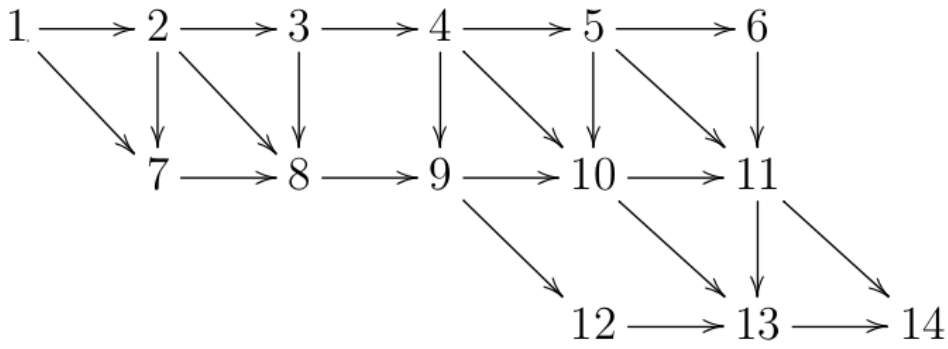
CSCI 3104 Spring 2018 Problem Set 8
Mohamed Al-Rasbi (07/25)

Problem 1

(10 pts) Ginerva Weasley is playing with the network given below. Help her calculate the number of paths from node 1 to node 14.

Hint: assume a "path" must have at least one edge in it to be well defined, and use dynamic programming to fill in a table that counts number of paths from each node j to 14, starting from 14 down to 1.



We know to things about this network:
1. Each node has at least one edge.
2. Each node is connected to all its surrounding nodes. In other words, there is always going to be a path between node i and node j.
This is a pseudo-code for my algorithm:

```
'''
Initialize an object that is an adjacent list of the nodes in the graph
Each element in the adjacent list is another object,
that consists of a node, and a list of its edges.
For example: the last element(let's say it's node 14), is an object
             nodeName = 14, adjList = [11, 13]
'''
Graph g
```

```
startNode = 1 #node 1
count = 0 #count number of paths
for node in g:
    for adj in node.adjList:
        if adj == startNode or adj == 1:
            return count
    count += len(node.adjList)

return -1 #startNode not found in graph
```

We can see that these types of networks has $\Theta(n * e)$ paths, for a graph that has n nodes starting from i=1 and j=14, and an average of e edges for each node

14: [11 , 13] count = 2

11: [6, 5, 10] count = 4
6− > 5− > 4− > 3− > 2− > 1
5− > 4− > 3− > 2− > 1
10: [5, 4, 9] count = 6
5− > 4− > 3− > 2− > 1
4− > 3− > 2− > 1
9: [4, 8] count = 7
4− > 3− > 2− > 1
8: [3, 2, 7] count = 9
3− > 2− > 1 2− > 1 7: [2, 1] count = 10
2− > 1
1
13: [11, 10, 12] count = 12

11: [6, 5, 10] count = 14
from above, eventually, count = 20

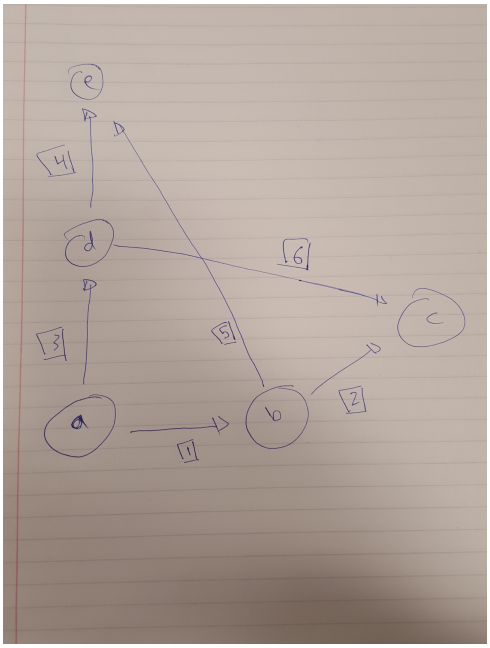
10: [5, 4, 9]
from above eventually, count = 26

12− > 9
9: [4, 8]
from above, eventually, count = 30

Therefore, the number of paths from 1 to 14 is 30.

Problem 2

(10 pts) Ginny Weasley needs your help with her wizardly homework. Shes trying to come up with an example of a directed graph $G = (V, E)$, a start vertex $v \in V$ and a set of tree edges $E_T \subseteq E$ such that for each vertex $v \in V$, the unique path in the graph (V, E_T) from s to v is a shortest path in G, yet the set of edges E_T cannot be produced by running a depth-first search on G, no matter how the vertices are ordered in each adjacency list. Include an explanation of why your example satisfies the requirements.



