MA/CSSE 473
Day 36

Student Questions

More on Minimal Spanning Trees

Kruskal

Prim

Kruskal and Prim

ALGORITHMS FOR FINDING A MINIMAL SPANNING TREE

Kruskal's algorithm

- To find a MST (minimal Spanning Tree):
- Start with a graph T containing all n of G's 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 to T an edge that has minimal weight.
  - Details of Data Structures later

Prim's algorithm

- 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 a minimum-weight edge (and the vertex at the other end of T).
  - Details of Data Structures later

MST lemma

- Let G be a weighted connected graph,
- let T be any MST of G,
- let G′ be any nonempty subgraph of T, and
- let C be any connected component of G′.
- Then:
  - 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,
  - 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]

Summary: If G' is a subgraph of an MST, so is G'∪{e}

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:

✓If e is in T, we are done, so we assume that e is not in T.

✓Since T does not contain edge e, adding e to T creates a cycle.

✓Removing any edge of that cycle from T∪{e} gives us another spanning tree.

✓If we want that tree to be a *minimal* spanning tree for G that contains G' and e, we must choose the "removable" edge carefully.

✓Details on next page…

Choosing the edge to remove

- ✓ Along the unique simple path in T from v to w, let w′ be the first vertex that is not in C, and let v′ be the vertex immediately before it.
- ✓ Then e′ = (v′, w′) is also an edge from C to G-C.
- ✓ Note that by the minimal-weight choice of e, weight(e′) ≥ weight(e) .
- ✓ Let T′ be the (spanning) tree obtained from T by removing e′ and adding e.
- ✓ Note that the removed edge is **not** in G′,
- ✓ Because e and e′ are the only edges that are different, weight(T) ≥ weight(T′).
- ✓ Because T is a MST, weight(T) ≤ weight(T′).
- ✓ Thus the weights are equal, and T′ is an MST containing G′ and e, which is what we wanted.

Recap: MST lemma

Let G be a weighted connected graph with an 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*.

Recall Kruskal's algorithm

- To find a MST for G:
  - Start with a connected weighted 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 actually 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 of G
- Proof of claim: 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
  - Details:

Does Prim produce an MST?

- Proof similar to Kruskal (but slightly simpler)
- 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

We now examine Prim more closely

Main Data Structures 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 summary

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

MinHeap overview

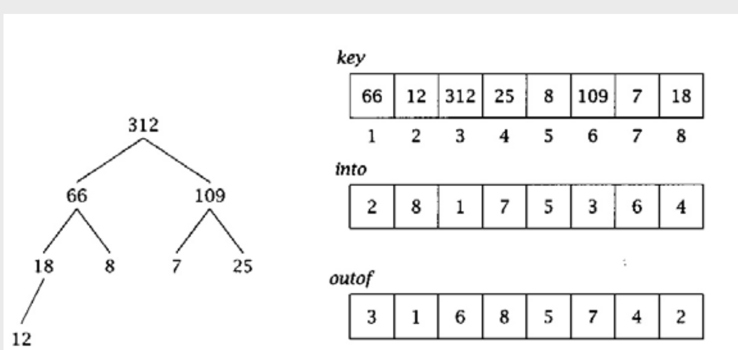
- We need an operation that a standard binary heap doesn't support:
  `decrease(vertex, newWeight)`
  - Decreases the value associated with a heap element
- Instead of putting vertices and associated edge weights directly in the heap:
  - Put them in an array called **key[]**
  - Put references to them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

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)

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
```

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
```

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)
```

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)

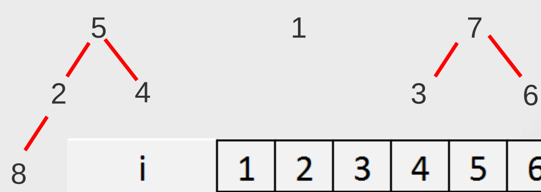
```
for i = 1..n: makeset(i)
i, count, tree = 1, 0, []

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