

MA/CSSE 473

Day 38

Prim Data Structures
Dijkstra



MA/CSSE 473 Day 38

- HW 15 due now
- HW 16 due Friday
 - It was explained in class on Day 36
- Don't forget the Boyer-Moore demonstration animation, due Nov 20
- **Today:**
- Prim's Algorithm
- Efficient MinHeap implementation
- (Dijkstra's algorithm)



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



Data Structures for Prim

- Start with adjacency-list representation of G
 - How is this different than Kruskal?
- Let V be all of the vertices of G , and V_T the subset consisting of the vertices that we have placed in the tree so far
- Define E and E_T similarly
- Let 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 $V - V_T$
- Need to be ordered by edge weight
- I.e. a priority queue
- What is the most efficient way to implement a priority queue?



Prim's algorithm

- Take the data structures into account
- Work with another student to write it down
- Minheap operations:

operation	description	run time
init(keys)	build a heap from the array of keys	$\Theta(n)$
del()	delete and return the (location in keys[] 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	$\Theta(1)$
decrease(w, newWeight)	changes the weight associated with vertex w to newWeight (must be smaller than the current weight)	$\Theta(\log n)$

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 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])

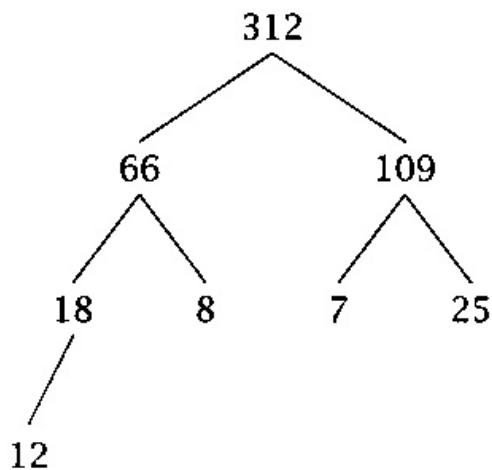
    def getList(self, v):
        for ver in self.adjacencyList:
            if ver.v == v:
                return ver.adj
        return None

    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)
```

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.
- $i = \text{into}[j]$ iff $j = \text{out of}[i]$
- Picture shows it for a maxHeap, but the idea is the same:



key

66	12	312	25	8	109	7	18
----	----	-----	----	---	-----	---	----

1 2 3 4 5 6 7 8

into

2	8	1	7	5	3	6	4
---	---	---	---	---	---	---	---

outof

3	1	6	8	5	7	4	2
---	---	---	---	---	---	---	---



```

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 1



MinHeap code part 2

```
def delete(self):
    """delete the minimum 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 out of() 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.inout[w]
    p = c/2
    while p >= 1:
        if self.key[self.outof[p]] <= newWeight:
            break
        self.outof[c] = self.outof[p]
        self.inout[self.outof[c]] = c
        c = p
        p = c/2
    self.outof[c] = w
    self.inout[w] = c
```

Dijkstra's Algorithm

- For a selected vertex, (call it *start*) in a connected, weighted graph G , find a shortest (minimum total weight) path from start to each other vertex
- Assumption: All weights are positive



Dijkstra's algorithm approach

- The shortest path from vertex v to vertex w is either
 - empty, if $v = w$, or
 - obtained by adding an edge (u, w) to a shortest path from v to u .



Dijkstra's algorithm start-up

- Begin with a subgraph that is only the starting vertex.
- Among all edges (start, v) , choose the one with the smallest weight, and add it (along with the edge that connects it) to our subgraph. Clearly (start, v) is the shortest path from start to v .



Dijkstra's algorithm continues

- Among all vertices not in a path yet, and which are also adjacent to a node that is in the path, choose one with minimal path length from start, and add it to the path.



Dijkstra's algorithm data structures

- A predecessor array that is similar the parent array in the Prim's algorithm.
- A minheap that stores “unpathed” nodes as keys and minimum path length (as extension of minimum paths that have been discovered already) as weights.
 - Use an indirect binary heap, just as we did for Prim.



Dijkstra's algorithm details 1

First, the part that's just like Prim:

Input Parameters: *adj, start*

Output Parameters: *parent*

```
dijkstra(adj, start, parent) {
```

```
    n = adj.last
```

```
    for i = 1 to n
```

```
        key[i] = ∞ // key is a local array
```

```
    key[start] = 0
```

```
    parent[start] = 0
```

```
    ...
```

```
    // the following statement initializes the
```

```
    // container h to the values in the array key
```

```
    h.init(key, n)
```

I ran out of time and did not get to actually implement this one.



Dijkstra's algorithm details 2

The part that is different than Prim:

```
for  $i = 1$  to  $n$  {  
     $v = h.min\_weight\_index()$   
     $min\_cost = h.keyval(v)$   
     $v = h.del()$   
     $ref = adj[v]$   
    while ( $ref \neq null$ ) {  
         $w = ref.ver$   
        if ( $h.isin(w) \ \&\&$   
             $min\_cost + ref.weight < h.keyval(w)$ ) {  
             $parent[w] = v$   
             $h.decrease(w, min\_cost + ref.weight)$   
        } // end if  
         $ref = ref.next$   
    } // end while  
} // end for  
}
```

