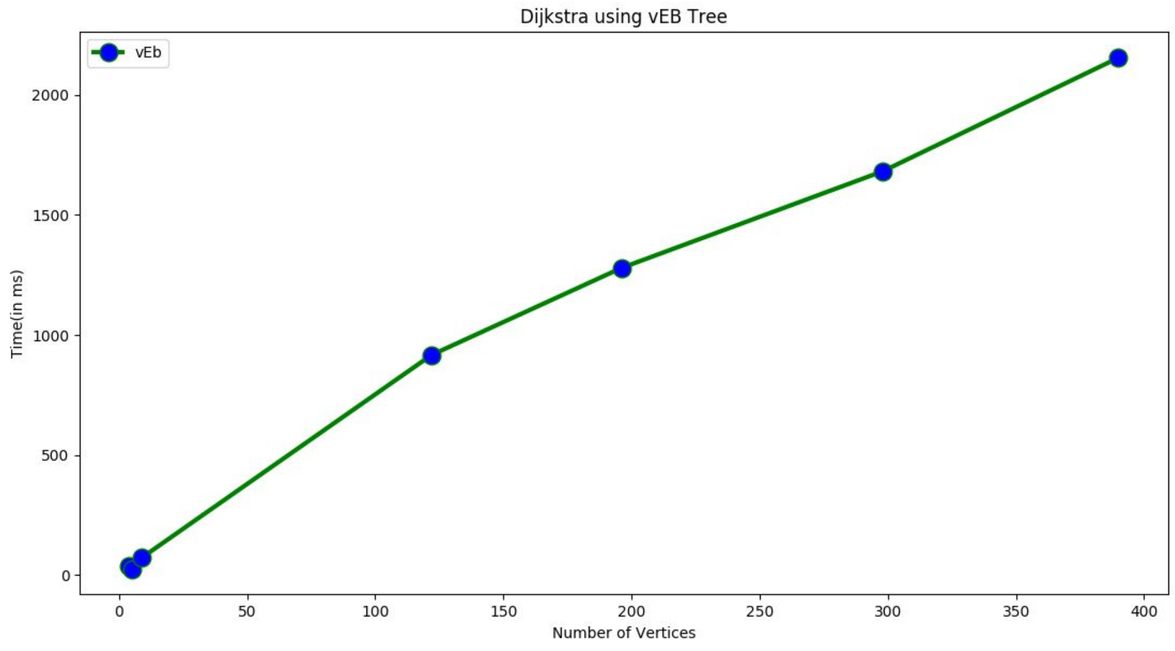
Dijkstra using vEB Tree

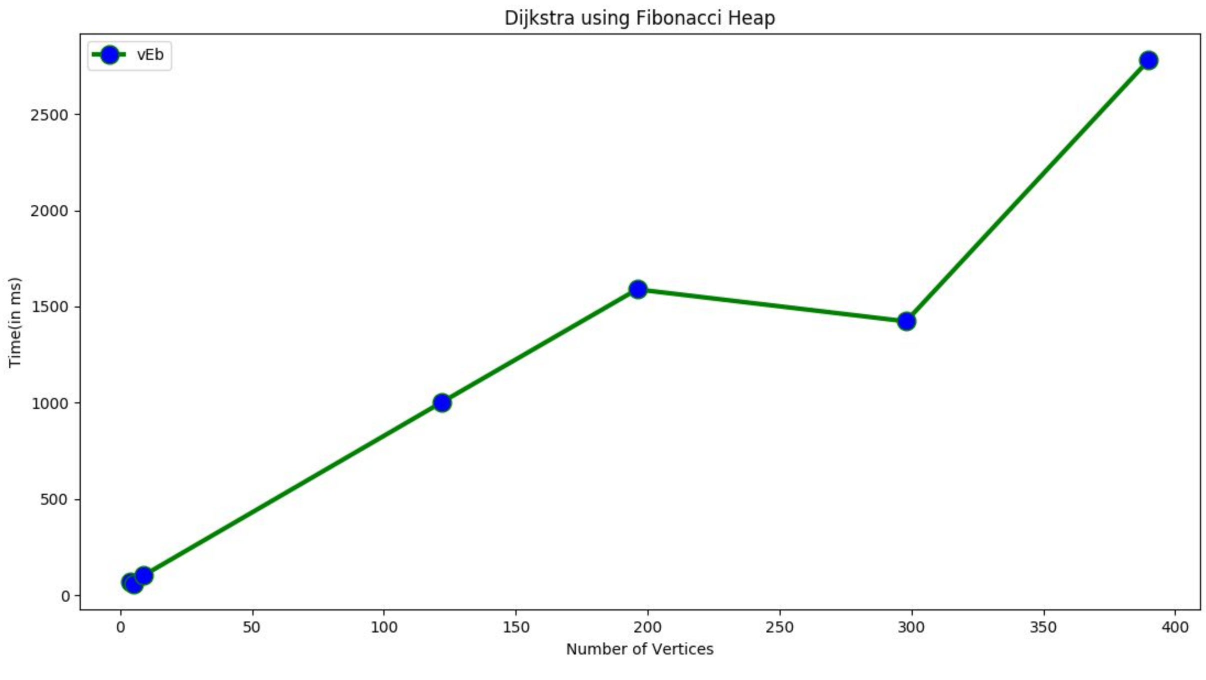
Priority Queue is implemented using vEB tree in the implementation of Dijkstra. The priority queue implementation using vEB is done by using the INSERT and SUCCESSOR operations of vEB tree along with an unordered_map. The unordered_map is used for storing the nodes having same weights at a given point of time.

