Greedy Algorithms | Set 6 (Prim’s MST for Adjacency List Representation)

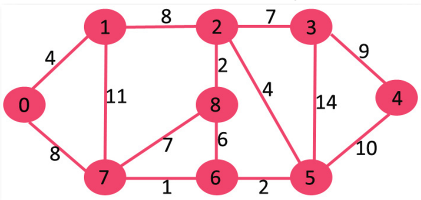
We recommend to read following two posts as a prerequisite of this post.

1. Greedy Algorithms | Set 5 (Prim’s Minimum Spanning Tree (MST))
2. Graph and its representations

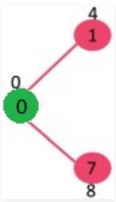
We have discussed Prim’s algorithm and its implementation for adjacency matrix representation of graphs. The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.
As discussed in the previous post, in Prim’s algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using BFS. The idea is to traverse all vertices of graph using BFS and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the cut. Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is O(LogV) in Min Heap.

Following are the detailed steps.
1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.
2) Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).
3) While Min Heap is not empty, do following
…..**a)** Extract the min value node from Min Heap. Let the extracted vertex be u.
…..**b)** For every adjacent vertex v of u, check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of u-v, then update the key value of v as weight of u-v.

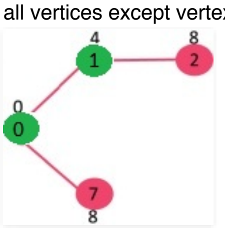
Let us understand the above algorithm with the following example:



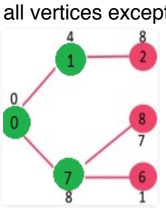
Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.
The vertices in green color are the vertices included in MST.



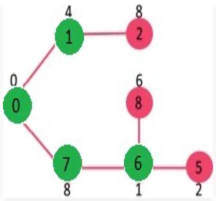
Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.



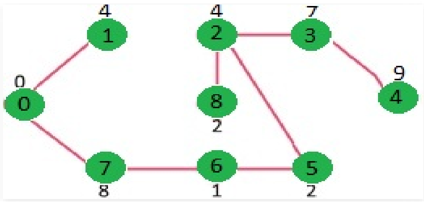
Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.



Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



C++ Python

```
# A Python program for Prims's MST for
# adjacency list representation of graph

from collections import defaultdict
import sys

class Heap():

    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newMinHeapNode(self, v, dist):
        minHeapNode = [v, dist]
        return minHeapNode

    # A utility function to swap two nodes
    # min heap. Needed for min heapify
    def swapMinHeapNode(self,a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    # A standard function to heapify at gi
    # This function also updates position
    # when they are swapped. Position is r
    # for decreaseKey()
    def minHeapify(self, idx):
        smallest = idx
        left = 2*idx + 1
        right = 2*idx + 2

        if left < self.size and self.array
                                self.array
            smallest = left

        if right < self.size and self.arra
                                self.array
            smallest = right

        # The nodes to be swapped in min h
        # if idx is not smallest
        if smallest != idx:

            # Swap positions
            self.pos[ self.array[smallest]
            self.pos[ self.array[idx][0] ]

            # Swap nodes
            self.swapMinHeapNode(smallest,

            self.minHeapify(smallest)

    # Standard function to extract minimum
    def extractMin(self):

        # Return NULL wif heap is empty
        if self.isEmpty() == True:
            return

        # Store the root node
        root = self.array[0]

        # Replace root node with last node
        lastNode = self.array[self.size -
        self.array[0] = lastNode

        # Update position of last node
        self.pos[lastNode[0]] = 0
        self.pos[root[0]] = self.size - 1

        # Reduce heap size and heapify roo
        self.size -= 1
        self.minHeapify(0)

        return root
```

```
    def isEmpty(self):
        return True if self.size == 0 else

    def decreaseKey(self, v, dist):

        # Get the index of v in  heap arra

        i = self.pos[v]

        # Get the node and update its dist
        self.array[i][1] = dist

        # Travel up while the complete tre
        # hepified. This is a O(Logn) loop
        while i > 0 and self.array[i][1] <
                    self.array[(i - 1) / 2

            # Swap this node with its pare
            self.pos[ self.array[i][0] ] =
            self.pos[ self.array[(i-1)/2][
            self.swapMinHeapNode(i, (i - 1

            # move to parent index
            i = (i - 1) / 2;

    # A utility function to check if a giv
    # 'v' is in min heap or not
    def isInMinHeap(self, v):

        if self.pos[v] < self.size:
            return True
        return False


def printArr(parent, n):
    for i in range(1,n):
        print "%d - %d" % (parent[i], i)


class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    # Adds an edge to an undirected graph
    def addEdge(self, src, dest, weight):

        # Add an edge from src to dest.  A
        # added to the adjacency list of s
        # is added at the begining. The fi
        # the node has the destination and
        # elements has the weight
        newNode = [dest, weight]
        self.graph[src].insert(0, newNode)

        # Since graph is undirected, add a
        # dest to src also
        newNode = [src, weight]
        self.graph[dest].insert(0, newNode

    # The main function that prints the Mi
    # Spanning Tree(MST) using the Prim's
    # It is a O(ELogV) function
    def PrimMST(self):
        # Get the number of vertices in gr
        V = self.V

        # key values used to pick minimum
        key = []

        # List to store contructed MST
        parent = []

        # minHeap represents set E
        minHeap = Heap()

        #  Initialize min heap with all ve
        # vertices (except the 0th vertex)
        for v in range(V):
            parent.append(-1)
            key.append(sys.maxint)
            minHeap.array.append( minHeap.
            minHeap.pos.append(v)

        # Make key value of 0th vertex as
        # that it is extracted first
        minHeap.pos[0] = 0
        key[0] = 0
        minHeap.decreaseKey(0, key[0])

        # Initially size of min heap is eq
        minHeap.size = V;

        # In the following loop, min heap
        # not yet added in the MST.
        while minHeap.isEmpty() == False:

            # Extract the vertex with mini
            newHeapNode = minHeap.extractM
            u = newHeapNode[0]

            # Traverse through all adjacen
            # (the extracted vertex) and u
            # distance values
```

```
            for pCrawl in self.graph[u]:

                v = pCrawl[0]

                # If shortest distance to
                # yet, and distance to v t
                # its previously calculate
                if minHeap.isInMinHeap(v)
                    key[v] = pCrawl[1]
                    parent[v] = u

                    # update distance valu
                    minHeap.decreaseKey(v,

        printArr(parent,V)


# Driver program to test the above functic
graph = Graph(9)
graph.addEdge(0, 1, 4)
graph.addEdge(0, 7, 8)
graph.addEdge(1, 2, 8)
graph.addEdge(1, 7, 11)
graph.addEdge(2, 3, 7)
graph.addEdge(2, 8, 2)
graph.addEdge(2, 5, 4)
graph.addEdge(3, 4, 9)
graph.addEdge(3, 5, 14)
graph.addEdge(4, 5, 10)
graph.addEdge(5, 6, 2)
graph.addEdge(6, 7, 1)
graph.addEdge(6, 8, 6)
graph.addEdge(7, 8, 7)
graph.PrimMST()

# This code is contributed by Divyanshu Me
```

Run on IDE

Output:

```
0 - 1
5 - 2
2 - 3
3 - 4
6 - 5
7 - 6
0 - 7
2 - 8
```

Time Complexity: The time complexity of the above code/algorithm 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) (For a connected graph, V = O(E))

References:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

http://en.wikipedia.org/wiki/Prim's_algorithm

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   **Company Wise Coding Practice**