**IMPLEMENTATION OF DATA STRUCTURES AND ALGORITHMS**

**(MP3 REPORT)**

**By,**

**PRIYANKA BURA**

**(pxb144230)**

Following functions are implemented as part of **BinaryHeap** Implementation:

* **Add** - function to add/insert new element
* **Remove** - function to remove first or minimum value element and return it
* **Peek** - function to return the first element in binary heap
* **percolateUp** - function to maintain the heaporder with respect to parent when pq[i] has been changed
* **percolateDown** - function to maintain the heaporder with respect to children when pq[i] has been changed
* **move** - function to change value of an element at given index
* **buildHeap** - function to create a heap
* **heapsort** - function to sort the given binary heap

Following functions are implemented as part of **IndexedHeap** Implementation:

* **decreaseKey** - function to restore heap order property after the priority of x has decreased
* **move** - function to assign given value to element at given index

Following functions are implemented as part of **Vertex** Implementation:

* **getIndex** - function to get the index of vertex (used by IndexedHeap)
* **putIndex** - function to update the index of vertex (used by IndexedHeap)
* **compare** - function to return the difference between the distance of given two vertices

Following functions are implemented as part of **MST** Implementation:

* **PrimMST** - function to find weight of MST using the priority of edges and Java's priority queue. Graph and start vertex are passed as inputs to this function. This algorithm uses a priority queue (min heap) to store all the unvisited edges, each edge is removed from the queue and it’s to/from vertex is visited and its other end vertex’s adjacent edges are added to queue and weight of MST is incremented by current processed edge’s weight, the whole process continues until the queue is empty. This algorithm finds the minimum spanning tree weight.
* **PrimMST2** - function to find weight of MST using the priority of vertices and indexed heap. Graph and start vertex are passed as inputs to this function. This algorithm uses an indexed heap to store all the vertices. Each vertex has distance ‘d’ and parent ‘p’ as attributes. Each vertex from the heap is removed to process where weight of MST is incremented by distance of current vertex and all the unvisited other end vertices of adjacent edges of current vertex are visited and their weight is updated if its previous weight is greater than weight of its edge and update the heap order with respect to parent vertex of other end vertex by perculating up, this process continues until the heap is empty. This algorithm finds the minimum spanning tree weight.

Following functions are implemented as part of **ShortestPath** Implementation:

* **DijkstraShortestPaths** - function to find ShortestPath using the priority of vertices and indexed heap. Graph and start vertex are passed as inputs to this function. This algorithm uses an indexed heap to store all the vertices. Each vertex has distance ‘d’ and parent ‘p’ as attributes. Each vertex from the heap is removed to process where all the other end vertices of adjacent edges of current vertex which are unvisited and its current vertex distance+edge weight is less than other end vertex distance are visited and their weights are changed to current vertex distance+edge weight and the heap order with respect to parent vertex of other end vertex is updated by perculating up, this process continues until the heap is empty. This algorithm find the shortest paths from given start vertex to all other vertices.

Comparison between PrimMST algorithms using Java’s PriorityQueue and IndexedHeap

PrimMST(priority queue of edges) using Java’s PriorityQueue:

|  |  |  |
| --- | --- | --- |
| File Name | Output | Time(msec) |
| G1.txt | 84950 | 1 |
| G2.txt | 110419 | 2 |
| G3.txt | 153534 | 3 |
| G4-big.txt | 10000 | 6236 |

PrimMST(priority queue of vertices) using IndexedHeap:

|  |  |  |
| --- | --- | --- |
| File Name | Output | Time(msec) |
| G1.txt | 84950 | 1 |
| G2.txt | 110419 | 3 |
| G3.txt | 153534 | 3 |
| G4-big.txt | 10000 | 513 |

From the above run times, it is evident that the prim’s algorithm using indexed heap runs **12** times as fast as prim’s algorithm using java’s priority queues.

Output and Runtime for Dijkstra'a algorithm for shortest paths using indexed heaps:

|  |  |  |
| --- | --- | --- |
| File Name | Output | Time(msec) |
| G1.txt | 12020 | 1 |
| G2.txt | 9106 | 3 |
| G3.txt | 10672 | 3 |
| G4-big.txt | 4 | 484 |