

# MA/CSSE 473

## Day 36

Kruskal proof recap

Prim Data Structures  
and detailed  
algorithm.



### Recap: MST lemma

Let  $G$  be a weighted connected graph with a MST  $T$ ;  
let  $G'$  be any subgraph of  $T$ , and let  $C$  be any connected  
component of  $G'$ .

If we add to  $C$  an edge  $e=(v,w)$  that has minimum-  
weight among all edges that have one vertex in  $C$  and  
the other vertex not in  $C$ ,

**then  $G$  has an MST that contains the union of  $G'$  and  $e$ .**

[WLOG  $v$  is the vertex of  $e$  that is in  $C$ , and  $w$  is not in  $C$ ]

**Proof: We did it last time**



## Recall Kruskal's algorithm

- To find a MST:
- Start with a graph containing all of  $G$ 's  $n$  vertices and none of its edges.
- for  $i = 1$  to  $n - 1$ :
  - Among all of  $G$ 's edges that can be added without creating a cycle, add one that has minimal weight.

Does this algorithm produce an MST for  $G$ ?



## Does Kruskal produce a MST?

- Claim: After every step of Kruskal's algorithm, we have a set of edges that is part of an MST
- Base case ... **Work on the quiz questions with one or two other students**
- Induction step:
  - Induction Assumption: before adding an edge we have a subgraph of an MST
  - We must show that after adding the next edge we have a subgraph of an MST
  - Suppose that the most recently added edge is  $e = (v, w)$ .
  - Let  $C$  be the component (of the "before adding  $e$ " MST subgraph) that contains  $v$ 
    - Note that there must be such a component and that it is unique.
  - Are all of the conditions of MST lemma met?
  - Thus the new graph is a subgraph of an MST of  $G$



## Does Prim produce an MST?

- Proof similar to Kruskal.
- It's done in the textbook



## Recap: Prim's Algorithm for Minimal Spanning Tree

- Start with  $T$  as a single vertex of  $G$  (which *is* a MST for a single-node graph).
- for  $i = 1$  to  $n - 1$ :
  - Among all edges of  $G$  that connect a vertex in  $T$  to a vertex that is not yet in  $T$ , add to  $T$  a minimum-weight edge.

At each stage,  $T$  is a MST for a connected subgraph of  $G$ . **A simple idea; but how to do it efficiently?**

Many ideas in my presentation are from Johnsonbaugh, *Algorithms*, 2004, Pearson/Prentice Hall



## Main Data Structure for Prim

- Start with adjacency-list representation of  $G$
- Let  $V$  be all of the vertices of  $G$ , and let  $V_T$  the subset consisting of the vertices that we have placed in the tree so far
- We need a way to keep track of "fringe" edges
  - i.e. edges that have one vertex in  $V_T$  and the other vertex in  $V - V_T$
- Fringe edges need to be ordered by edge weight
  - E.g., in a priority queue
- What is the most efficient way to implement a priority queue?



## Prim detailed algorithm step 1

- **Create an indirect minheap** from the adjacency-list representation of  $G$ 
  - Each heap entry contains a vertex and its weight
  - **The vertices in the heap are those not yet in  $T$**
  - Weight associated with each vertex  $v$  is the minimum weight of an edge that connects  $v$  to some vertex in  $T$
  - **If there is no such edge,  $v$ 's weight is infinite**
    - **Initially all vertices except *start* are in heap, have infinite weight**
  - Vertices in the heap whose weights are not infinite are the *fringe vertices*
  - **Fringe vertices are candidates to be the next vertex (with its associated edge) added to the tree**



## Prim detailed algorithm step 2

- **Loop:**
  - Delete min weight vertex  $w$  from heap, add it to  $T$
  - We may then be able to decrease the weights associated with one or more vertices that are adjacent to  $w$



## Indirect minheap overview

- We need an operation that a standard binary heap doesn't support:
  - decrease(vertex, newWeight)**
    - Decreases the value associated with a heap element
    - We also want to quickly find an element in the heap
- Instead of putting vertices and associated edge weights directly in the heap:
  - Put them in an array called **key[]**
  - Put references to these keys in the heap



## Indirect Min Heap methods

operation	description	run time
init(key)	build a MinHeap from the array of keys	$\Theta(n)$
del()	delete and return the (location in key[ ] of the) minimum element	$\Theta(\log n)$
isIn(w)	is vertex w currently in the heap?	$\Theta(1)$
keyVal(w)	The weight associated with vertex w (minimum weight of an edge from that vertex to some adjacent vertex that is in the tree).	$\Theta(1)$
decrease(w, newWeight)	changes the weight associated with vertex w to newWeight (which must be smaller than w's current weight)	$\Theta(\log n)$



## Indirect MinHeap Representation

key array	15	70	7	85	92	10	19	63
into array	2	8	1	7	5	3	6	4
outof array	3	1	6	8	5	7	4	2

