ADS 2021: Week 4 Solutions - DRAFT

Solutions for week 4 of Algorithms and Data Structures.

4.1.1 - Green The maximum number of edges in a graph with V vertices and no parallel edges is $(V \cdot (V-1)/2) + V$. Since we do not have parallel edges, each vertex can connect to $V-1$ other vertices plus itself. In an undirected graph vertex v connected to vertex w is the same as vertex w connected to vertex v, so we divide the result by 2 for all edges that are not self-loops, and then add the V self-loops.

Example (self-loops not shown):

```
o - o
| X |
o - o
```

The minimum number of edges in a graph with V vertices, none of which are isolated (have degree 0) is $\lceil V/2 \rceil$.

Example:

```
o - o  o - o
```

4.2.1 - Green The maximum number of edges in a digraph with V vertices and no parallel edges is V^2. Since we do not have parallel edges, each vertex can connect to V other vertices, including itself.

Example (without self-loops):

```
o < = > o
^       ^
|  X X  |
v       v
o < = > o
```

The minimum number of edges in a digraph with V vertices, none of which are isolated (have degree 0) is $\lceil V/2 \rceil$.

Example:

```
o -> o -> o
```

4.1.28 - Green There are 2 non-isomorphic graphs with 2 vertices:

```
o o                 o-o
```

There are 4 non-isomorphic graphs with 3 vertices:

```
 o                   o                   o-o-o                o
o o                 o-o                                      / \
                                                             o-o
```

There are 11 non-isomorphic graphs with 4 vertices:

```
o o o o               o                 o-o                   o
                      |                 | |                  / \
o-o o o               o                 o-o                 o---o
                     / \                                     \ /
o-o-o o             o   o                 o                   o
                                          |
o-o-o-o               o                   o                    o
                     / \                 / \                  /|\
o-o o-o             o---o o             o---o                / o \
                                                            / / \ \
                                                            o------o
```

4.2.8 - Green There are 2 non-isomorphic DAGs with 2 vertices:

```
o o                 o->o
```

There are 6 non-isomorphic DAGs with 3 vertices:

```
 o                  o->o->o               o                  o
o o                                      ^ ^                / \
                     o                  /   \               v  v
 o                  / \                o     o              o->o
o->o                v  v
                    o  o
```

There are 31 non-isomorphic DAGs with 4 vertices:

```
o o o o             o->o o
                     \                  o->o<-o o             o
o->o o o              v                    o                  |
                      o                    |                  v
o->o->o o                                  v                  o
                    o<-o->o                o                 / \
o->o->o->o             |                  ^ ^               v   v
                       v                 /   \              o   o
o->o o->o              o                o     o
```

```
   o
   |
   v
   o
  ^ \
 /   v
o     o
```

```
   o
   |
   v
   o
  ^ ^
 /   \
o <-- o
```

```
  o
  |
  v
  o
 / \
v   v
o<--o
```

```
   o
   |
   v
   o
  ^ \
 /   v
o --> o
```

```
o    o
| /\ |
vv  vv
o    o
```

```
o -> o
| /\ |
vv  vv
o    o
```

```
o    o
| /\ |
vv  vv
o <- o
```

```
o -> o
| /\ |
vv  vv
o <- o
```

```
  o
 / \
v   v
o-->o o
```

```
  o
 ^ ^
/   \
o-->o o
```

```
o->o
|  |
v  v
o->o
```

```
o-->o
|\ /|
| X |
vv vv
o-->o
```

```
  o
  |
  v
  o
 / \
v   v
o-->o
```

```
  o
  ^
  |
  o
 / \
v   v
o-->o
```

```
  o
  |
  v
  o
 ^ ^
/   \
o-->o
```

```
  o
  ^
  |
  o
 ^ ^
/   \
o-->o
```

```
  o
 / \
v   v
o--->o
 \   /
  v v
   o
```

```
  o
 ^ ^
/   \
o--->o
 \   /
  v v
   o
```

```
   o
  /|\
  /v \
 / o \
/ / \ \
vv   v v
o----->o
```

```
    o
   ^|^
   /v \
  / o  \
 / ^ ^  \
/ /   \  \
o------->o
```

```
    o
   ^^^
   /| \
  / o  \
 / ^ ^  \
/ /   \  \
o------->o
```

4.1.12 - Yellow We consider undirected graphs. Let L_v and L_w be the BFS-layers of the v and w respectively. (i.e. the distance to the root). For the distance $d(v, w)$ of u and w, we can conclude $|L_v - L_w| \leq d(v, w) \leq L_v + L_w$. (The path via the root exists and leads to the upper bound, a shortest path from v to w leads to it being an upper bound on the difference between L_v and L_w. More precisely, we can observe that neighboring nodes can have a difference of level of at most one, the claim follows inductively.)

