Spanning Trees

December 5, 2017

0.1 Spanning Trees

0.1.1 Disjoint Sets

- Also know as union-find data structures
- Data structure to keep track of a set of elements partitioned into a number of disjoint (non everlapping) subsets
- Three main operations: **union** and **find** and **makeSet**
- Disjoint sets can be represented with the help of linked lists but usually we implment it as a tree like structure
- In Kruskal algorithm it will be useful: with disjoint sets we can decide in approximately O(1) time whether two vertices are in the same set or not

0.1.2 Make Set

```
function makeSet(x)
    x.parent = x
```

- So the make sets operation is quite easy to implement. ~ we set the parent of the given node to the itself
- Basically we create a distinct set to all the items/nodes

0.1.3 find

```
function find(x)
    if x.parent == x
        return x
    else
        return find(x.parent)
```

- Several items can belong to the same set -> we usually represent the set with one of its items "representative of the set"
- When we search for an item with find() then the operation is going to return with the representative

0.1.4 Union

```
function union(x,y)
    xRoot = find(x)
```

```
yRoot = find(y)

xRoot.parent = yRoot
```

- The union operation is merge two disjoint sets togethere by connecting them according tot he representatives
- Problems: this tree like structure can become unbalanced
  - 1) Union by rank -> always attach the smaller tree to the root of the larger one. The tree will become more balanced: faster!!!
  - 2) Path compression -> flattening the structure of the tree. We set every visited node to be connected to the root directly !!!

0.1.5 Applications

- It is used mostly in Kruskal-algorithm implementation
- We have to check whether adding a given edge to the MST would form a cycle or not
- For checking this -> union-find data structure is extremely helpful
- We can check whether a cycle is present -> in asymtotically O(1) constant time complexity!!!

0.1.6 Spanning trees

- A spanning tree of an undirected **G** graph is a subgraph that includes all the vertices of **G**
- In general, a tree may have several spanning trees
- We can assign a weight to each edge
- A minimum spanning tree is then a spanning tree with wight less than or equal to the weight of every other spanning tree
- Has lots of applications: in big data analysis, clustering algorithms, finding minimum cost for a telecommunications company laying cable to a new neighborhhod
- Standard algorithms: Prim's-Jarnik, Kruskal -> greedy algorithms

0.2 Kruskal-algorithm

- We sort the edges according to ehir edge weights
- It can be done in **O(NlogN)** with mergesort or quicksort
- Union find data structure: **disjoint set**
- We start start adding edges to the MST and we want to make sure there will be no cycles in the spanning tree. It can be done in **O(LogV)** with the help of union find data structure
- We could use a heap instead sorting the edges in the beginning but the running time would be the same. So sometimes Kruskal's algorithm is implemented with priority queues
- Worst case running time: **O(E*logE)**, so we can use it for huge graphs too
- If the edges are sorted: the algorithm will be quasi-linear
- If the edges are sorted: the algorithm will be quasi-linear
- If we multiply the weights with a constant or add a constant to the edge weights: the result will be the same

In [20]:
```
class Vertex(object):
    def __init__(self, name):
        self.name = name;
```

```
        self.node = None;

class Node(object):
  def __init__(self, height, nodeId, parentNode):
    self.height = height;
    self.nodeId = nodeId;
    self.parentNode = parentNode;

class Edge(object):
  def __init__(self, weight, startVertex, targetVertex):
    self.weight = weight
    self.startVertex = startVertex;
    self.targetVertex = targetVertex;

  def __cmp__(self, otherEdge):
    return self.cmp(self.weight, otherEdge.weight);

  def __lt__(self, other):
    selfPriority = self.weight;
    otherPriority = other.weight;
    return selfPriority < otherPriority;
```

```
In [21]: class DisjointSet(object):
           def __init__(self, vertexList):
             self.vertexList = vertexList;
             self.rootNodes = [];
             self.nodeCount = 0;
             self.setCount = 0;
             self.makeSets(vertexList);

           def find(self, node):
             currentNode = node;
             print(node);

             while currentNode.parentNode is not None:
               currentNode = currentNode.parentNode;

             root = currentNode;
             currentNode = node;

             while currentNode is not root:
               temp = currentNode.parentNode;
               currentNode.parentNode = root;
               currentNode = temp;

             return root.nodeId
```

```
    def merge(self, node1, node2):
      index1 = self.find(node1);
      index2 = self.find(node2);

      if index1 == index2:
        return; # they are in the same set!!!

      root1 = self.rootNodes[index1];
      root2 = self.rootNodes[index2];

      if root1.height < root2.height:
        root1.parentNode = root2;
      elif root1.height > root2.height:
        root2.parentNode = root1;
      else:
        root2.parentNode = root1;
        root1.height = root1.height + 1;

    def makeSets(self, vertexList):
      for v in vertexList:
        self.makeSet(v);

    def makeSet(self, vertex):
      node = Node(0, len(self.rootNodes), None);
      vertex.parentNode = node;
      self.rootNodes.append(node);
      self.setCount = self.setCount + 1;
      self.nodeCount = self.nodeCount + 1;
```

```
In [22]: class KruskalAlgorithm(object):

    def spanningTree(self, vertexList, edgeList):
      disjointSet = DisjointSet(vertexList);
      spanningTree = [];

      edgeList.sort();


      for edge in edgeList:
        u = edge.startVertex;
        v = edge.targetVertex;

        if disjointSet.find(u.node) is not disjointSet.find(v.node):
          spanningTree.append(edge);
          disjointSet.merge(u.node, v.node);

      for edge in spanningTree:
```

```
                print(edge.startVertex.name, " - ", edge.targetVertex.name);
```