If you see the graph above, 1 and 2, 3 and 4 are the shortest paths in E_T.
What will happen when we apply depth-first search on the graph is as follows:
1. Edges 1:(a, b) and 3:(a, d) are added to the stack.
2. If 3:(a, d) is added first, then the edge 1:(a, b) will be traversed first. Hence, 2:(b, c) and 5:(b, e) will be added next. Therefore, the path 1, 5 is not the shortest from a to e. The path 3, 4 is the shortest.
3. If 1:(a, b) is added first, then the edge 4:(a, d) will be traversed first. Hence, 4:(d, e) and 6:(d, c) will be added next. Therefore, the path 3, 6 is not the shortest from a to c. The path 1, 2 is the shortest.

Hence, this graph satisfies the requirements.

Problem 3

(15 pts) Prof. Dumbledore needs your help to compute the in- and out-degrees of all vertices in a directed multigraph G. However, he is not sure how to represent the graph so that the calculation is most efficient. For each of the three possible representations, express your answers in asymptotic notation (the only notation Dumbledore understands), in terms of V and E, and justify your claim.

(a) *An **adjacency matrix** representation. Assume the size of the matrix is known.*

The adjacency matrix is of the size VxV. The rows are out degrees, and columns are in degrees.
So, basically, the sum of number of edges in a row is out degree, and the sum of the number of edges in a column is in degree.
Therefore, this can be done in $\Theta(V^2)$.

(b) *An **edge list** representation. Assume vertices have arbitrary labels.*

The problem here that we do not know V, the number of vertices. My algorithm is as follows:
1. We create a dictionary (in python) for in degrees and another one for out degrees. The key is the value of the node, the value is the degree.
2. We iterate through the list. Assume it looks like this [[out1, in1], [out2, in2], ...]. So the edge list is a two dimensional list. We go through each element and increment the values in the in and out dictionaries.
For example:
edge list = [[x, y], [x, z]]
dict out = x: 2, y: 0, z:0
dict in = x: 0, y: 1, z: 1

The running time depends on the search algorithm that is used in the dictionary to get a particular element. In python, to get an element from the dictionary it takes a constant time. However, I will assume a search algorithm is used and it takes about $\Theta(logV)$.
Therefore, the running time for this algorithm is $\Theta(ElogV)$.

(c) *An **adjacency list** representation. Assume the vectors length is known.*

Since we know the vector's length, that means we have V. This is the algorithm that can find the in and out degrees for adjacency list:
1. Initialize two arrays, in and out. Each element in the adjacency list correspond to its degree in "in" and "out" arrays. In other words, the first vertex in the vector, index 0, has the degree in[0] and out[0], and so on.

2. Go through the vector, and then go through the edges of each vertex.
3. Add the values of in and out degree of each vertex in the arrays.

Example shows my algorithm:
V = [1, 2, 3]
adjList = [1:[2, 3], 2:[1], 3:[1, 2]]
Step one:
in = [0, 0, 0], out = [0, 0, 0]
Step two and three:
node = 1
edges = [2, 3]
out = [2, 0, 0]
in = [0, 1, 1]

node = 2
edges = [1]
out = [2, 1, 0]
in = [1, 1, 1]

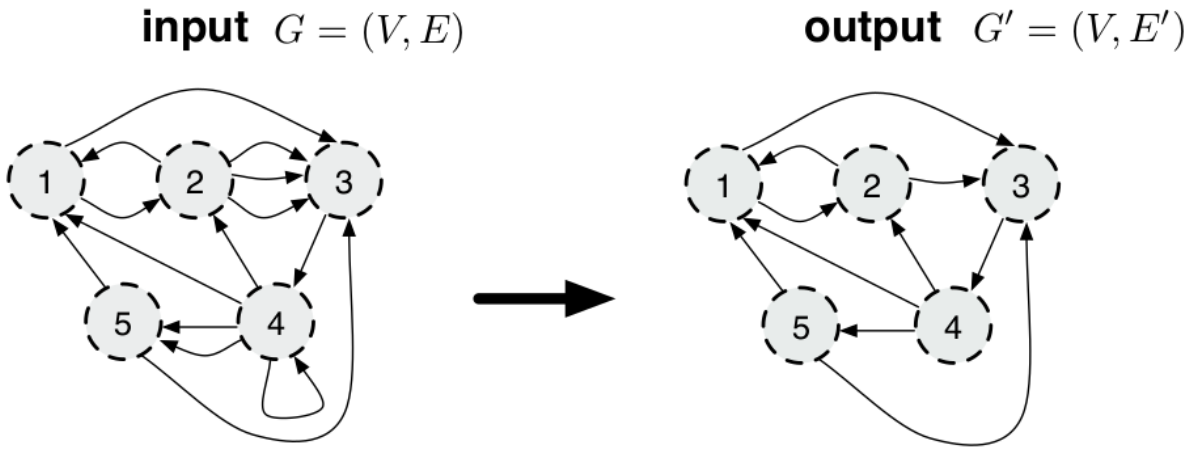
node = 3
edges = [1, 2]
out = [2, 1, 2]
in = [2, 2, 1]

Therefore, we can see the running time depends on V (number of vertices) and E (number of edges), so the running time is $O(V + E)$.

