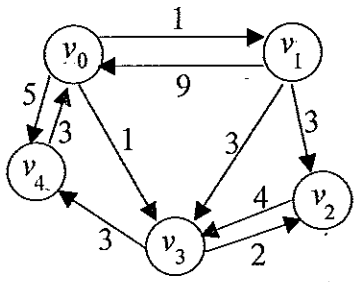
Data Structures (CS 1520) Lecture 25 Name:_____

1. Consider the following directed graph (digraph) $G = (V, E)$:



a) What is the set of vertices? $V = \{v_0, v_1, v_2, v_3, v_4\}$

b) An edge can be represented by a tuple (from vertex, to vertex [, weight]). What is the set of edges?

$E = \{(v_0, v_1, 1), (v_0, v_3, 1), (v_0, v_4, 5), (v_1, v_0, 9), (v_1, v_2, 3), (v_1, v_3, 3), \ldots$

c) A path is a sequence of vertices that are connected by edges. In the graph G above, list two different pathes from v_0 to v_3. v_0, v_3 or v_0, v_1, v_2, v_3

d) A cycle in a directed graph is a path that starts and ends at the same vertex. Find a cycle in the above graph.

v_0, v_3, v_4, v_0 or $v_0\ v_1\ v_0$

2. Like most data structures, a graph can be represented using an array, or as a linked list of nodes. The array representation is a two-dimensional array (called an *adjacency matrix*) whose elements contain information about the edges and the vertices corresponding to the indices. (Python could use a list-of-lists)

a) Complete the following adjacency matrix for the above graph. (Here a missing edge is represented by ∞.)

adjMatrix[#row][ ] (to vertex)

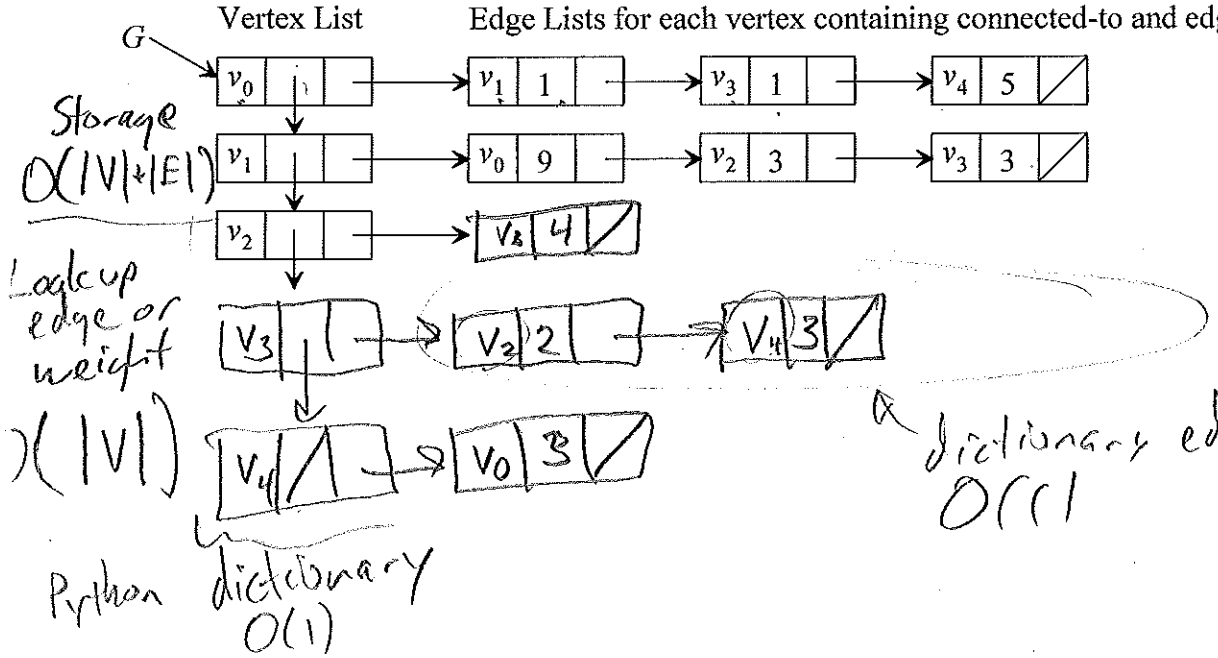
Storage $O(|V|^2)$

lookup edge existence or weight $O(1)$

(from vertex)

	v_0	v_1	v_2	v_3	v_4
v_0	0	1	∞	1	5
v_1	9	0	3	3	∞
v_2	∞	∞	0	4	∞
v_3	∞	∞	2	0	3
v_4	3	∞	∞	∞	0

b) The linked representation maintains a linked-list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Complete the adjacency list representation below:



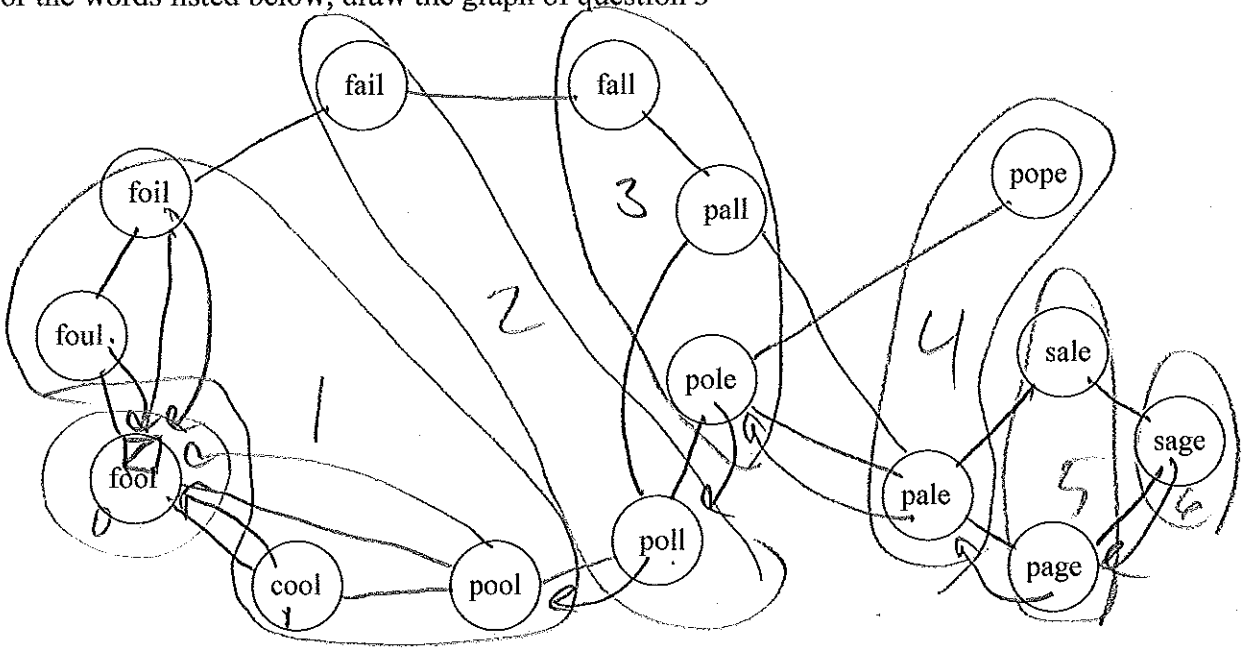
3. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider the *word-ladder puzzle* where you tranform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

We can use a graph algorithm to solve this problem by constructing a graph such that

- a word represents a vertex
- an edge represents?

- a word ladder transformation from one word to another represents?

4. For the words listed below, draw the graph of question 3



a) List a different transformation from FOOL to SAGE

fool → pool → poll → pole → pale → page → sage

b) If we wanted to find the shortest transformation from FOOL to SAGE, what does that represent in the graph?

shortest path from fool to sage

c) There are two general approaches for traversing a graph from some starting vertex *s*:

- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from s, before finding all vertices a distance 2 from s, etc.
- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a "dead end," we backtrack to the deepest vertex that allows us to try a different path.

Which of these traversals would be helpful for finding the **shortest** solution to the word-ladder puzzle?

BFS

Data Structures (CS 1520) Lecture 26 Name:_____

1. There are two general approaches for traversing a graph from some starting vertex *s*:

- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a "dead end," we backtrack to the deepest vertex that allows us to try a different path.
- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from s, before finding all vertices a distance 2 from s, etc.

What data structure would be helpful in each type of search? Why?

a) Breadth First Search (BFS):

fifo queue

myQ = empty queue
mark all vertices as white
set start vertex's distance 0 to.
gray, predictor = None
myQ.enqueue(start vertex)
while myQ is not empty do
currentVert = myQ.dequeue()
for nextVert its connected to do.
if nextVert is white then
update distance, color
predecessor to currentVert
enqueue nextVert
currentVert color black

b) Depth First Search (DFS):

stack instead of queue
recursion with run-time stack as the stack

2. On the next page is the textbook's edge, vertex, and graph implementations.

a) How does this graph implementation maintain its set of vertices?

VertList dictionary O(1) by key

b) How does this graph implementation maintain its set of edges?

Vertex object has connected to dictionary
O(1) to key in edge

3. Assuming a graph `g` containing the word-ladder graph from lecture 25, on the diagram trace the `bfs` algorithm by showing the value of each vertex's `color`, `predecessor`, and `distance` attributes?

```
""" File:  vertex.py """
class Vertex:
    def __init__(self, key, color = 'white',
                 dist = 0, pred = None):
        self.id = key
        self.connectedTo = {}
        self.color = color
        self.predecessor = pred
        self.distance = dist
        self.discovery = 0
        self.finish = 0

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: '
        + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]

    def getColor(self):
        return self.color

    def setColor(self, newColor):
        self.color = newColor

    def getPred(self):
        return self.predecessor

    def setPred(self, newPred):
        self.predecessor = newPred

    def getDiscovery(self):
        return self.discovery

    def setDiscovery(self, newDiscovery):
        self.discovery = newDiscovery

    def getFinish(self):
        return self.Finish

    def setFinish(self, newFinish):
        self.finish = newFinish

    def getDistance(self):
        return self.distance

    def setDistance(self, newDistance):
        self.distance = newDistance
```

white = nonvisited
gray = partially
black = completely done

```
""" File:  graph.py """
from vertex import Vertex

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor \
                        (self.vertList[t], cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

```
""" File:  graph_algorithms.py """

from graph import Graph
from vertex import Vertex
from linked_queue import LinkedQueue

def bfs(g,start):
  start.setDistance(0)
  start.setPred(None)
  vertQueue = LinkedQueue()
  vertQueue.enqueue(start)
  while (vertQueue.size() > 0):
    currentVert = vertQueue.dequeue()
    for nbr in currentVert.getConnections():
      if (nbr.getColor() == 'white'):
        nbr.setColor('gray')
        nbr.setDistance(currentVert.getDistance()+1)
        nbr.setPred(currentVert)
        vertQueue.enqueue(nbr)
    currentVert.setColor('black')
```