COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 18: 10-Apr-2019

Lecturer: Dr. Andrew Hines *Scribes: Xinlin Ruan and Martin Horgan*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

18.1 What is a Spanning Tree?

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees.

18.1.1 A minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. Also, there can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Kruskal and Prim algorithms are considered below.

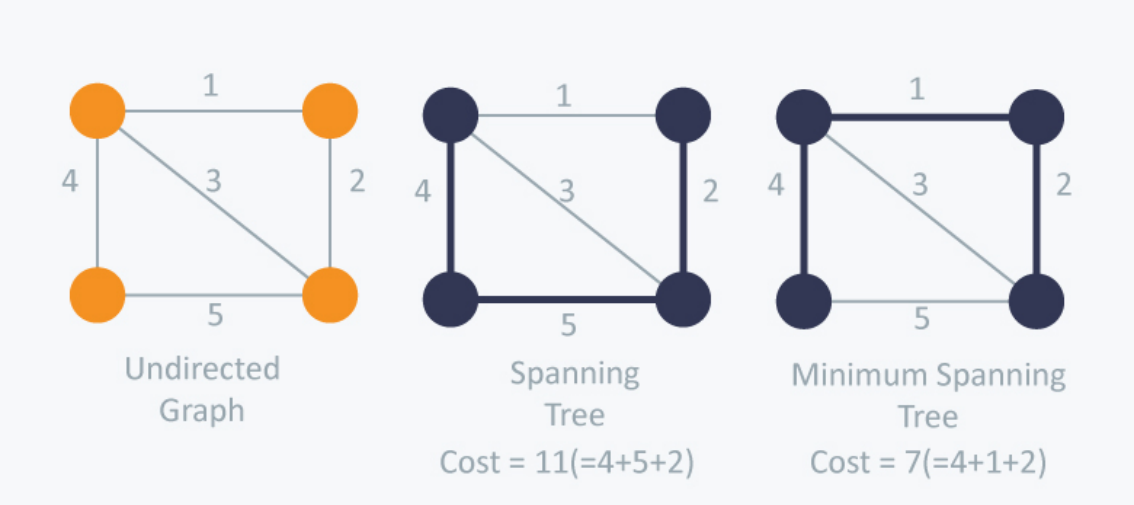

Figure 18.1: Spanning Tree: Concept

Practical applications are:

- Cluster Analysis
- Handwriting recognition

- Image segmentation
- Real-time face tracking and verification (i.e. locating human faces in a video stream)
- Protocols in computer science to avoid network cycles
- Dithering (adding white noise to a digital recording in order to reduce distortion).

18.2 Kruskals Algorithm:

This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one edge (the cheapest one) so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

18.2.0.1 Algorithm steps:

- The forest is constructed - with each node in a separate tree.
- The edges are placed in a priority queue.
- Until we've added n-1 edges;
  - Extract the cheapest edge from the queue,
  - If it forms a cycle, reject it,
  - Else add it to the forest. Adding it to the forest will join two trees together.

Consider following example:

In Kruskals algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we cant have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

18.2.1 Time complexity

Running Time = O(m log n) where m = edges and n = nodes. Testing if an edge creates a cycle can be slow unless a complicated data structure called a union-find structure is used. It usually only has to check a small fraction of the edges, but in some cases (like if there was a vertex connected to the graph by only one edge and it was the longest edge) it would have to check all the edges. This algorithm works best, of course, if the number of edges is kept to a minimum.

