CS 1501
www.cs.pitt.edu/~nlf4/cs1501/
Union Find
Dynamic connectivity problem

- For a given graph $G$, can we determine whether or not two vertices are connected in $G$?
- Can also be viewed as checking subset membership
- Important for many practical applications
- We will solve this problem using a union/find data structure
A simple approach

- Have an id array simply store the component id for each item in the union/find structure
  - How do we determine if two vertices are connected?
  - How do we establish the connected components?
    - Add graph edges one at a time to UF data structure using union operations
Example

U(2, 0)
U(4, 7)
U(1, 2)
U(3, 2)
U(4, 5)
U(5, 7)
U(6, 3)

ID:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Analysis of our simple approach

- Runtime?
  - To find if two vertices are connected?
  - For a union operation?
Union Find API

- `__init__(self, n)`
  - Initialize with n items numbered 0 to n-1

- `union(self, p, q)`
  - Connect p with q

- `find(self, p)`
  - Return id of the connected component that p is in

- `connected (self, p, q)`
  - True if p and q are connected

- `count(self)`
  - Number of connected components
def count(self):
    return self.count

def connected(self, p, q):
    return self.find(p) == self.find(q)
def __init__(self, n):
    self.count = n
    self.id = [i for i in range(n)]

def find(self, p):
    return self.id[p]

def union(self, p, q):
    pID = self.find(p)
    qID = self.find(q)
    if pID == qID:
        return
    for i in range(len(self.id)):
        if self.id[i] == pID:
            self.id[i] = qID
    count -= 1
Kruskal’s algorithm

- With this knowledge of union/find, how, exactly can it be used as a part of Kruskal’s algorithm?
  - What is the runtime of Kruskal’s algorithm?
Kruskal's example revisited

PQ:
1: (0, 2)
2: (3, 5)
3: (1, 4)
4: (2, 5)
5: (2, 3)
5: (0, 3)
5: (1, 2)
6: (0, 1)
6: (2, 4)
6: (4, 5)
Can we improve on union()’s runtime?

- What if we store our connected components as a forest of trees?
  - Each tree representing a different connected component
  - Every time a new connection is made, we simply make one tree the child of another
Tree example
Implementation using the same id array

def find(self, p):
    while p != self.id[p]:
        p = self.id[p]
    return p

def union(self, p, q):
    i = self.find(p)
    j = self.find(q)
    if i == j:
        return
    self.id[i] = j
    self.count -= 1
Forest of trees implementation analysis

- Runtime?
  - find():
    - Bound by the height of the tree
  - union():
    - Bound by the height of the tree
- What is the max height of the tree?
  - Can we modify our approach to cap its max height?
Weighted tree example
def __init__(self, n):
    self.count = n
    self.id = [i for i in range(n)]
    self.sz = [1 for i in range(n)]

def union(self, p, q):
    i = self.find(p)
    j = self.find(q)
    if (i == j)
        return
    if self.sz[i] < self.sz[j]:
        self.id[i] = j
        self.sz[j] += self.sz[i]
    else:
        self.id[j] = i
        self.sz[i] += self.sz[j]
    self.count -= 1

Weighted trees
Weighted tree approach analysis

- Runtime?
  - find()?
  - union()?

- Can we do any better?
What is the runtime of Kruskal’s algorithm?
Path Compression

- find(4)
  - 4
  - 5
- find(0)
  - 0
  - 2