Problem 4

(30 pts) Deep in the heart of the Hogwarts School of Witchcraft and Wizardry, there lies a magical grey parrot that demands that any challenger efficiently convert directed multigraphs into directed simple graphs. If the wizard can correctly solve a series of arbitrary instances of this problem, the parrot will unlock a secret passageway.

Let $G = (E, V)$ denote a directed multigraph. A directed simple graph is a $G' = (V, E')$, such that E' is derived from the edges in E so that (i) every directed multiedge, e.g., $(u, v), (u, v)$ or even simply (u, v), has been replaced by a single directed edge (u, v) and (ii) all self-loops (u, u) have been removed.



1	3	2			
2	1	3	3	3	
3	4				
4	4	5	2	5	1
5	3	1			

1	2	3	
2	1	3	
3	4		
4	1	2	5
5	1	3	

An example of transforming $G \to G'$

Describe and analyze an algorithm (explain how it works, give pseudocode if necessary, derive its running time and space usage, and prove its correctness) that takes $O(V + E)$ time and space to convert G into G', and thereby will solve any of the Sphinxs questions. Assume both G and G' are stored as adjacency lists.

Hermiones hints: Dont assume adjacencies `Adj[u]` *are ordered in any particular way, and remember that you can add edges to the list and then remove ones you don't need.*

I will assume that the elements of the adjacent list are objects, each object consists of node name and a list of the adjacent edges to that node. The list of edges is merely a list of strings (node names).
This is my algorithm:

```
def convertGraph(adjList):
    newAdjList = []
    for node in adjList:
        object newObj #create object that have name and list methods
        newObj.name = node.name
        for adj in node.list:
            if adj != node.name and adj not in newObj.list:
                #append adj to newObj, but we need to keep them sorted
                for i in len(newObj.list):
                    if newObj.list[i] > adj:
                        newObj.list.insert(adj, i) #insert adj at index i and shift
        newAdjList.append(newObj)

    return newAdjList
```

Explanation:
We iterate through the adjacency list, and iterate through the list of edges for each vertex. Add the edges that are not repeated and not self-loops edges to the new list of edges that is stored in the new object. Then add the new object (vertex) to the new adjacency list. And iterate again through all vertices.

Running time:
As you can see from the algorithm, we have three loops.
The first for loop iterates through the adjacency list: iterates V times.
The second for loop iterates through the list of edges of each vertex: that is E times.
The third for loop iterates through the list of edges of each vertex of the new adjacency list, and that is for inserting the name of edges in an increasing order: that is at most E times.
Therefore, the running time of this algorithm with sorting is $O(V + E^2)$.
However, the running time without sorting is $O(V+E)$, which is, I think, what is required.

Space usage:
The space of the original graph if $V + E$ and the space for the new graph is $O(V + E)$, since we are deleting edges. Therefore, this is the space required.

Proof of correctness: using loop invariant
Initialization: Before the loop, we have an empty graph, and that is true since we did not start the iterations and did not change anything on the original graph.
Maintenance: We iterate through each list of edges of each vertex. And then we add the edges to the new list of edges in the new object that is stored in the new adjacency list.

By contradiction, let's assume there is a self-loop edge or a repeated edge and it is going to be stored to the new list. When we go through the list of edges, we compare if the edge does not equal the vertex's name (check if not self-loop) and does not equal another edges in the new list of edges (no repetition). If that is true, then add the edge. And in this case it is wrong. Therefore, if an edge is repeated or is a self-loop, it will not be added to the new list of edges. Hence, loop invariant holds true.
`Termination:` At the end of the function, we will have the same graph, different graph or an empty graph. If we have the same graph, that means the graph was already simple graph and the condition (if adj != node.name and adj not in newObj.list) is true in all iterations. If the graph is different, then some edges were not added. If the graph is empty, then that means the original graph is empty and we did not go through the loop.

Therefore, using loop invariant, we proved that the algorithm is correct.

Problem 5

(15 pts extra credit) Professor McGonagall has provided the young wizard Ron with three magical batteries whose sizes are 42, 27, and 16 morts, respectively. (A mort is a unit of wizard energy.) The 27-mort and 16-mort batteries are fully charged (containing 27 and 16 morts of energy, respectively), while the 42-mort battery is empty, with 0 morts. McGonagall says that Ron is only allowed to use, repeatedly, if necessary, the ***mort transfer spell*** *when working with these batteries. This spell transfers all the morts in one battery to another battery, and it halts the transfer either when the source battery has no morts remaining or when the destination battery is fully charged (whichever comes first).*

McGonagall challenges Ron to determine whether there exists a sequence of mort-transfer spells that leaves exactly 12 morts either in the 27-mort or in the 16-mort battery.

(a) *Ron knows this is actually a graph problem. Give a precise definition of how to model this problem as a graph, and state the specific question about this graph that must be answered.*

We can model this directed graph so that each node is a battery, and the weights are the morts(42, 27, 16). The question that should be answered is to find a path that results 12.

(b) *What algorithm should Ron apply to solve the graph problem?*

The A heuristic algorithm.

(c) *Apply that algorithm to McGonagalls question. Report and justify your answer.*