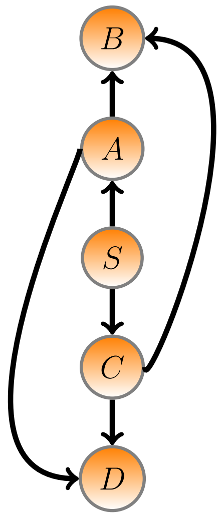
Adam Ten Hoeve (11/12) adam.tenhoeve@colorado.edu/

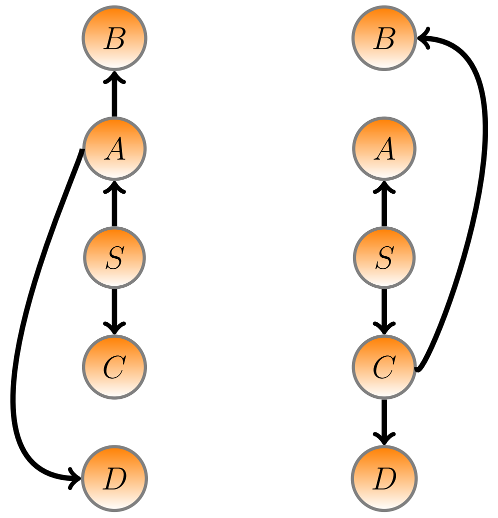
Problem 1 (10 points)

Hermione needs your help with her wizardly homework. She's trying to come up with an example of a directed graph $G = (V, E)$, a source vertex $s \in V$ and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique path in the graph (V, E) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running a breadth-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.

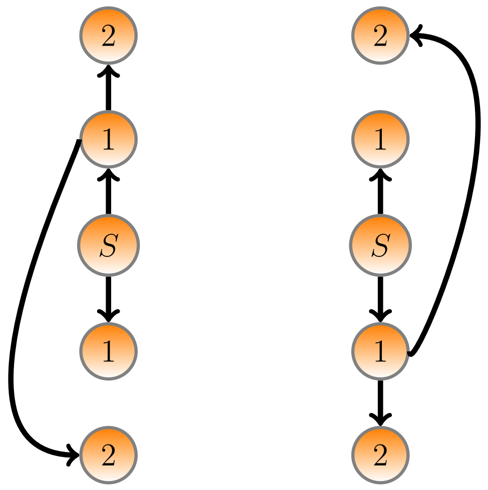
A directed graph G with source vertex S that could be used for this problem:



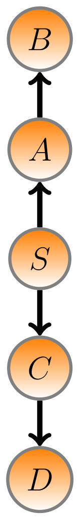
If we perform a breadth first search on G, starting at S, there are two sets of edges that can be created.

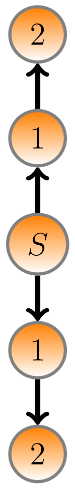


Through proof by exhaustion, we can see that there are four possible combinations of popped edges using BFS:
$(S, A), (S, C), (A, B), (A, D)$
(S, A), (S, C), (A, D), (A, B)
(S, C), (S, A), (A, B), (A, D)
(S, C), (S, A), (A, D), (A, B)
$These four combinations create the two subgraphs shown above. We can see that in both subgraphs, the distance t$



However, there is a subgraph E_π of G that has these same disances:





As proven through exhaustion above, it is not possible for BFS to find E_π. Therefor E_π is a tree with $s, v \in V$ where the path s to v is a shortest path in G and can not be found by running a breadth-first search on G.

Problem 2 (15 points)

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 edge list representation. Assume vertices have arbitrary labels.*

(b) *An adjacency list representation. Assume the vector's length is known.*

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

(a) $\mathcal{O}(E)$. A edge list representation is this order because an edge list is a list of tuples representing the edges, where the first element is the preceding vertex and the second element is the receiving vertex. To find the out-degree, we would traverse the list and count all of the times the first element is that vertex and to find the in-degree, we would do the same thing but count the second element. For both of these, we would traverse the entire list which can only have a max length of E because it has one element for each edge in the graph, therefor it is $\mathcal{O}(E)$.

(b) $\mathcal{O}(V + E)$. An adjacency list is a list of vertices, where at each position is a linked list of all edges going out from that vertex. To find the out-degree, we would count how many nodes in the linked list there is for each vertex in the adjacency list. To find the in-degree, we need to iterate through the adjacency list and the linked lists at each vertex to find every occurrence of the desired vertex. To iterate through the adjacency list would take V time because its length is determined by the number of vertices, and the number of nodes in all of the linked lists with always total E because each node represents an edge from that vertex to another node. The total number of edges is E, not per vertex, so the total run time is $\mathcal{O}(V + E)$.

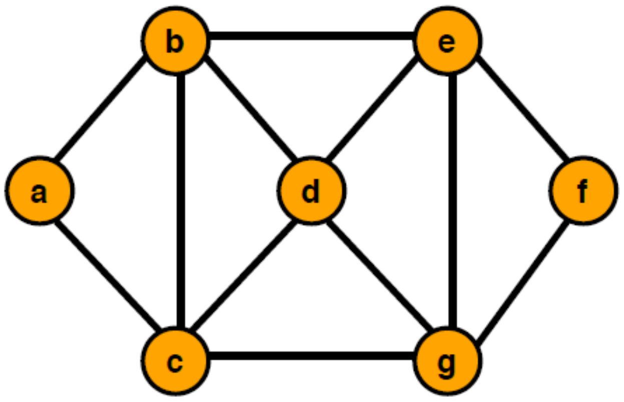
(c) $\mathcal{O}(V^2)$. To find the out-degree of a vertex, we would iterate through a row on the matrix and sum the number of positive entries. To find the in-degree, we would do the same iteration along a row but add all of the negative entries. To find the in and out-degrees for all of the vertices, we must iterate through every element of every row of the matrix. The matrix has width and height V each row/column corresponds to one vertex. Therefor, to traverse the entire matrix takes $\mathcal{O}(V^2)$ time.

