

MA/CSSE 473

Day 37

Disjoint Sets

Prim Data Structures



MA/CSSE 473 Day 37

- HW 15 due Tuesday
- HW 16 due Friday
 - It was explained in class in Day 36
- Don't forget the Boyer-Moore demonstration animation, due Nov 20
- **Student Questions**
- Efficient Disjoint Sets
- Data Structures for Prim's algorithm



Data Structures for Kruskal

- A sorted list of edges
- 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 set.



Kruskal Algorithm

Assume vertices are numbered 1...n

Sort edgelist by weight

```
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 n and m)?



Disjoint set ADT

- A collection of numbers (taken from the set $\{1, \dots, n\}$) 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 for the set that contains i . Note that i and j are in the same set iff **$\text{findset}(i) == \text{findset}(j)$**
 - **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)



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 i 's parent.
 - If i is the root, $parent[i]=i$
- Show the parent array for the previous example
- The parent array contains all of the info that we need for our first representation



Using this representation

```
def makeset1(i):
```

- `makeset(i):` `parent[i] = i`

- `findset(i):`

```
def findset1(i):
```

```
    while i != parent[i]:
```

```
        i = parent[i]
```

```
    return i
```

- `mergetrees(i,j):`

- assume that i and j are the marked elements from different sets.

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

```
    parent[i] = j
```

- `union(i,j)`

- assume that i and j are elements from different sets

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

```
    mergetrees1(findset1(i), findset1(j))
```



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?



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 $\Theta(t \alpha(t, n))$, 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



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?



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

