CSCI 2300 — Algo
Homework 4
Hayden Fuller

- **Q1** An undirected graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. Design a linear time, i.e., $O(|V| + |E|)$, time algorithm to check if a graph is bipartite or not using a list of lists representation:

```
def dfs(g):
    v=[]     #visited
    c=[None for i in range(len(g))]     #"color", starts None, painted 1 or 0
    for i in range(len(g)):     #visit everything not yet visited
        if i not in v:
            v,c,b = dfss(g,v,c,i)
            if not b:     #boolean return value
                return False
    return True


def dfss(g,v,c,i):
    v.append(i)
    flag0=False
    flag1=False
    for j in g[i]: #visit all connected nodes
        if j in v: #only those that have been visited (and therefore colored)
            if c[j]==0: flag0=True #track if any neighbors are colored 0 or 1
            elif c[j]==1: flag1=True
    if flag0 and flag1: return v,c,False #is touching nodes of both colors
    elif flag0: c[i]=1 #if touching a 0, color this 1
    else: c[i]=0 #if touching a 1 or nothing, color this 0
    for j in g[i]: #visit all connected nodes again
        if j not in v: #this time the other ones that haven't been visited
            v,c,b = dfss(g,v,c,j) #continue dfss on them
            if not b: return v,c,False
                #if any of them return false, return false all the way back
    return v,c,True #if none are false, all branches are bipartite, return true
```

- **Q2** Answer the following questions:

  (a) Prove that a non-empty DAG must have at least one source.
  Assume that the DAG does not have a source.
  This means every vertex must have at least one incoming edge
  The only way to remove a source is to make another vertex point to it
  The only two ways this can be done are to add a new source pointing to it or to have an existing vertex point to it
  we can not point an existing vertex to the source since the source points indirectly to

the existing vertex, which would create a directed cycle
assuning we don't allow an infinite chain, we also can't just keep adding sources to infinity
Therefore, a non-empty DAG must have at least one source.

(b) What is the time complexity of finding a source in a directed graph or to determine such a source does not exist if the graph is represented by its adjacency matrix? Describe the algorithm.
$O(V^2)$. The matrix is of size V x V, and you must find a column(or row, depending on construction of the matrix) where all values are zero/false, meaning that a vertex has nothing pointing to it, making it a source. While $O(V^2)$ is quite slow, but the average is faster since each column search ends as soon as it sees a 1 and the algorithm ends as soon as it finds a collumn of all 0's.

(c) What is the time complexity of finding a source in a directed graph or to determine such a source does not exist if the graph is represented by its adjacency list? Describe the algorithm.
$O(V+E)$ This results in V lists, but with the size of those lists being e_i where $\sum e_i = E$, we only need to go through E items. You go through V lists and look at each item in each one, keeping track of what's been seen. This gives you a list of everything that's *not* a source, meaning you must still compare to a list of vertecis to find which ones are sources (likely by starting with that list and subtracting as you go), giving you a +V to the O notation.

- **Q3** Describe a linear time algorithm to compute the neighbor degree for each vertex in an undirected graph. The neighbor degree of a node x is defined as the sum of the degree of all of its neighbors.
$O(V+E)$

```
output=[0 for i in range(len(g))]
for i in range(len(g)):
  for j in g[i]:
    output[j]+=len(g[i]
```

- **Q4** Consider a directed graph that has a weight w(v) on each vertex v. Define the reachability weight of vertex v as follows:
$r(v) = \max\{w(u) | u \text{ is reachable from } v\}$
That is, the reachability weight of v is the largest weight that can be reached from v. Answer the following questions:

(a) Assume the graph is a DAG. Describe a linear time algorithm to compute the reachability weight for all vertices.
$O(V+E)$
For each vertiсy, if it's not yet been visited, call depth first search
Each call will calculate it's $r(v)$ by first setting $r(v) = w(v)$, and then continuing with it's dfs.
If a vertex i it points to has already been visited, it will set $r(v) = max(r(v), r(i))$, where r(i) is grabbed from the vector used for storage/output.
If a vertex i it points to has not been visited, it will recursively call dfs(i) and use that value as $r(i)$ to set $r(v) = max(r(v), r(i))$

Finally, it will save it's $r(v)$ in a vector for storage/output and return it's $r(v)$ value to the vertex above it.

(b) Assume that the graph is a general directed graph (with possible cycles). Describe a linear time algorithm to find the reachability weight for all vertices.
First, run the same strongly connected components algorithm from the lab, which is linear time.
Any part of a strongly connected component can access any other part of that strongly connected component, meaning they'll share the same $r(v)$
Find the local $r(v)$ of these components as the max $w(v)$ of the local components.
Then run the same DFS algorithm as above, but now on strongly connected components rather than individual vertecies.
Each verticy is assigned $r(v) = r(c)$, their reachability is the same as that of their strongly connected component.