6,8:15

MA/CSSE 473
Day 37

Student Questions

Kruskal Data Structures and detailed algorithm

Disjoint Set ADT

Data Structures for Kruskal

- A sorted list of edges (edge list, not adjacency list)
  - Edge e has fields `e.v` and `e.w` (#s of its end vertices)
- 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 vertex i.
  - **findset(i)**: returns the "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.

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)

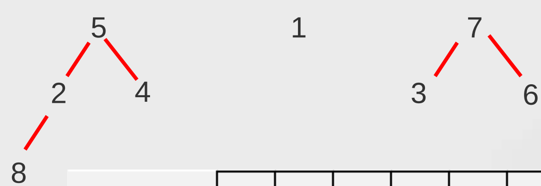
```
for i = 1..n:
  makeset(i)
i, count, result = 1, 0, []

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=|V| and m=|E|)?

Implement Disjoint Set ADT

- Each disjoint set is a tree, with the "marked" (canonical) element as its root
- Efficient representation of these 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):

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

- findset(i):

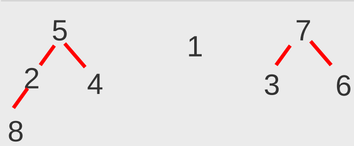
```
def findset1(i):
    while i != parent[i]:
        i = parent[i]
    return i
```

- 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))
```



Write these procedures on the board

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

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 nodes and 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 $1 + h2 \leq$ …

    Added after class because we did not get to it:

  $1 + h2 <= 1 + \lfloor \lg k_2 \rfloor <= 1 + \lfloor \lg k/2 \rfloor = 1 + \lfloor \lg k - 1 \rfloor = \lfloor \lg k \rfloor$

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