Draw the tree diagram of the heap

- outof[i] tells us which key is in location i in the heap
- into[j] tells us where in the heap key[j] resides
- into[outof[i]] = i, and outof[into[j]] = j.
- To swap the 15 and 63 (not that we'd want to do this):

```
temp = outof[2]
outof[2] = outof[4]
outof[4] = temp
```

```
temp = into[outof[2]]
into[outof[2]] = into[outof[4]]
into[outof[4]] = temp
```



## MinHeap class, part 1

```
class MinHeap:
    """ Implements an indirect heap so it can efficiently support
        the Isin and Decrease operations that are not
        supported efficiently by an ordinary binary heap. """
    def __init__(self, key):
        """key: list of values from which we build initial heap"""
        self.n = len(key)-1
        self.key = key
        self.into = [i for i in range(self.n + 1)]
        self.outof = [i for i in range(self.n + 1)]
        self.heapify()
    def heapify(self):
        for i in range(self.n//2, 0, -1):
            self.siftdown(i, self.n)
```



## MinHeap class, part 2

```
def siftdown(self, i, n):
    """ sift down for a minHeap. i is the
        heap index, outof[i] is index into key array) """
    s = self.outof[i]
    temp = self.key[s]
    while 2*i <= n:
        c = 2*i # c is for child
        if c < n and self.key[self.outof[c+1]] < \
            self.key[self.outof[c]]:
            c += 1
        if self.key[self.outof[c]] < temp:
            self.outof[i] = self.outof[c]
            self.into[self.outof[i]] = i
        else:
            break
    i = c
    self.outof[i] = s
    self.into[s] = i
```



## MinHeap class, part 3

```
def delete(self):
    """delete the minimum value and return it"""
    result = self.outof[1]
    temp = self.outof[1]
    self.outof[1] = self.outof[self.n]
    self.into[self.outof[1]] = 1
    self.outof[self.n] = temp
    self.into[temp] = self.n
    self.n -= 1
    self.siftdown(1, self.n)
    return result

def isIn(self, w):
    """ returns True iff key[w] is in this heap """
    return self.into[w] <= self.n

def keyVal(self, w):
    """ returns the weight corresponding to w"""
    return self.key[w]
```

## MinHeap class, part 4

```
def decrease(self, w, newWeight):
    """ change the weight corresponding to
    vertex w to newWeight (which must be no
    larger than its current weight) """
    # p is for parent, c is for child
    self.key[w] = newWeight
    c = self.into[w]
    p = c//2
    while p >= 1:
        if self.key[self.outof[p]] <= newWeight:
            break
        self.outof[c] = self.outof[p]
        self.into[self.outof[c]] = c
        c = p
        p = c//2
    self.outof[c] = w
    self.into[w] = c
```

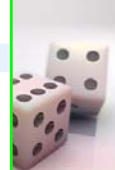


## Prim Algorithm

```
INFINITY = 1234567890
VERTEX = 0 # An edge is a list of two numbers:
WEIGHT = 1 # These are what the subscripts (0 and 1) mean.

def prim(adj, start):
    """ parent[v] = parent of v in MST rooted at start """
    n = adj.length() # vertices in graph
    key = [None] + [INFINITY]*n # later they will be decreased
    parent = [None] + [0]*n # placeholders
    key[start] = 0
    parent[start] = 0
    heap = MinHeap(key) # non-infinity value in heap represents fringe vertex
    for i in range(1, n+1):
        v = heap.delete()
        edges = adj.getList(v) # all vertices adjacent to v
        for edge in edges: # an edge is a list of: other vertex and weight
            w = edge[VERTEX]
            if heap.isIn(w) and edge[WEIGHT] < heap.keyVal(w):
                parent[w] = v
                heap.decrease(w, edge[WEIGHT])
    return parent

def edgeListFromParentArray(parent):
    result = []
    for i in range(1, len(parent)):
        if parent[i] > 0:
            result.append([parent[i], i])
    return result
```




## AdjacencyListGraph class

```
class AdjacencyListGraph:
    def __init__(self, adjlist):
        self.vertexList = [v[0] for v in adjlist]
        self.adjacencyList = [Vertex(v) for v in self.vertexList]
        for v in adjlist:
            self.setVertex(v[0], v[1])

    def getList(self, v):
        for ver in self.adjacencyList:
            if ver.v == v:
                return ver.adj
        return None

    def length(self):
        return len(self.adjacencyList)

    def setVertex(self, v, vList):
        i = self.vertexList.index(v)
        for v in vList:
            if v[0] not in self.vertexList:
                print "Illegal vertex in graph"
                exit()
            self.adjacencyList[i].add(v)
```



## Preview: Data Structures for Kruskal

- A sorted list of edges (edge list, not adjacency list)
- Disjoint subsets of vertices, representing the connected components at each stage.
  - Start with  $n$  subsets, each containing one vertex.
  - End with one subset containing all vertices.
- Disjoint Set ADT has 3 operations:
  - **makeset(i)**: creates a singleton set containing  $i$ .
  - **findset(i)**: returns a "canonical" member of its subset.
    - I.e., if  $i$  and  $j$  are elements of the same subset,  $\text{findset}(i) == \text{findset}(j)$
  - **union(i, j)**: merges the subsets containing  $i$  and  $j$  into a single subset.



Q37-1