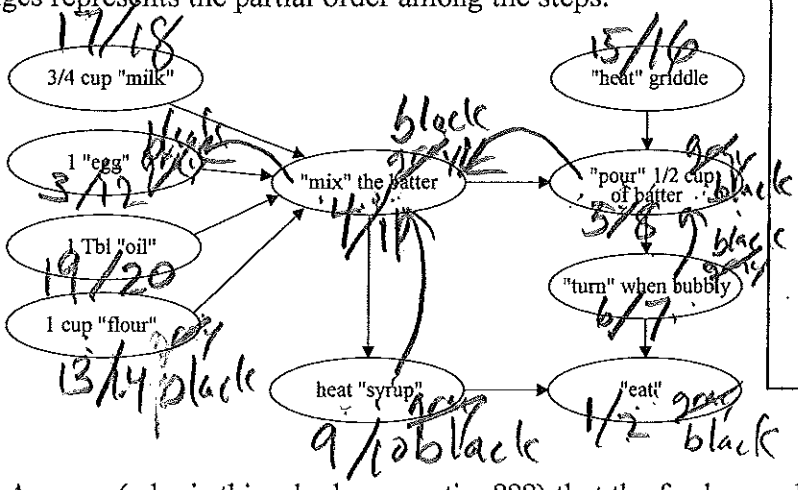
4. Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The `DFSGraph` uses a `time` attribute to note when a vertex if first encountered (`discovery` attribute) in the depth-first search and when a vertex in backtracked through (`finish` attribute). Consider the graph for 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)
```

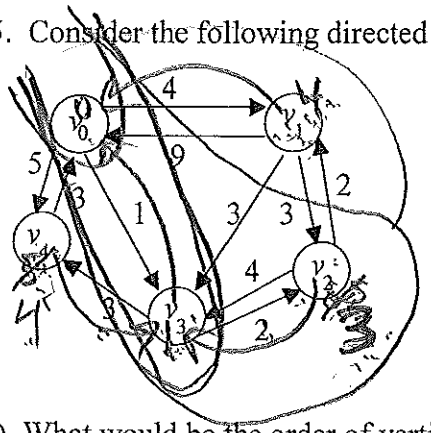
a) Assume (why is this a bad assumption???) that the for-loops alway iterate through the vertexes alphabetically (e.g., "eat", "egg", "flour", ...) by their id. Write on the above graph the `discovery` and `finish` attributes assigned to each vertex by executing the `dfs` method.

(see above)

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 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. Outline the steps to perform a topological sort.

(1) Load the graph information into a DFSGraph.
(2) Involve its dfs method to set the discovery and finish times
(3) Extract the topological sort by printing the vertex ids 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 or v_0, v_4, v_1

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

Vertex not in the subgraph with shortest distance to v_0 [illegible] in the su

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

Since we want the minimum distance, use a priority queue "min-Heap" of vertices not in the subgraph with their distance as the priority

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