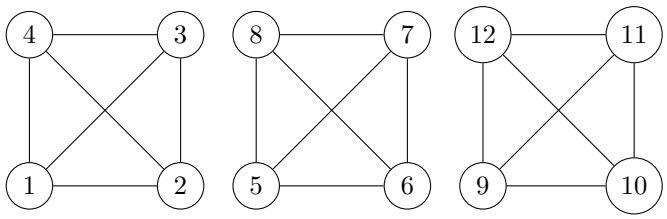
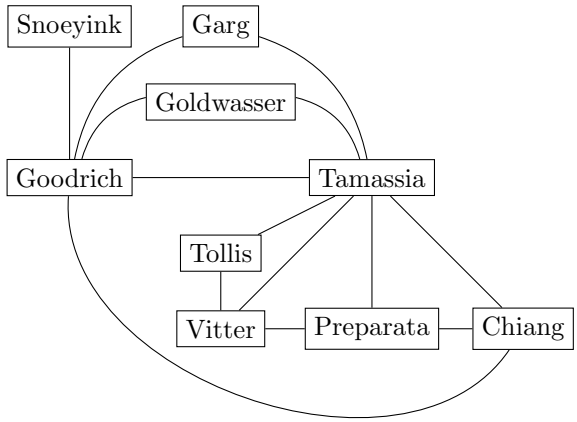
Homework 6 - Chapter 14 Graphs - Nick Palumbo
Due: Before November 23, 2015 11:59pm

R-14.1 Draw a simple undirected graph G that has 12 vertices, 18 edges, and 3 connected components.



R-14.3 Draw an adjacency matrix representation of the undirected graph shown in Figure 14.1.

Figure 14.1



			0	1	2	3	4	5	6	7	8
Snoeyink	→	0	0	0	0	1	0	0	0	0	0
Garg	→	1	0	0	0	1	1	0	0	0	0
Goldwasser	→	2	0	0	0	1	1	0	0	0	0
Goodrich	→	3	1	1	1	0	1	0	0	0	1
Tamassia	→	4	0	1	1	1	0	1	1	1	1
Tollis	→	5	0	0	0	0	1	0	1	0	0
Vitter	→	6	0	0	0	0	1	1	0	1	0
Preparata	→	7	0	0	0	0	1	0	1	0	1
Chiang	→	8	0	0	0	1	1	0	0	1	0

R-14.6 Suppose we represent a graph G having n vertices and m edges with the edge list structure. Why, in this case, does the insert vertex method run in O(1) time while the remove vertex method runs in $O(m)$ time?

insert_vertex(v): An edge list can insert a vertex in $O(1)$ because a new vertex instance will be added to the vertex object list. No other modifications need to be done to the edge list because when inserting a vertex there does not need to be an edge insert.

remove_vertex(v): An edge list will remove a vertex in $O(m)$ because in order to remove a vertex, first the vertex must be removed form the vertex object list in $O(1)$, but then the edges incident to the vertex that was removed must also be removed. This is where the m comes from. There are m edges and the edge object list must be traversed and each incedent edge to the removed vertex must be removed.

R-14.9 Can edge list E be omitted from the adjacency matrix representation while still achieving the time bounds given in Table 14.1? Why or why not?

The edges method depends on a secondary edge list so that the method is capable of running in $O(m)$. Without the secondary edge list, the adjacency matrix would have a running time of $O(n^2)$ because in order to find all the edges, the function must traverse the matrix that is of size $O(n^2)$.

R-14.11 Would you use the adjacency matrix structure or the adjacency list structure in each of the following cases? Justify your choice.

(a) **The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.**
An adjacency list structure has a space usage of $O(n + m)$ because the structure stores two object lists. One list is for the verex objects and the other is for the edge objects. On the other hand, the adjacency matrix has a space usage of $O(n^2)$ because of its n * n matrix size. Since there are only twice as many edges as vertecies, the more space efficient structure to use would be the adjaceny list structure.

verticies	$n = 10,000$
edges	$m = 20,000$

adjacency matrix space useage	$n * n = 100,000$
adjacency list space useage	$n + m = 30,000$

(b) **The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.**
Knowing that the adjacency matrix structure takes $O(n^2)$ space usage and the adjacency list structures takes $O(n + m)$ space usage, his scenario has a different choice. Sine the edges are significantly larger than the verticies the graph is much more dense and this makes the adjacency matrix structure more space efficient the the adjacency list structure.

verticies	$n = 10,000$
edges	$m = 20,000,000$

adjacency matrix space useage	$n * n = 100,000$
adjacency list space useage	$n + m = 20,010,000$

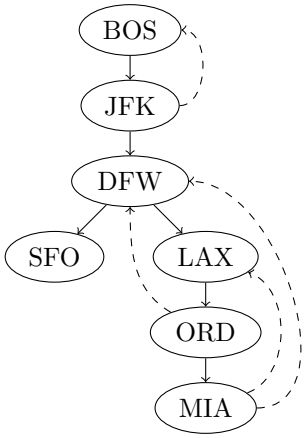
(c) **You need to answer the query get_edge(u,v) as fast as possible, no matter how much space you use.**
$get\_edge(u, v)$ is a function that will return the edge that connects vertex u to vertex v. The time complexity of this function for an adjacency matrix structure is $O(1)$ and

the time complexity of this function for an adjacency list structure is $O(min(d_u, d_v))$. Both vertecies u and v can have the same edges within their incident collections. Instead of going through both incident collections the minimum of the two is taken and that is the time complexity used when calling $get\_edge(u, v)$. However, $O(1)$ is constant time and the fastest time, therefore, since space usage does not matter, an adjacency matrix structure should be used.

