EC330 HW7 Solution

Emre Ateş

November 28, 2016

1. (a) This algorithm traverses the connected components by adding every node to a list, and removing the traversed nodes from that list. While traversing the vertices, it also counts the vertices and edges.

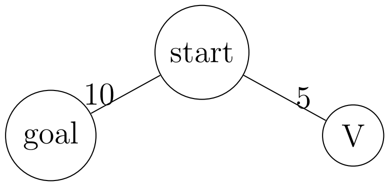
```
Function Main($G$ = $(E, V)$):
    List L = V
    List result
    while $L$ is not empty do
        List temp                                /* Start a connected component */
        $v$ = L.pop()
        temp.add($v$)
        edges = 0
        vertices = 0
        while temp is not empty do
            $t$ = temp.pop()                     /* We do a breadth-first search */
            vertices++
            for edge($t, u$) do
                edges++
                if $u \in L$ then
                    L.remove($u$)
                    temp.add($u$)
                end
            end
            result.add((vertices, edges/2))      /* Since we're counting every edge
             twice */
        end
    end
    return results       /* The number of connected features is the number of
     tuples in results */
```

Depth-first search can also be used to construct a successful algorithm.

(b) Since we don't need to find the optimal solution, we can begin constructing an independent set, until we reach a local maxima. To do so, we can start by coloring a node green; color every neighbor of the green node red; find a node neighboring a red node and color it green; color every neighbor of the green node red and so on, until we color every node. At the end, the green nodes would consist an independent set, because every green node only has red neighors, and every red node has at least one green neighbor. This way, if we add a red node to the set, it is guaranteed to be neighboring a node already in the set, causing our set to be not independent anymore.

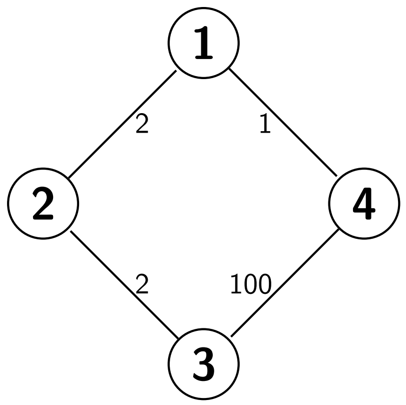
```
Function Main(G = (E,V)):
    List targets, neighbors        /* targets are green, neighbors are red */
    List nodes = V
    newtarget = NULL
    while nodes ≠ ∅ do
        for v ∈ neighbors do
            for u | (v,u) ∈ E ∧ u ∈ nodes do
                nodes.remove(u)
                targets.add(u)
                newtarget = u
                break                          /* break from both loops */
            end
        end
        if newtarget = NULL then
            /* Both for start, and for if we have more than one connected
               components                                                  */
            newtarget = nodes.pop()
            targets.add(newtarget)
        end
        for u | (newtarget,u) ∈ E ∧ u ∈ nodes do
            nodes.remove(u)
            neighbors.add(u)
        end
        newtarget = NULL
    end
    return targets
```

(c) No, it does not find the shortest path. It might even fail to find a path altogether:



In this case, the algorithm will go to node V and then terminate, because it won't go back to the start node.

Here's an example of the algorithm finding a path, but not the shortest one:

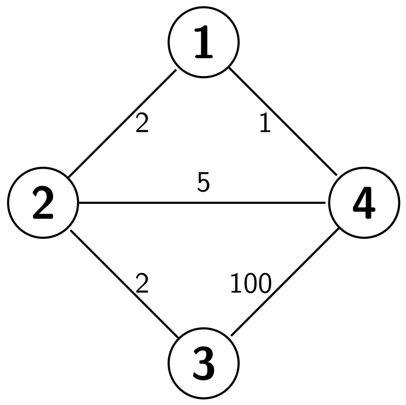


When looking for a path from 1 to 3, the algorithm would go to 4, because it's cheaper, and then the only path going to 4 would have a total weight of 101; however the shortest path is going through 2.

2. (a) The complexity of Prim's algorithm can decrease. If we implement the queue as an array [1..W] of linked lists. We will add every edge to the corresponding slot in the array according to its weight. Extract-min is $O(1)$, and decrease key is also $O(1)$. This way, we only need to go over the edge list once, resulting in a $O(E)$ runtime.

   (b) The complexity of Kruskal's algorithm is $O(E + V \log V)$, since we can sort the edges in linear time using counting sort or radix sort. After the sorting, we only need to go over the list once, and add the edge to our list if it connects a previously disconnected component.

3. (a) Since this algorithm doesn't look at the edge weights at all, it can't find the minimum spanning tree.



   In this graph, the algorithm might randomly add the edge with weight 100 to the tree, resulting in a tree that is not minimal.

   (b) This algorithm is essentially the inverse of Kruskal's algorithm. It would find the edges with the higher weights to be removed. It would find the minimum spanning tree.