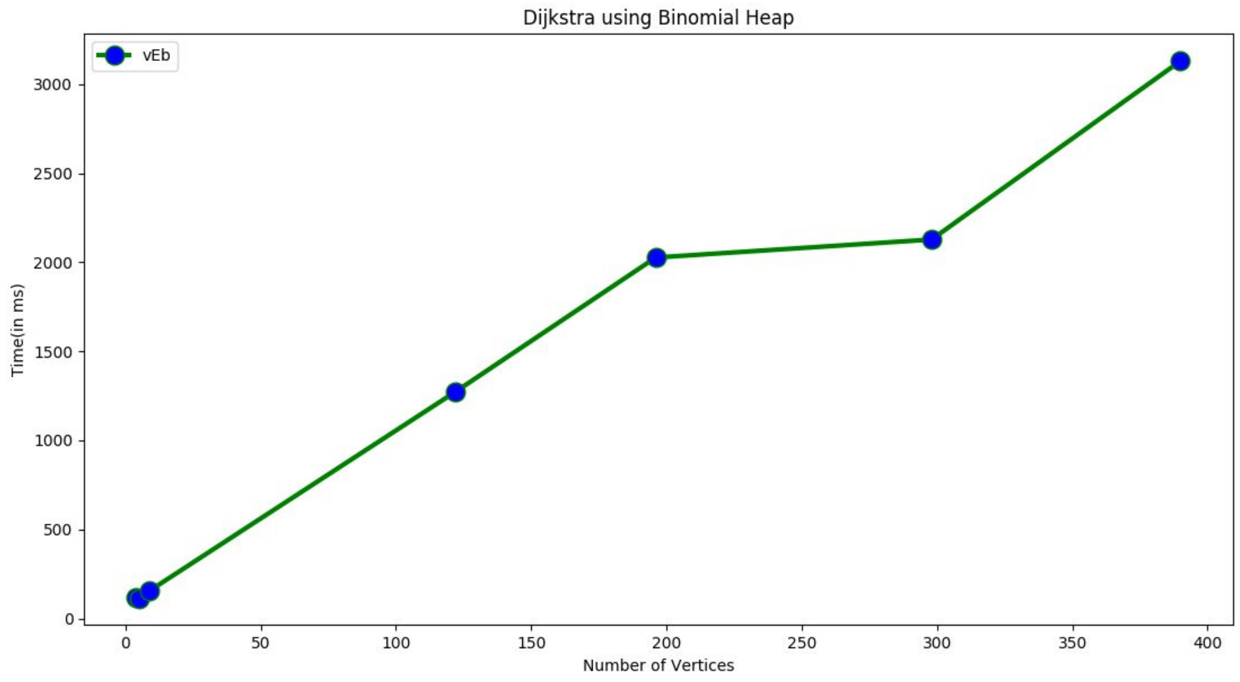
An adjacency list is created for the given graph using an array of vectors of pair. Now, we start from the source node, assign it a weight of 0 and all other nodes are assigned a weight of INT_MAX. The bit vector corresponding to weight 0 is set to 1 in the vEB tree. Using the adjacency list, the adjacent vertices of a given vertex are relaxed, i.e, their minimum distance from the source is updated and the corresponding weight is inserted to the vEB tree. In each iteration we find the successor of the current minimum weight which gives the next minimum vertex required for Dijkstra.

