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 …
- 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

Work on the quiz questions with one or two other students

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 vertices"
  - i.e. edges that have one vertex in V_T and the other vertex in $V - V_T$
- Fringe vertices 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	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)

Indirect MinHeap Representation

key array	15	70	7	85	92	10	19	63
	0	1	2	3	4	5	6	7	8
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 mimimum 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
```

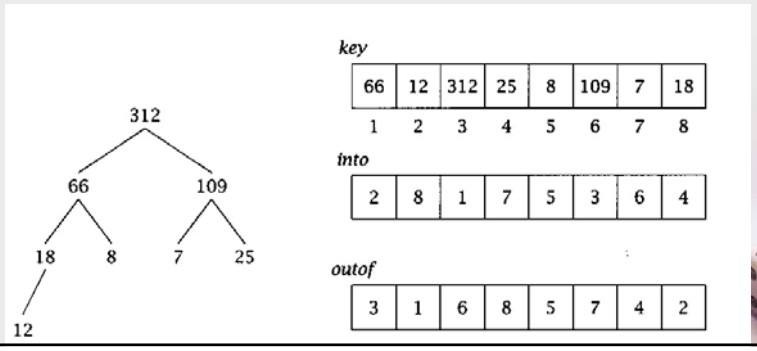
```
def edgeListFromParentArray(parent):
    result = []
    for i in range(1, len(parent)):
        if parent[i] > 0:
            result.append([parent[i], i])
    return result
```

AdjacencyListGraph class

```
class 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:



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

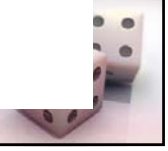
MinHeap code part 2

```
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]
```

NOTE: delete could be simpler, but 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
```



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

Q37-1

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?

Kruskal Algorithm

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

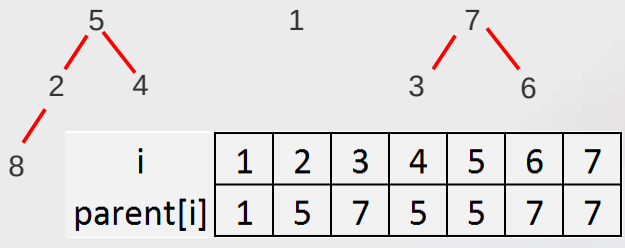
Sort edge list by weight (increasing order)

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

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

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

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

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

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

Q37-5

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 nodes and height h_2
  - What can we say 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 $1 + h2 \leq \ldots$

Q37-5

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