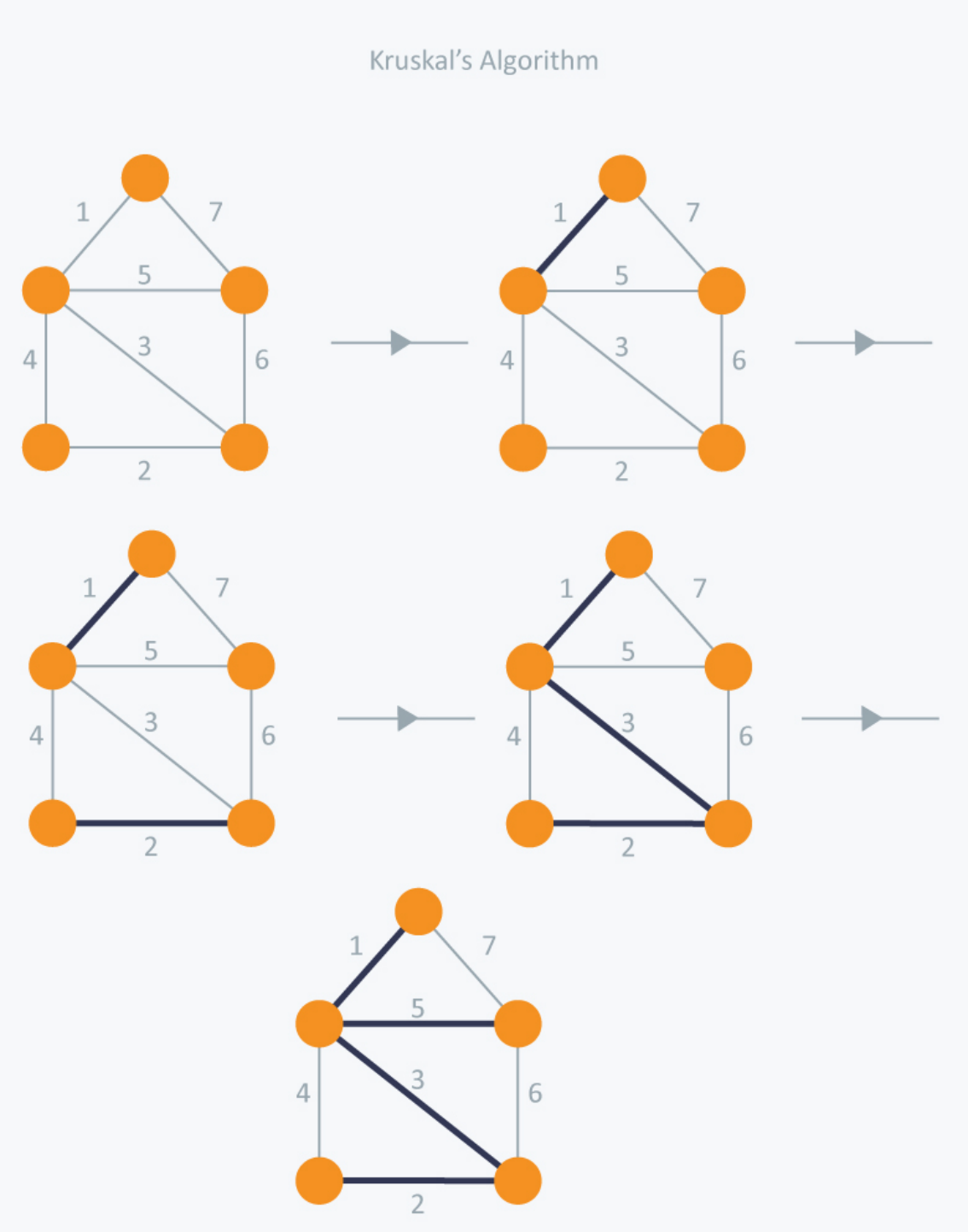

Figure 18.2:

18.2.2 Python Implementation

```
#Python program for Kruskal's algorithm to find Minimum Spanning Tree of a given connected, u

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = [] # default dictionary to #store graph

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else :
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's
        # algorithm
    def KruskalMST(self):

        result =[] #This will store the resultant MST

        i = 0 # An index variable, used for sorted edges
        e = 0 # An index variable, used for result[]
```

```
        # Step 1: Sort all the edges in non-decreasing
            # order of their
            # weight. If we are not allowed to change the
            # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,key=lambda item: item[2])

        parent = [] ; rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V -1 :

            # Step 2: Pick the smallest edge and increment
                    # the index for next iteration
            u,v,w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent ,v)

            # If including this edge does't cause cycle,
                        # include it in result and increment the index
                        # of result for next edge
            if x != y:
                e = e + 1
                result.append([u,v,w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        # print the contents of result[] to display the built MST
        print ("Following are the edges in the constructed MST")
        for u,v,weight in result:
            #print str(u) + " -- " + str(v) + " == " + str(weight)
            print ("%d -- %d == %d" % (u,v,weight))

# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)
g.KruskalMST()
```

18.3 Prims Algorithm:

Prims Algorithm also use Greedy approach to find the minimum spanning tree. In Prims Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

18.3.0.1 Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:

