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

![Graph Diagram]

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?

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

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

$V_0, v_1, v_2, v_3$ and $V_0, v_1, v_3, v_2, v_5$

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.

Complete the following adjacency matrix for the above graph.

<table>
<thead>
<tr>
<th></th>
<th>$v_0$</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$v_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td>9</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_2$</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_3$</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$v_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

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:
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 transform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by

\[
\text{FOOL} \rightarrow \text{FOIL} \rightarrow \text{FAIL} \rightarrow \text{FALL} \rightarrow \text{PALL} \rightarrow \text{PALE} \rightarrow \text{SALE} \rightarrow \text{SAGE}.
\]

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

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

\[\text{FOOL} \rightarrow \text{POLL} \rightarrow \text{POLE} \rightarrow \text{PALE} \rightarrow \text{PAGE} \rightarrow \text{SAGE}\]

a) List a different transformation from FOOL to SAGE

\[\text{FOOL} \rightarrow \text{POOL} \rightarrow \text{POLL} \rightarrow \text{POLE} \rightarrow \text{PALE} \rightarrow \text{PAGE} \rightarrow \text{SAGE}\]

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

\text{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.
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):

```
Queue
```

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 (connects) 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, 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)
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')