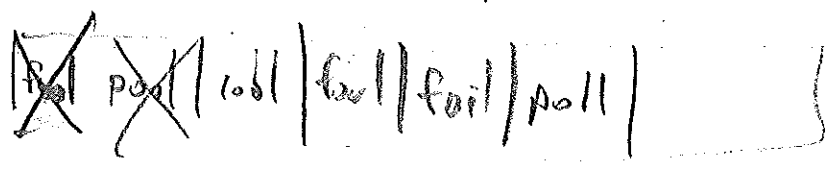


- There are two general approaches for traversing a graph from some starting vertex  $s$ :
  - Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a "dead end," we backtrack to the deepest vertex that allows us to try a different path.
  - Breadth First Search (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): queue

empty queue  
 enqueue start vertex with color gray  
 while queue is not empty  
 dequeue next vertex  
 for each neighboring vertex  
 if color is white then  
 enqueue neighbor with color gray  
 make next vertex as black



b) Depth First Search (DFS):

- use stack instead of queue to remember where to backtrack to

or use recursion and the run-time stack for DFS.

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?

As a dictionary, vertList, with (id, pointer to corresponding vertex object)

key: value

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

As a dictionary, connectedTo, with (pointer to vertex object, weight of edge to vertex)

key: weight

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

(done in Lab 11)

```

""" 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

```

```

""" 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())

```

{ "milk", }

```

""" 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')

```

4. Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The DFSGraph uses a time attribute to note when a vertex is first encountered (discovery attribute) in the depth-first search and when a vertex is backtracked through (finish attribute). Consider the graph for making pancakes where vertices are steps and edges represents the partial order among the steps.

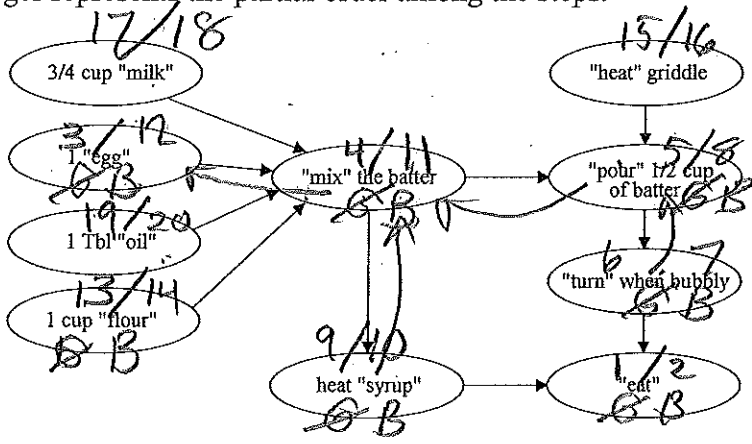
```

from graph import Graph
class DFSGraph(Graph):

    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self, startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
    
```

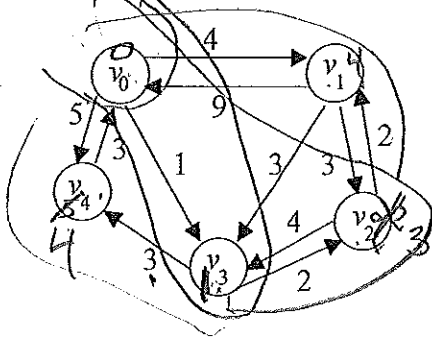


a) Assume (why is this a bad assumption???) that the for-loops always iterate through the vertices alphabetically (e.g., "eat", "egg", "flour", ...) by their id. Write on the above graph the discovery and finish attributes assigned to each vertex by executing the dfs method?

b) A *topological sort* algorithm can use the dfs discovery and finish attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour 1/2 cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. Outline the steps to perform a topological sort.

Perform steps in descending order of finish times

5. Consider the following directed graph (diagraph).



Dijkstra's Algorithm is a *greedy algorithm* that finds the shortest path from some vertex, say  $v_0$ , to all other vertices. A *greedy algorithm*, unlike divide-and-conquer and dynamic programming algorithms, DOES NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices). Dijkstra's algorithm builds a subgraph by repeatedly selecting the next closest vertex to  $v_0$  that is not already in the subgraph. Initially, only vertex  $v_0$  is in the subgraph with a distance of 0 from itself.

a) What would be the order of vertices added to the subgraph during Dijkstra's algorithm?

$v_0, v_3, v_2, v_1, v_4$

b) What *greedy criteria* did you use to select the next vertex to add to the subgraph?

Not in subgraph  
Vertex closest to  $v_0$  using only vertices currently in the subgraph

c) What data structure could be used to efficiently determine that selection?

min. heap to find the closest.

d) How might this data structure need to be modified?

A vertex in the subtree will have its distance reduced and need to be percolated up the heap. A bigger problem is finding the vertex in the heap.