Application of Union Find in Undirected Graph Algorithm Problems

Neo Hao

https://github.com/Neo-Hao/union-find

Union Find / Disjoint Set

Union-find is a data structure that keeps track of a set of elements partitioned into a number of disjoint subsets. It typically has three operations:

- ***makeSet***: Create a set using the only one element that is given
- ***findSet***: Determine which subset a particular element is in. *findSet* typically returns an item from this set that serves as its "representative"; by comparing the result of two Find operations, one can determine whether two elements are in the same subset.
- ***union***: Join two subsets into a single subset.

Time Complexity:

- Naïve Approach: O(n) for per *findSet* and *union* operation.
- Optimized approach: Amortized O(α(n)) per operation, where α(n) is less than 5 for all remotely practical values of n.

Naïve Approach:

```
makeSet(v):
       n = Node(v)
       n.parent = n

findSet(n):
    if n.parent == n:
        return n
    return findSet(n.parent)

union(n1, n2):
       i = findSet(n1)
       j = findSet(n2)
       if i == j:
              return
       else:
              j.parent = i
```

```
class Node(object):
    def __init__(self, v):
        self.v = v
        self.parent = None

class Union(object):
    def __init__(self):
        self.table = {}

    # makeSet method
    def makeSet(self, v):
        # input: vertex
        node = Node(v)
        node.parent = node
        self.table[v] = node

    # union method
    def union(self, v1, v2):
        # input: vertex1, vertex2
        i = self.findSet(v1)
        j = self.findSet(v2)
        if i == j:
            return
        j.parent = i

    # findSet method
    def findSet(self, v):
        # input vertex
        n = self.table[v]
```

```python
        return self.findSetHelper(n)

    def findSetHelper(self, n):
        # input node
        if n.parent == n:
            return n
        return self.findSetHelper(n.parent)
```

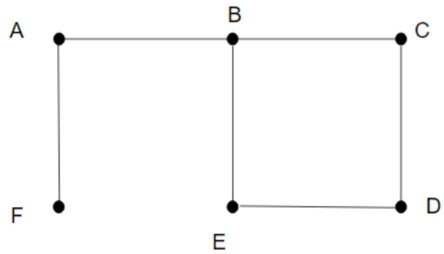
Optimized Approach:

```
makeSet(v):
        n = Node(v)
        n.parent = n

union(n1, n2):
        i = findSet(n1)
        j = findSet(n2)
        if i == j: return
        if i.rank > j.rank:
             j.parent = i
          elif i.rank < j.rank:
             i.parent = j
          else:
             j.parent = i
             i.rank += 1

findSet(n):
    if n.parent != n:
        n.parent =
            findSet(n.parent)
     return n.parent
```

```python
class Node(object):
    def __init__(self, v):
        self.v = v
        self.parent = None
        self.rank = 0

class Union(object):
    def __init__(self):
        self.table = {}

    # makeSet method
    def makeSet(self, val):
        node = Node(val)
        node.parent = node
        self.table[val] = node

    # union method
    def union(self, v1, v2):
        i = self.findSet(v1)
        j = self.findSet(v2)

        if i == j: return

        if i.rank > j.rank:
            j.parent = i
        elif i.rank < j.rank:
            i.parent = j
        else:
            j.parent = i
            i.rank += 1

    # findSet method
    def findSet(self, v):
        node = self.table[v]
        return self.findSetHelper(node)

    def findSetHelper(self, n):
        # representative itself
        if n == n.parent:
            return n
        n.parent = self.findSetHelper(n.parent)
        return n.parent
```

Cycle Detection

Cycle detection refers to the algorithmic problem of finding a cycle in a sequence of iterated function values.



Implementations:

<table>
<tr>
<td>

```
cycleDetect(G):
   A = ∅
   foreach v ∈ G.V:
      MAKE-SET(v)
   foreach (u, v) in G.E:
       if FIND-SET(u) ≠ FIND-SET(v):
          UNION(u, v)
       else:
          return True
   return False
```

</td>
<td>

```
def cycleDetection(edges):
   union = Union()
   # step 1: makeSet
   for u, v in edges:
      union.makeSet(u)
      union.makeSet(v)
   # step 2: traverse the edges
   for u, v in edges:
      if union.findSet(u) ==
         union.findSet(v):
         return True
      else:
         union.union(u, v)
   return False
```

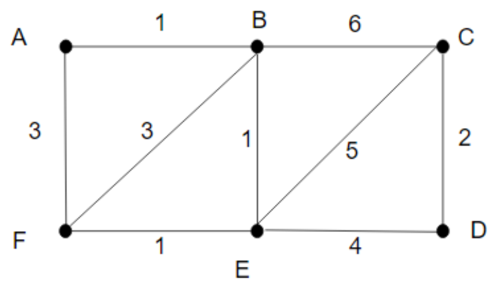
</td>
</tr>
</table>

Time Complexity: *O(V)*

Kruskal's algorithm

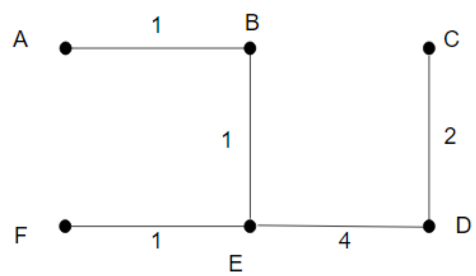
Weighted Graph:

A weighted graph refers to an edge-weighted graph, where edges have weights or values.



Minimum Spanning Tree:

A minimum spanning tree is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Implementation:

```
KRUSKAL(G):
   result = ∅
   foreach v ∈ G.V:
      MAKE-SET(v)
   foreach (u, v) in G.E ordered by
     increasing order of weight(u, v):
      if FIND-SET(u) ≠ FIND-SET(v):
         result = result ∪ {(u, v)}
         UNION(u, v)
   return result
```

```
def kruskal(weightedEdges):
   union = Union()
   result = []
   # step 1
   for u, v in weightedEdges:
      union.makeSet(v)
   # step 2
   weightedEdges.sort()
   # step 3
   for w, u, v in weightedEdges:
      if union.findSet(u) !=
        union.findSet(v):
         result.append((u, v))
         union.union(u, v)

   return result
```

Time Complexity: *O(E log E)*

Q1. Number of Connected Components in an Undirected Graph

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

Example 1:

```
0          3
|          |
1 --- 2    4
```

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], return 2.

Example 2:

```
0           4
|           |
1 --- 2 --- 3
```

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3, 4]], return 1.

Requirement:

```
def countComponents(self, n, edges):
    """
    :type n: int
    :type edges: List[List[int]]
    :rtype: int
    """
```

Q2. Number of Islands

A 2d grid map of `m` rows and `n` columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each *addLand* operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example:

Given m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]].
Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).

```
0 0 0

0 0 0

0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1 0 0

0 0 0   Number of islands = 1

0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1 1 0

0 0 0   Number of islands = 1

0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1 1 0

0 0 1   Number of islands = 2

0 0 0
```

We return the result as an array: [1, 1, 2].

Requirement:

```
    def numIslands2(self, m, n, positions):
        """
        :type m: int
        :type n: int
        :type positions: List[List[int]]
        :rtype: List[int]
        """
```

Q3. Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Requirement:

```
def validTree(self, n, edges):
    """
    :type n: int
    :type edges: List[List[int]]
    :rtype: bool
    """
```