R-14.12 Explain why the DFS traversal runs in $O(n^2)$ time on an n-vertex simple graph that is represented with the adjacency matrix structure.

A Depth-first search (DFS) traversal will run in $O(n^2)$ time when represented with an adjacency matrix structure because the DFS traversal must go to all possible vertecies. A adjacency matrix's space usage is $O(n^2)$ and the traversal from DFS must go to all possible incidents on each vertex. Therefore, te traversal will run in worst case $O(n^2)$ because of the way an adjacency matrix is stored in memory.

R-14.13 In order to verify that all of its nontree edges are back edges, redraw the graph from Figure 14.8b so that the DFS tree edges are drawn with solid lines and oriented downward, as in a standard portrayal of a tree, and with all nontree edges drawn using dashed lines.

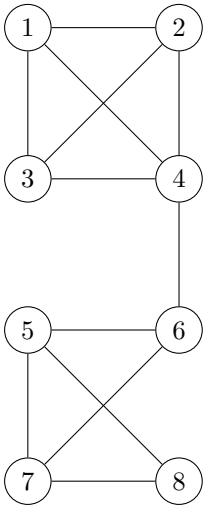


R-14.16 Let G be an undirected graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

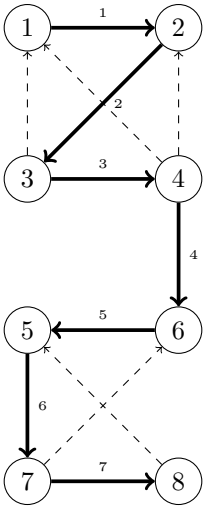
vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the table above.

(a) Draw G.



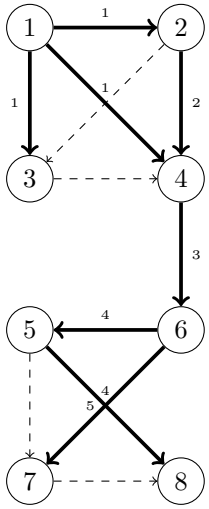
(b) Give the sequence of vertices of G visited using a DFS traversal starting at vertex 1.



DFS

1	2	3	4	6	5	7	8

(c) Give the sequence of vertices visited using a BFS traversal starting at vertex 1.

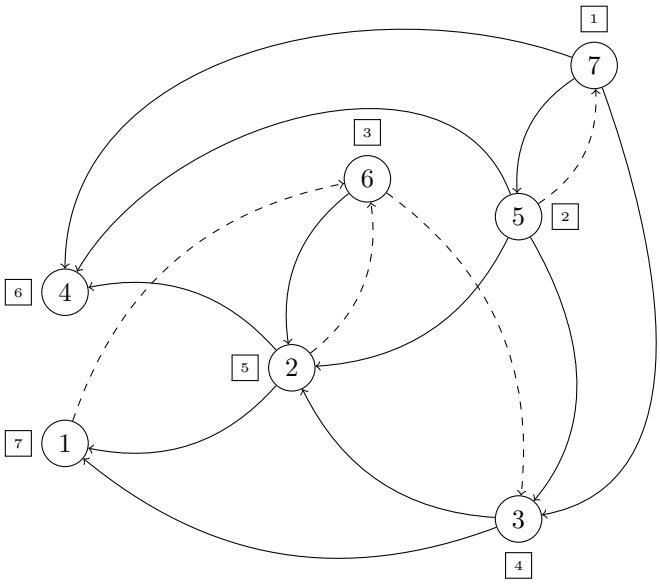


BFS

1	2	3	4	6	5	7	8

R-14.21 Compute a topological ordering for the directed graph drawn with solid edges in Figure 14.3d.

Figure 14.3d



Topological Ordering

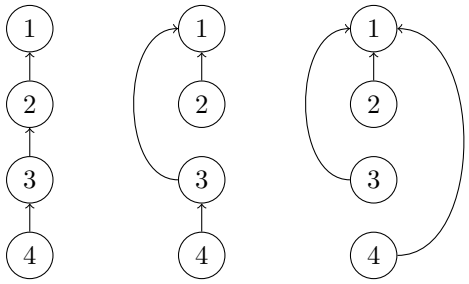
7	5	6	3	2	4	1

C-14.38 Give a Python implementation of the *remove_edge*(e) method for our adjacency map implementation of Section 14.2.5, making sure your implementation works for both directed and undirected graphs. Your method should run in $O(1)$ time.

For this implementation to work in $O(1)$ the method must be called with the vertex of the edge known. If the vertex is not known then the vertecies must be traversed to find out which vertecies use the edge. To remove the edge from a vertex pair, the vertices can change their value of edge to None so that no edge is connected. In the case of a directed graph the edges that are input are the incoming and outgoing edge from the vertecies. The *remove_edge*(e) must be called to remove all instances of the edge in a directed graph.

```
def remove_edge(self,e):
return None
```

C-14.73 Karen has a new way to do path compression in a tree-based union/find partition data structure starting at a position p. She puts all the positions that are on the path from p to the root in a set S. Then she scans through S and sets the parent pointer of each position in S to its parents parent pointer (recall that the parent pointer of the root points to itself). If this pass changed the value of any positions parent pointer, then she repeats this process, and goes on repeating this process until she makes a scan through S that does not change any positions parent value. Show that Karens algorithm is correct and analyze its running time for a path of length h.



Karen's new path compression is a successful function. However, the way of having to go through each vertex and set the parents up individually makes the function non-linear.