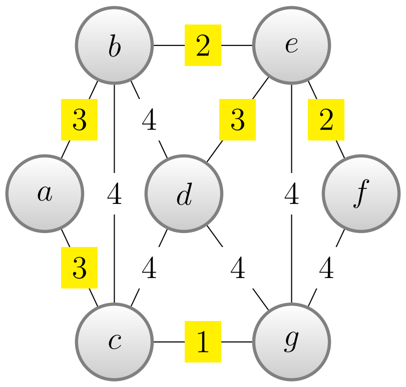
Problem 3 (25 points)

Professor Snape gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees and single-source shortest path trees:

- *The MST is distinct from any of the seven SSSP trees.*
- *The order in which Jarnik/Prim's algorithm adds the safe edges is different from the order in which Kruskal's algorithm adds them.*

Justify your solution by (i) giving the edges weights, (ii) showing the corresponding MST and all the SSSP trees, and (iii) giving the order in which edges are added by each of the three algorithms.

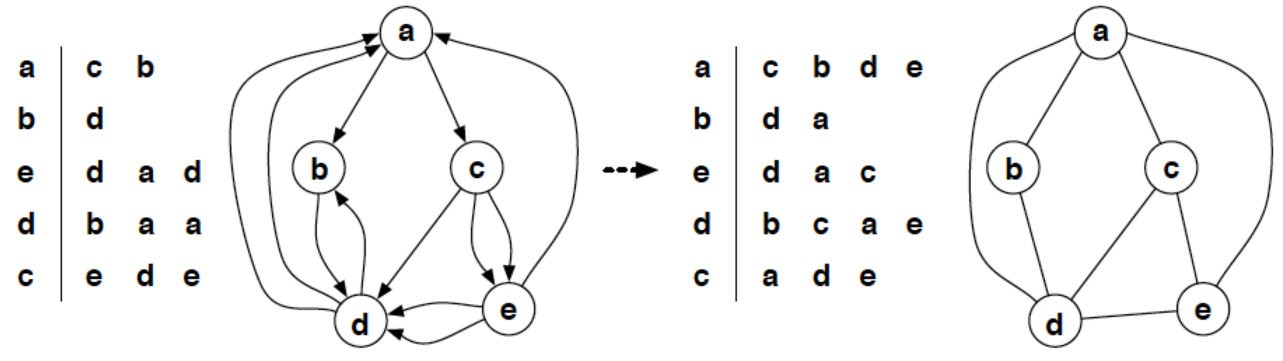




The graph above shows the solution, where the paths with the red numbers represent the MST. With these sets of weights, the MST will never be the same as a SSSP tree. The SSSP tree for vertices b, c, e, g will not be the MST because they will always take one of the vertical lines to reach the top/bottom, which is not the MST. Vertex d will have a SSSP where it goes directly from $d \rightarrow g$, which is not part of the MST. Vertex f will take the edge $f \rightarrow g$ instead of going all the way around, which the MST does. Vertex a will either go $a \rightarrow b \rightarrow d$ or $a \rightarrow c \rightarrow d$, and not $a \rightarrow b \rightarrow e \rightarrow d$, and not the same as the MST. All of the vertecies have some path that they will take that will lead to a different SSSP tree than the MST. Therefor the MST, which does have both branches, will never be the same as a SSSP.
The order in which the MST is created is different between Prim's and Kruskal's aglorithms. Kruskal's begins with the smallest edge weight on the graph, which would be the edge (c, g) with weight 1. The next edge added would be $(b.e)$ with weight 2, and so on. With Prim's algorithm, the only way to start with the (c, g) edge would be if it starts at vertex c or g. If that is true, then Prim's will add edge (c, g), then the next edge added will always be (a, c). This is a different order than Kruskals and the second condition is met.

Problem 4 (25 points extra credit)

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



a	c	b		
b	d			
e	d	a	d	
d	b	a	a	
c	e	d	e	

a	c	b	d	e
b	d	a		
e	d	a	c	
d	b	c	a	e
c	a	d	e	

An example of transforming $G \to G'$

Let $G = (E, V)$ denote a directed multigraph. An undirected simple graph is a $G' = (V, E')$, such that E' is derived from the edges in E so that (i) every directed multi-edge, e.g., $\{(u,v),(u,v)\}$ or even simply $\{(u,v)\}$, has been replaced by a single pair of directed edges $\{(u,v),(v,u)\}$ and (ii) all self-loops (u,u) have been removed.
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 Sphinx's questions. Assume both G and G' are stored as adjacency lists.
Hermione's hints: Don't 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.*

```
createUndirectedGraph(E,V):
initialize G' # list adj that will contain new edges
for i in V:
    for j in i.adj: # linked list of vertices connected to j
        if j is a new adj vertex and j != i: # checks for self loops:
            adj[i].append(j)
            adj[j].append(i)
return G'
```

The algorithm will loop over all of the vertices in G and compare the edges that have already been added. If the edge has been added (as in there was already an edge from or to both vertices) or if the start and end vertices are the same, meaning it is a self loop, then it is ignored. Otherwise the algorithm adds the edge to both start and end to each vertex because it is undirected. The algorithm loops over all of the verticies, taking V time, and compares every edge within the adjacency list, which is a total of E edges. Therefor, the total time complexity of $\mathcal{O}(V+E)$. The space complexity creates an adjacency list of size V and stores a total of E edges inside of it, so the space complexity is also $\mathcal{O}(E+V)$.