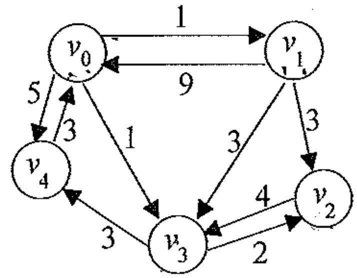
1. Consider the following directed graph (diagraph) $G = (V, E)$:



a) What is the set of vertices? $V =$ $\{V_0, V_1, \ldots, V_4\}$

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

$E =$ $\{(V_0, V_1, 1), (V_0, V_3, 1), (V_0, V_4, 5), \ldots \}$

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_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.

$V_0, V_1, V_2, V_3, V_4, V_0$ or V_2, V_3, V_2

2. Like most data structures, a graph can be represented using an array, or as a linked list of nodes.

a) The array representation is called an *adjacency matrix* which consists of a two-dimensional array (matrix) whose elements contain information about the edges and the vertices corresponding to the indices.

Edge value $O(1)$

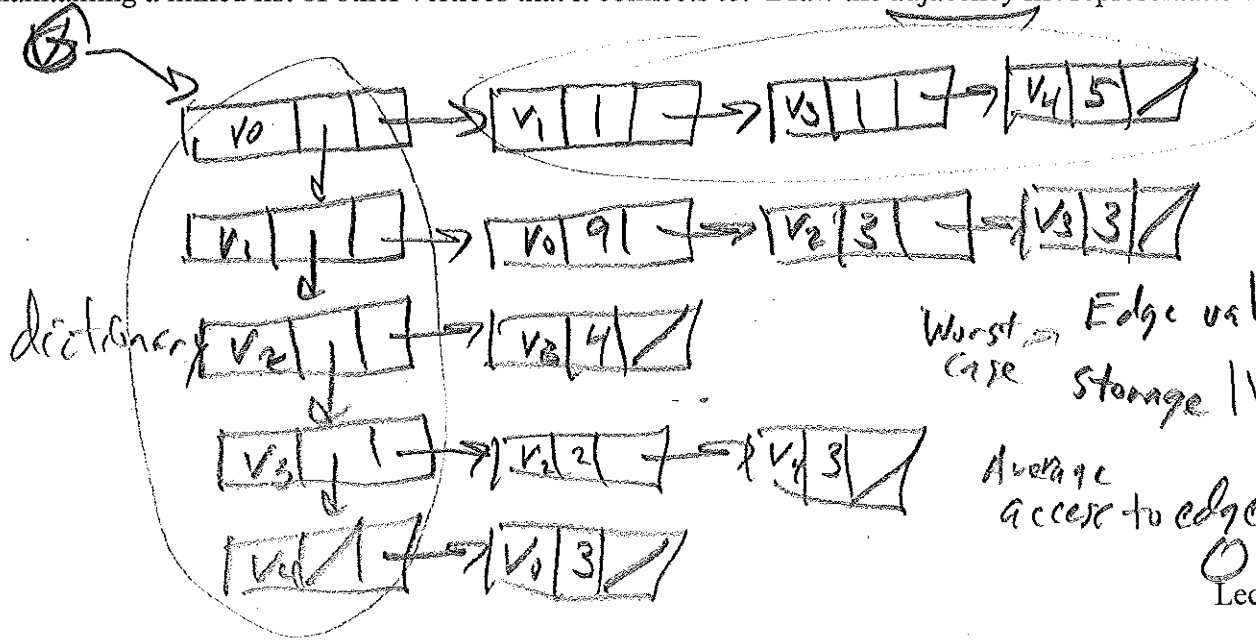
Complete the following adjacency matrix for the above graph.

storage: $|V|^2$

(to vertex)

(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

3. The linked representation maintains a array/Python list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Draw the adjacency list representation below:



Worst case Edge value $O(|V| \times ?)$

Storage $|V| + |E| \ll |V|^2$

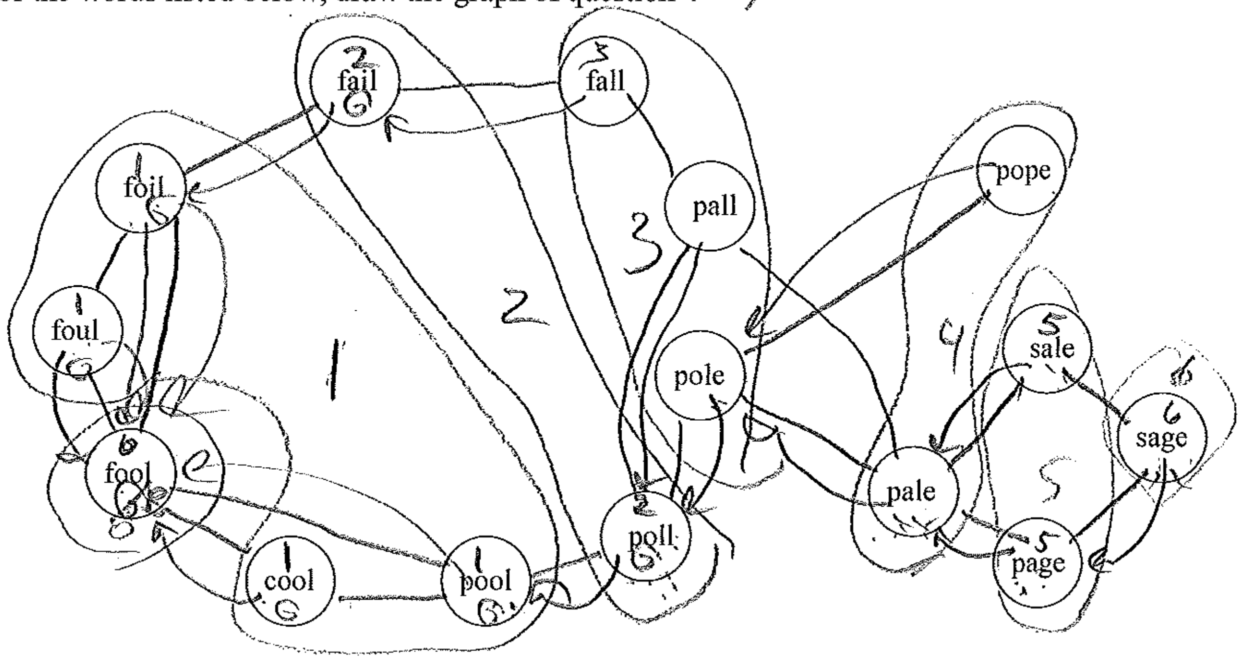
Average access to edge value $O(1)$

4. 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? connects words that differ by single letter
- a word ladder transformation from one word to another represents? path in the graph from starting word to ending word

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



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

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 - when finding SAGE it will be by the shortest path, but DFS will find SAGE possibly by a longer path.

Data Structures (CS 1520) Lecture 26 Name:____________

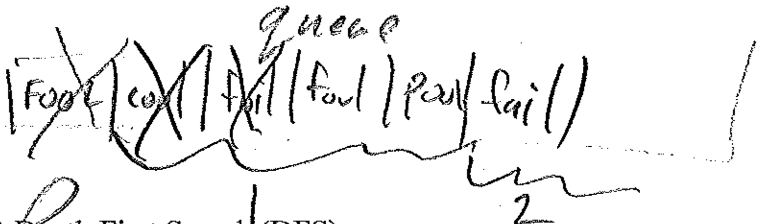
1. There are two general approaches for traversing a graph from some starting vertex *s*:

- 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.

set all vertices to "w"
myQ = empty queue
add FOOL to queue with distance of 0 and G.
while myQ is not empty:
currentVertex = myQ.dequeue()
for every neighbor of currentVertex
if neighbor is 'w' then
enqueue neighbor with distance of currentVertex's distance +1 and 'G'
Mark current Vertex as 'B'

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

a) Breadth First Search (BFS):



b) Depth First Search (DFS):

Stack

or run-time stack with recursion.

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?

A dictionary (vertList) with the vertex id/label as the key and the corresponding Vertex object as its value.

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

Each Vertex object maintains a dictionary (connectedTo) with Vertex objects that have edges to as keys and edge weights as values.

3. Assuming a graph g containing the word-ladder graph from lecture ~~26~~ 25, on the diagram trace the `bfs` algorithm by showing the value of each vertex's `color`, `predecessor`, and `distance` attributes?

(see graph on Lect. 25 page 2)

```python
""" 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
```

```python
""" 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())
```

```python
""" 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')
```