4.1.16 - Yellow To solve all four questions, we make an array of the eccentricity of each vertex in the graph. To fill this array, as well as finding the diameter, radius and center of the graph, we do the following for each vertex:

- Run a BFS from that vertex. Then check each distTo in the resulting BFS to find the highest distTo value. This value is the eccentricity for that vertex. To solve (a) by itself, you would do this step for that vertex only.
- Keep track of the diameter, radius and center as you calculate the eccentricities.
  - If the eccentricity of the current vertex is higher than the current diameter, update the diameter.
  - If the eccentricity of the current vertex is less than the current radius, update the radius and set the current vertex as the center. Note that there could be other center vertices.

Alternative efficient but incorrect solution Run BFS (from any node) and find the node furthest away. The eccentricity of that node is the answer.

c+d) if you have found the longest shortest path in b, then halfway along that path a node with radius of the graph must exists. If not, then you would not have the diameter, since a longer path could be constructed.

These things only work in undirected graphs, but are quite efficient compared to calculating all the eccentricities.

4.1.21 - Yellow (Traversals should be done 'dynamically' on blackboard instead of as a static image. Important part is to communicate that (a) BFS and DFS do visit the vertices in different orders, and (b) the "choreography" of which vertex to visit next is handled by the (implicit) recursion stack in DFS and the (explicit) queue in BFS.)

The colourings found by the two approaches are the same, up to renaming of colours in each connected component. Both algorithms run in time $O(n+m)$. (Each edge is visited exactly twice, once from each endpoint.)

Important difference: Both Python and Java have ridiculously tiny recursion stacks, so none of the two programming languages is able to handle a graph of 10^5 vertices with the DFS implementation(!), defeating the whole idea of having a fast algorithm. In fact, the course library Python implmementation of Bipartite crashes with a run time exception when run on C_{1000}, the cycle of 1000 vertices(!). Message for Python or Java: Either always implement graph traversals as BFS, or implement DFS non-recursively (just like BFS, but using a stack instead of a queue), or (hacky) set the recursion limit to a larger value.

4.1.32 - Red Assume the graph is given as adjacency lists. For each vertex, consider the neighbors, count how many single-edge-connections there are to each neighbor. If there are at least two, all edges are parallel edges. To achieve linear running time, reuse the array and reset it instead of creating a new one. Below is code for this $O(V+E)$ time solution:

```
private int countParallelEdges(Graph graph) {
        int parallelEdges = 0;
        int[] connections = new int[graph.vertices()];
        // conveniently initialized with zeros

        for(int vertex = 0; vertex < graph.vertices(); vertex++) {
            for(int neighbor : graph.adjacent(vertex)) {
              connections[neighbor] += 1
            }
            // Clean up the connections array for the next vertex
            for(int neighbor : graph.adjacent(vertex)) {
              if (connections[neighbor] > 1) {
                parallelEdges += connectios[neighbor]
                }
              connections[neighbor] = 0;
              // importantly, nothing is added on a second visit
            }
        }

        // Divide by two as a parallell edge from a to b will be counted
        // once from a to b and once from b to a
        return parallelEdges / 2;
    }
```

4.1.36 - Red Start by checking that the graph is connected, by running either DFS or BFS and ensuring that every vertex is reachable. Then notice that the running time requirement is $O(E \cdot (V+E))$, suggesting we can run either BFS or DFS for each edge in the graph. For edge e to be a bridge, its removal from the graph must make one or more vertices unreachable. So for each edge, remove the edge from the graph (doable in constant time in adjacency list format, but even linear cost would not be problematic) and run either DFS or BFS. If the new graph without edge e is still connected, e is not a bridge and we can add e back to the graph and check the next edge. If we get through all edges without finding a bridge, the graph is edge connected.