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):

b) Depth First Search (DFS):

2. 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?
Lecture 26

File: graph_algorithms.py

```python
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')
```

File: vertex.py

```python
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 setColor(self, newColor):
        self.color = newColor
    def getPred(self):
        return self.predecessor
    def setPred(self, newPred):
        self.predecessor = newPred
```

File: graph.py

```python
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0
    def addVertex(self, key):
        if key not in self.vertList:
            self.numVertices = self.numVertices + 1
            newVertex = Vertex(key)
            self.vertList[key] = newVertex
            return newVertex
        else:
            return self.vertList[key]
    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())
```
3. Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The \texttt{DFSGraph} uses a \texttt{time} attribute to note when a vertex if first encountered (\texttt{discovery} attribute) in the depth-first search and when a vertex in backtracked through (\texttt{finish} attribute). Consider the graph for making pancakes where vertices are steps and edges represents the partial order among the steps.

![Graph diagram]

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

b) A \textit{topological sort} algorithm can use the \texttt{dfs} \texttt{discovery} and \texttt{finish} attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour ½ cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. \textbf{Outline the steps to perform a topological sort.}
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? \( v_0, \)

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

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

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