In [23]:
```
vertex1 = Vertex("a");
vertex2 = Vertex("b");
vertex3 = Vertex("c");
vertex4 = Vertex("d");
vertex5 = Vertex("e");
vertex6 = Vertex("f");
vertex7 = Vertex("g");

edge1 = Edge(2,vertex1, vertex2)
edge2 = Edge(6,vertex1, vertex3)
edge3 = Edge(5,vertex1, vertex5)
edge4 = Edge(10,vertex1, vertex6)
edge5 = Edge(3,vertex2, vertex4)
edge6 = Edge(3,vertex2, vertex5)
edge7 = Edge(1,vertex3, vertex4)
edge8 = Edge(2,vertex3, vertex6)
edge9 = Edge(4,vertex4, vertex5)
edge10 = Edge(5,vertex4, vertex7)
edge11 = Edge(5,vertex6, vertex7)

vertexList = [];
vertexList.append(vertex1);
vertexList.append(vertex2);
vertexList.append(vertex3);
vertexList.append(vertex4);
vertexList.append(vertex5);
vertexList.append(vertex6);
vertexList.append(vertex7);


edgeList = [];
edgeList.append(edge1);
edgeList.append(edge2);
edgeList.append(edge3);
edgeList.append(edge4);
edgeList.append(edge5);
edgeList.append(edge6);
edgeList.append(edge7);
edgeList.append(edge8);
edgeList.append(edge9);
edgeList.append(edge10);
edgeList.append(edge11);

algorithm = KruskalAlgorithm();
algorithm.spanningTree(vertexList, edgeList);
```

```
None
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-23-5bb47987b6b8> in <module>()
     43
     44 algorithm = KruskalAlgorithm();
---> 45 algorithm.spanningTree(vertexList, edgeList);


<ipython-input-22-88162a13eba6> in spanningTree(self, vertexList, edgeList)
     12         v = edge.targetVertex;
     13
---> 14         if disjointSet.find(u.node) is not disjointSet.find(v.node):
     15           spanningTree.append(edge);
     16           disjointSet.merge(u.node, v.node);


<ipython-input-21-4dca8e6dac36> in find(self, node)
     11       print(node);
     12
---> 13       while currentNode.parentNode is not None:
     14         currentNode = currentNode.parentNode;
     15


AttributeError: 'NoneType' object has no attribute 'parentNode'
```

```
In [ ]:
```

```
In [ ]:
```

0.3 Prim-Jarnik algorithm

- In Kruskal implementation we build the spanning tree separately, adding the smallest edge to the spanning tree if there is no cycle
- Prims algorithm we build the spanning tree from a given vertex, adding the samllest edge to the MST
- Kruskal -> edge based
- Prims -> vertex based!!!
- There are two implementations: lazy and eager
- Lazy implementation: add the new neighbour edges to the heap without deleting its content
- Eager implementation: we keep updating the heap if the distance from a vertex to the MST has changed
- Average running time: **O(E*logE) **but we need additional memory space** O(E)**

- Worst case: **O(E*logV)**

0.4 Prims VS Kruskal

- Prim's algorithm is significantly faster in the limit when you've got a really dense graph with many more edges than vertices
- Kruskal performs vetter in typical situations (sparse graphs) because it used simpler data structures
- Kruskal can have better performance if the dges can be sorted in linear time or the edges are already sorted
- Prim's better if the number of edges to vertices is high (dense graphs)

In [24]:
```
class Vertex(object):
  def __init__(self, name):
    self.name = name;
    self.visited = False;
    self.predecessor = None;
    self.adjacenciesList = [];

  def __str__(self):
    return self.name;

class Edge(object):

  def __init__(self, weight, startVertex, targetVertex):
    self.weight = weight;
    self.startVertex = startVertex;
    self.targetVertex = targetVertex;

  def __cmp__(self, otherEdge):
    return self.cmp(self.weight, otherEdge.weight)

  def __lt__(self, other):
    selfPriority = self.weight;
    otherPriority = other.weight;
    return selfPriority < otherPriority;
```

In [25]:
```
import heapq;

class PrimsJarrik(object):
  def __init__(self, unvisitedList):
    self.unvisitedList = unvisitedList;
    self.spanningTree = [];
    self.edgeHeap = [];
    self.fullCost = 0;

  def calculateSpanningTree(self, vertex):
    self.unvisitedList.remove(vertex)
```

```
        while self.unvisitedList:
          for edge in vertex.adjacenciesList:
            if edge.targetVertex in self.unvistedList:
              heapq.heappush(self.edgeHeap, edge)

            minEdge = heapq.heappop(self.edgeHeap);

            self.spanningTree.append(minEdge);
            print("Edge added to spanning tree: %s - %s"  % (minEdge.startVertex.name, minE
            self.fullCost = self.fullCost + minEdge.weight;

            vertex = minEdge.targetVertex;
            self.unvisitedList.remove(vertex);

      def getSpanningTree(self):
        return self.spanningTree;
```

In [27]:
```
node1 = Vertex("A")
node2 = Vertex("B")
node3 = Vertex("C")


edge1 = Edge(100, node1, node2);
edge2 = Edge(100, node2, node1);
edge3 = Edge(1000, node1, node3);
edge4 = Edge(1000, node4, node1);
edge5 = Edge(0.01, node3, node2);
edge6 = Edge(0.01, node2, node3);
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-27-767eaaa8b572> in <module>()
      7 edge2 = Edge(100, node2, node1);
      8 edge3 = Edge(1000, node1, node3);
----> 9 edge4 = Edge(1000, node4, node1);
     10 edge5 = Edge(0.01, node3, node2);
     11 edge6 = Edge(0.01, node2, node3);


NameError: name 'node4' is not defined
```

In [ ]: