# MA/CSSE 473 Day 38

**Prim algorithm finale**

**Kruskal Data Structures and detailed algorithm**

**Disjoint Set ADT**

## MA/CSSE 473 Day 38

- HW 14 due Tomorrow at 8 AM
- HW 15 Due Friday (it's a little different!)
- Fill out the Course evaluation form
  - If everybody in a section does it, everyone in that section gets 10 bonus points on the Final Exam
  - I can't see **who** has completed the evaluation, but I can see **how many**
- Final Exam Wednesday evening, Nov 17
- **Student Questions**
- Prim data structures and detailed algorithm
- Kruskal data structures and detailed algorithm

# 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**

# Day 37 Quiz Questions

- To find a MST for G:
- Start with a graph G' containing all of the n vertices of G and no 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.

How do we know that v was already part of some connected component of G'?

Does the addition of e to C satisfy the hypothesis of the lemma? For each statement below, explain why it is true.

G' is a subgraph of some MST for G:

C is a connected component of G':

e connects a vertex in C to an vertex in G – C:

e satisfies the minimum-weight condition of the lemma:

# 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.
- What data structures do we need in order to implement this efficiently?
- Last time we came up with
  - adjacency list representation of the graph
  - minheap

# Prim detailed algorithm summary

- **Create a minheap from adjacency-list** representation of G
  - Each element contains a vertex and its weight
  - Vertices in the heap are in  not yet in T
  - Weight associated with each vertex v is the minimum weight of an edge that connects v to a vertex in T
  - If there is no such edge, v's weight is infinite
    - Initially all vertices except start 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
- **Loop:**
  - Delete min weight vertex from heap, add it to T
  - we may be able to decrease the weights associated with one or vertices that are adjacent to v.

# Prim Algorithm

```
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)
    for i in range(1, n+1):
        v = heap.delete()
        edges = adj.getList(v)
        for edge in edges:
            w = edge[VERTEX]
            if heap.isIn(w) and edge[WEIGHT] < heap.keyVal(w):
                parent[w] = v
                heap.decrease(w, edge[WEIGHT])
    return parent
```

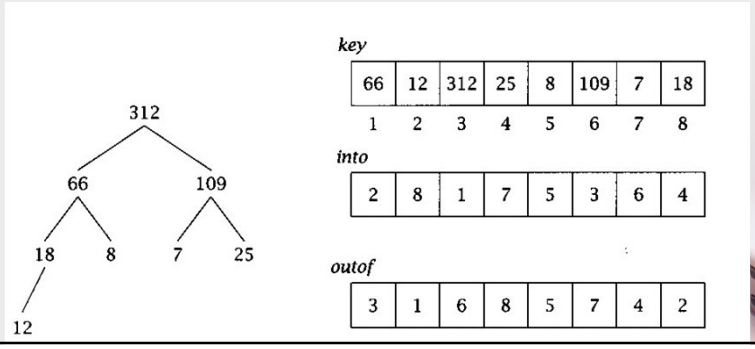
```
def edgeListFromParentArray(parent):
    result = []
    for i in range(1, len(parent)):
        if parent[i] > 0:
            result.append([parent[i], i])
    return result
```

# AdjacencyListGraph class

```
class AdjancencyListGraph:
    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)
```

# MinHeap implementation

- An indirect heap. We keep the keys in place in an array, and use another array, "outof", to hold the positions of these keys within the heap.
- To make lookup faster, another array, "into" tells where to find an element in the heap.
- i = into[j] iff j = out of[i]
- Picture shows it for a maxHeap, but the idea is the same:




Q1

# MinHeap methods

| operation | description | run time |
|---|---|---|
| init(key) | build a MinHeap from the array of keys | Θ(n) |
| del() | delete and return the (location in key[ ] of the) minimum element | Θ(log n) |
| isIn(w) | is vertex w currently in the heap? | Θ(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). | Θ(1) |
| decrease(w, newWeight) | changes the weight associated with vertex w to newWeight (which must be smaller than w's current weight) | Θ(log n) |

# MinHeap code part 1

```
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)

def siftdown(self, i, n):
    """ sift down for a minHeap.
    i is the heap index,(not the index into the 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
```

We will not discuss the details in class; the code is mainly here so we can look at it and see that the running times for the various methods are as advertised

# MinHeap code part 2

```
def delete(self):
    """delete the mimimum value from this heap, returning its value"""
    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 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]
```

delete could be simpler; I kept pointers to the deleted nodes around, to make it easy to implement Heapsort later. N calls to delete() leave the outof array in indirect reverse sorted order.

# MinHeap code part 3

```
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
```

# 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, findset(i) == findset(j)
  - union(i, j): merges the subsets containing i and j into a single subset.

Q2

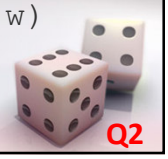
# Kruskal Algorithm

Assume vertices are numbered 1...n
(n = |V|)

Sort edge list by weight (increasing order)

```
for i = 1..n: makeset(i)
i, count, result = 1, 0, []

while count < n-1:
  if findset(edgelist[i].v) !=
          findset(edgelist[i].w):
    result += [edgelist[i]]
    count += 1
    union(edgelist[i].v, edgelist[i].w)
  i += 1
return result
```

What can we say about efficiency of this algorithm (in terms of |V| and |E|)?



Q2

# Disjoint set ADT

- A collection of numbers (taken from the set {1, . . , n} ) are partitioned into disjoint sets, each containing a (marked) representative element
- Operations:
  - makeset(i): Create a singleton set containing the number i
  - findset(i): Find the unique representative of the set that contains i. Note that i and j are in the same set iff
    findset(i) == findset(j)
  - union(i, j): Combine the sets containing i and j into a single set (before the union, i and j must be in different sets subsets)

# Example of operations

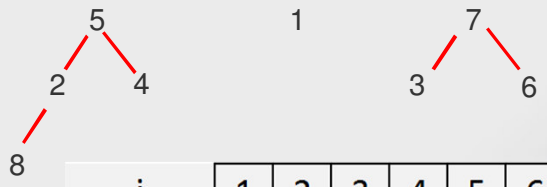
- makeset (1)
- makeset (2)
- makeset (3)
- makeset (4)
- makeset (5)
- makeset (6)

- union(4, 6)
- union (1,3)
- union(4, 5)
- findset(2)
- findset(5)

What are the sets after these operations?

# Set Representation

- Each disjoint set is a tree, with the "marked" element as its root
- Efficient representation of the trees:
  - an array called *parent*
  - parent[i] contains the index of i's parent.
  - If i is a root, parent[i]=i




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| parent[i] | 1 | 5 | 7 | 5 | 5 | 7 | 7 |

Q4

# Using this representation

- makeset(i):
- findset(i):
- mergetrees(i,j):
  - assume that i and j are the marked elements from different sets.
- union(i,j):
  - assume that i and j are elements from different sets

```
def makeset1(i):
    parent[i] = i
```

```
def findset1(i):
    while i != parent[i]:
        i = parent[i]
    return i
```

```
def mergetrees1(i,j):
    parent[i] = j
```

```
def union1(i,j):
    mergetrees1(findset1(i), findset1(j))
```

Q5

# Analysis

- Assume that we are going to do n makeset operations followed by m union/find operations
- time for makeset?
- worst case time for findset?
- worst case time for union?
- Worst case for all m union/find operations?
- worst case for total?
- What if m < n?
- Write the formula to use min

## Can we keep the trees from growing so fast?

- Make the shorter tree the child of the taller one
- What do we need to add to the representation?
- rewrite makeset, mergetrees.

```
def makeset2(i):
   parent[i] = i
   height[i] = 0
```

```
def mergetrees2(i,j):
   if height[i] < height[j]):
         parent[i] = j
   elif height[i] > height[j]:
      parent[j] = i
   else:
      parent[i] = j
      height[j] = height[j] + 1
```

- findset & union are unchanged.
- What can we say about the maximum height of a k-node tree?

## Theorem: max height of a k-node tree T produced by these algorithms is $\lfloor \lg k \rfloor$

- Base case…
- Induction hypothesis…
- Induction step:
  - Let T be a k-node tree
  - T is the union of two trees:
    $T_1$ with $k_1$ nodes and height $h_1$
    $T_2$ with $k_2$ nodesand height $h_2$
  - What can we about the heights of these trees?
  - Case 1: $h_1 \neq h_2$. Height of T is
  - Case 2: $h_1 = h_2$. WLOG Assume $k_1 \geq k_2$. Then $k_2 \leq k/2$. Height of tree is $1 + h2 \leq \ldots$

Q4

## Worst-case running time

- Again, assume n makeset operations, followed by m union/find operations.
- If m > n
- If m < n

## Speed it up a little more

- **Path compression:** Whenever we do a findset operation, change the parent pointer of each node that we pass through on the way to the root so that it now points directly to the root.
- Replace the height array by a rank array, since it now is only an upper bound for the height.
- Look at makeset, findset, mergetrees (on next slides)

# Makeset

This algorithm represents the set {*i*} as a one-node tree and initializes its rank to 0.

```
def makeset3(i):
   parent[i] = i
   rank[i] = 0
```

# Findset

- This algorithm returns the root of the tree to which i belongs and makes every node on the path from i to the root (except the root itself) a child of the root.

```
def findset(i):
   root = i
   while root != parent[root]:
      root = parent[root]
   j = parent[i]
   while j != root:
      parent[i] = root
      i = j
      j = parent[i]
   return root
```

# Mergetrees

This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller rank a child of the other root. If the trees have the same rank, we arbitrarily make the root of the first tree a child of the other root.

```
def mergetrees(i,j) :
   if rank[i] < rank[j]:
      parent[i] = j
   elif rank[i] > rank[j]:
      parent[j] = i
   else:
      parent[i] = j
      rank[j] = rank[j] + 1
```

# Analysis

- It's complicated!
- R.E. Tarjan proved (1975)*:
  - Let t = m + n
  - Worst case running time is Θ(t α(t, n)), where α is a function with an *extremely* slow growth rate.
  - Tarjan's α:
  - $\alpha(t, n) \leq 4$ for all $n \leq 10^{19728}$
- Thus the amortized time for each operation is essentially constant time.

* According to *Algorithms* by R. Johnsonbaugh and M. Schaefer, 2004, Prentice-Hall, pages 160-161