============================================================================

CSC 263 Tutorial 10 Winter 2019

============================================================================

1. Given a weighted connected undirected graph G = (V, E) with n vertices and m edges, we define the **Minimum Cycler** of G as a set S of edges of minimum total weight such that every cycle in G contains at least one edge in S.

也就是G里的每个cycle都至少有一个minimum weight的edge放进set S。Devise an algorithm that finds the Minimum Cycler in O(m logn) time, and justify its correctness and runtime.

Hint: The algorithm can be described in one short sentence.

answer:

Find the \*maximum\* spanning tree T for G using Prim's or Kruskal's algorithm  
(it's like the algorithm for finding minimum spanning trees but changing all choices from choosing the minimum to choosing the maximum), then the Minimum Cycler is simply the set of all edges NOT in T, i.e., S = E \ T.  
  
Justification: the minimum cycler must contain all edges in the S described  
above, because, if any x in S is not included, adding x to the spanning tree  
T would create a cycle where no edge is in the cycler, which violates the cycler's definition. The cycler is minimum since it's complement, T, is  
maximum. The runtime is simply the runtime of Prim's or Kruskal's.

2. Consider the following new operation for disjoint sets.

- PRINT(x): print every element in S\_x (the set containing x).

Explain how to **augment the tree data structure** for disjoint sets to

implement the PRINT operation. Your goal: achieve worst-case running

time O(|S\_x|) for operation PRINT(x), without affecting the running time

of the other operations.

Write down each operation's detailed pseudo-code carefully. You should first try to solve it by adding two additional attributes to each node, then, as a challenge, think: can you achieve everything by adding only one additional attribute?

answer:

"Easy" solution: make each node store two additional pointers -- .next  
   and .tail. Starting from the root of S\_x, the .next pointers define a  
   linked list that goes through every element in S\_x. The pointer  
   root.tail points to the last node in this linked list, to make UNION  
   easier to implement. Parent pointers are unaffected.  
  
          - MAKE-SET(x):  # Runtime unchanged.  
                return a new Node with  
                .elem <- x  
                .parent <- self  
                .rank <- 0  
                .next <- NIL  
                .tail <- self  
  
          - FIND-SET(x):  # Runtime unchanged.  
                # Unchanged!  
  
          - UNION(x,y):  # Runtime unchanged.  
                w <- FIND-SET(x)  
                z <- FIND-SET(y)  
                if w != z:  
                    if w.rank < z.rank:  
                        w.parent <- z  
                        # Append w's list to z's list.  
                        z.tail.next <- w  
                        z.tail <- w.tail  
                    else:  
                        z.parent <- w  
                        if z.rank = w.rank:  
                            w.rank <- w.rank + 1  
                        # Append z's list to w's list.  
                        w.tail.next <- z  
                        w.tail <- z.tail  
  
          - PRINT(x):  # Runtime = O(|S\_x|), as desired.  
                z <- FIND-SET(x)  
                while z != NIL:  
                    print(z.elem)  
                    z <- z.next  
  
    Using only one additional pointer? Try a circular linked list where each  
    node only keeps a next. To determine if the list is traversed, use the  
    parent pointer to check if it is back at the root.