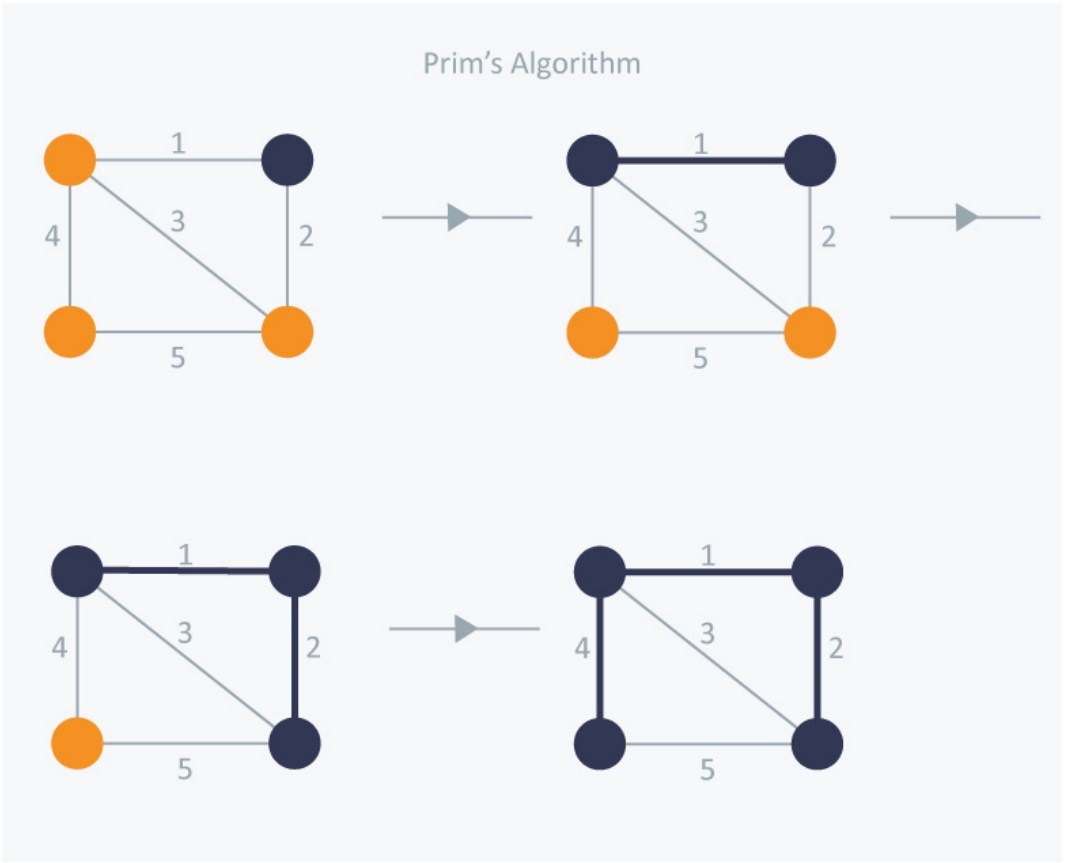

Figure 18.3: Prim's algorithm

In Prims Algorithm, we will start with an arbitrary node (it doesnt matter which one) and mark it. In each iteration we will mark a new As a greedy algorithm, Prims algorithm will select the cheapest edge and mark the vertex. So we will In Prims Algorithm, we will start with an arbitrary node (it doesnt matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prims algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we cant choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

18.3.1 Time complexity

The time complexity of the Prims Algorithm is O((V+E) log V) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

18.3.2 Python Implementation

```
# A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix representation of the graph

import sys # Library for INT_MAX

class Graph():

        def __init__(self, vertices):
                self.V = vertices
                self.graph = [[0 for column in range(vertices)]
                                        for row in range(vertices)]

        # A utility function to print the constructed MST stored in parent[]
        def printMST(self, parent):
                print("Edge \tWeight")
                for i in range(1,self.V):
                        print(parent[i],"-",i,"\t",self.graph[i][ parent[i] ] )

        # A utility function to find the vertex with
        # minimum distance value, from the set of vertices
        # not yet included in shortest path tree
        def minKey(self, key, mstSet):

                # Initilaize min value
                min = sys.maxsize

                for v in range(self.V):
                        if key[v] < min and mstSet[v] == False:
                                min = key[v]
                                min_index = v

                return min_index
```

```
        # Function to construct and print MST for a graph
        # represented using adjacency matrix representation
        def primMST(self):
            #Key values used to pick minimum weight edge in cut
            key = [sys.maxsize] * self.V
                parent = [None] * self.V # Array to store constructed MST
                # Make key 0 so that this vertex is picked as first vertex
                key[0] = 0
                mstSet = [False] * self.V

                parent[0] = -1 # First node is always the root of

                for cout in range(self.V):
                        # Pick the minimum distance vertex from
                        # the set of vertices not yet processed.
                        # u is always equal to src in first iteration
                        u = self.minKey(key, mstSet)

                        # Put the minimum distance vertex in
                        # the shortest path tree
                        mstSet[u] = True

                        # Update dist value of the adjacent vertices
                        # of the picked vertex only if the current
                        # distance is greater than new distance and
                        # the vertex in not in the shotest path tree
                        for v in range(self.V):
                        # graph[u][v] is non zero only for adjacent vertices of m
                        # mstSet[v] is false for vertices not yet included in MST
                        # Update the key only if graph[u][v] is smaller than key[v]
                        if self.graph[u][v] > 0 and mstSet[v] == False
                            and key[v] > self.graph[u][v]:
                                        key[v] = self.graph[u][v]
                                        parent[v] = u

                self.printMST(parent)

# Deliver code
g = Graph(5)
g.graph = [[0, 2, 0, 6, 0],
           [2, 0, 3, 8, 5],
           [0, 3, 0, 0, 7],
           [6, 8, 0, 0, 9],
           [0, 5, 7, 9, 0]]

g.primMST();
```

18.4 References;

[1] https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial (12-04-2019)

[2] https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15-082JF10-lec16.ppt. (12-04-2019)

[3] https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

[4] https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/