The unordered map contains a list of all the nodes that currently have a given weight in a particular iteration. If that vertex is already relaxed, i.e., the shortest path to the node has been calculated then that node is removed from other weight keys in the *unordered map*. Thus the *unordered map* has weights as the keys and corresponding nodes as the values.

The nodes and their corresponding weight after completing the iteration gives the final values of shortest path to these nodes.


Dijkstra using vEB Tree
vEb
Time(in ms)
Number of Vertices
0
500
1000
1500
2000
0
50
100
150
200
250
300
350
400

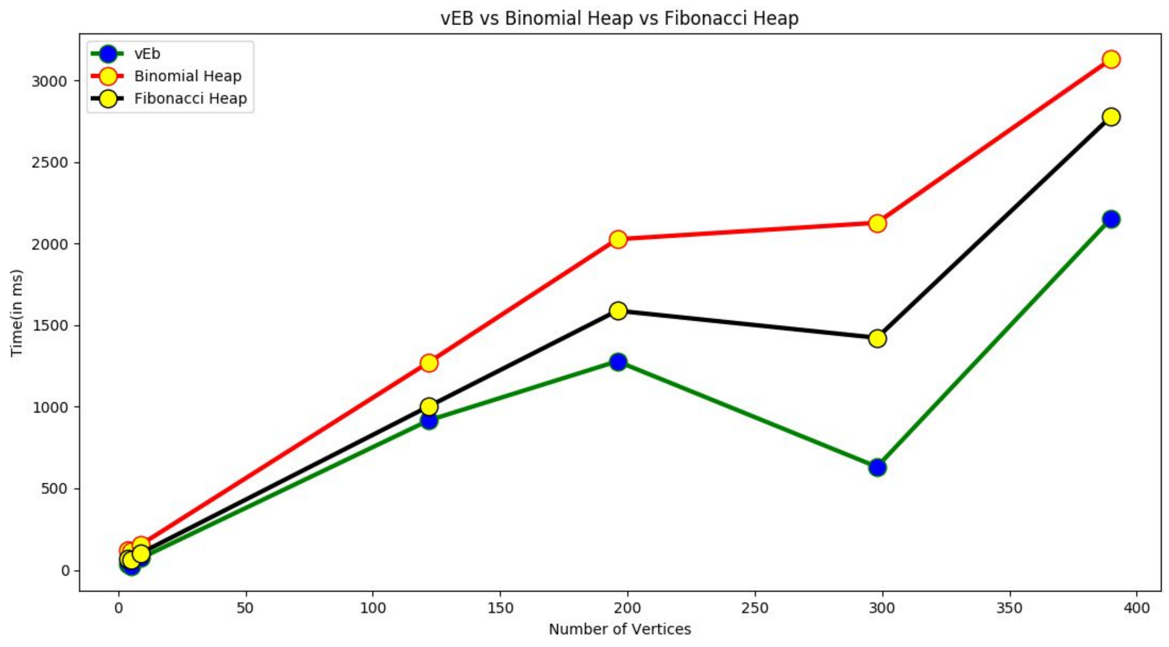

Dijkstra using Fibonacci Heap
vEb
Time(in ms)
Number of Vertices
0
500
1000
1500
2000
2500
0
50
100
150
200
250
300
350
400



Comparison with Fibonacci and Binomial Heap

The time complexity of implementing Dijkstra using **Binomial Heaps** looks O(V^2) as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed O(V+E) times (similar to BFS). The inner loop has **decreaseKey()** operation which takes O(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is O((E+V)*LogV) = **O(ELogV)**

Time complexity can be reduced to O(E + VLogV) using **Fibonacci Heap**. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.



References

- MIT Course Lecture: https://www.youtube.com/watch?v=hmReJCupbNU
- vEB Structure: https://www.youtube.com/watch?v=ZrV7GiuMNo4
- Wikipedia: https://en.wikipedia.org/wiki/Van_Emde_Boas_tree
- Wikipedia: https://en.wikipedia.org/wiki/Binomial_heap
- Wikipedia: https://en.wikipedia.org/wiki/Fibonacci_heap
- Wikipedia: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- MIT Course Lecture: https://www.youtube.com/watch?v=2E7MmKv0Y24
- Introduction to Algorithms, CLRS