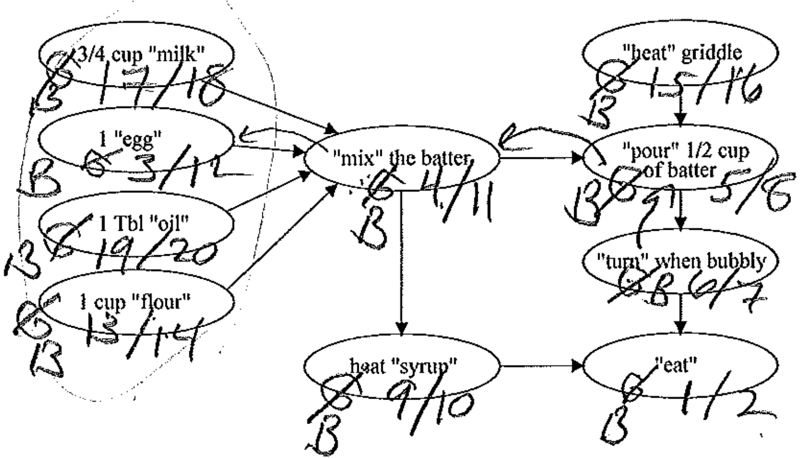
4. Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The `DFSGraph` uses a `time` attribute to note when a vertex if first encountered (`discovery` attribute) in the depth-first search and when a vertex in backtracked through (`finish` attribute). Consider the graph for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

time

~~0 1 2 3 4 5 6~~

~~7 8 9 10 11~~ 12



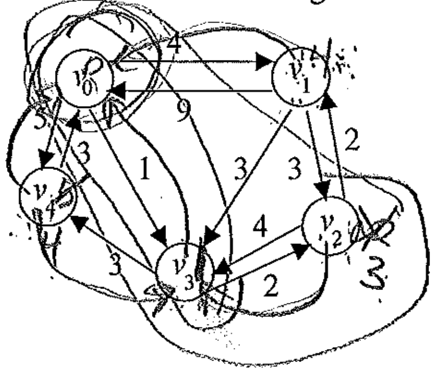
a) Assume (why is this a bad assumption???) that the for-loops alway iterate through the vertexes alphabetically (e.g., "eat", "egg", "flour", ...) by their id. Write on the above graph the `discovery` and `finish` attributes assigned to each vertex by executing the `dfs` method.

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.

Descending order of finish times

oil
milk
heat
flour
egg
mix
syrup
pour
turn
eat

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 / V_4, V_1}

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

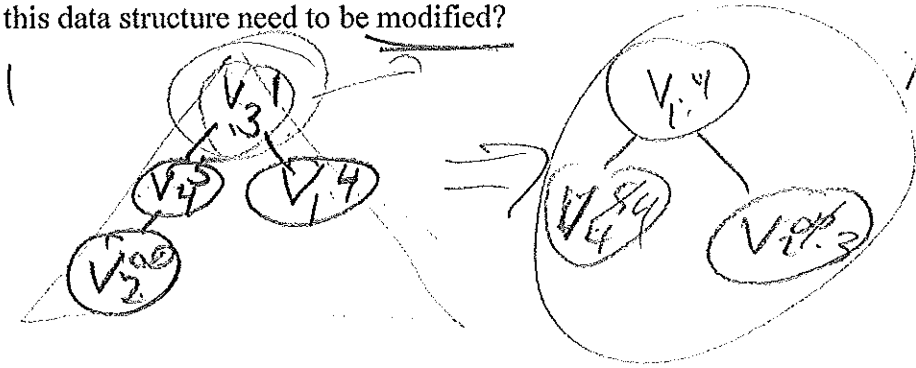
One with shortest distance to V_0

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

priority queue / min heap

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

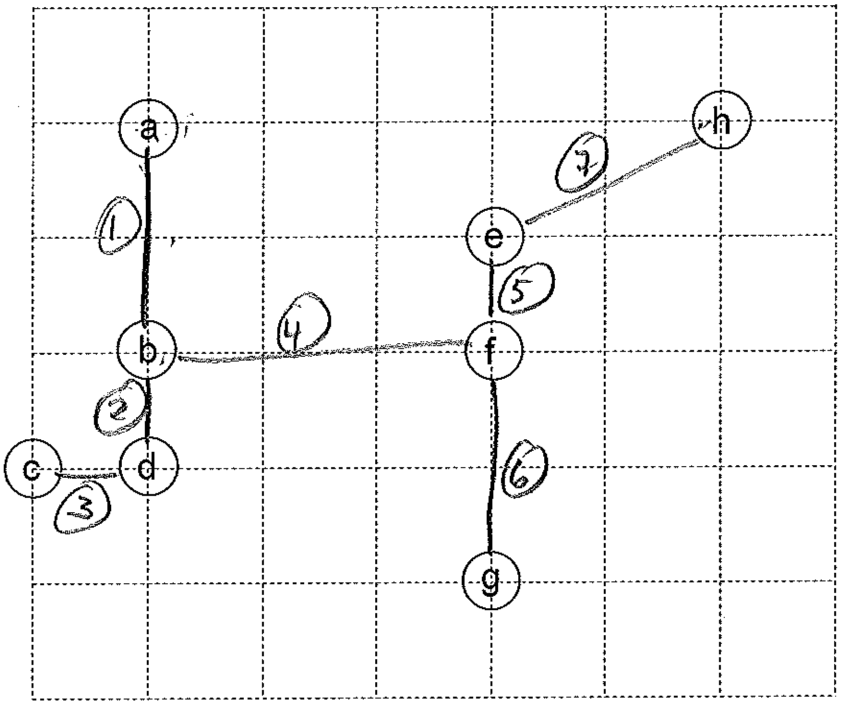
initial



need ~~it~~ to adjust priority/distance causing Vertices to percUp

Data Structures (CS 1520) Lecture 27 Name:________________

1. Suppose you had a map of settlements on the planet X
(Assume edges could connecting all vertices with their Euclidean distances as their costs)



We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads build to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at "a".

a) Assuming we start at city "a" which city would you connect first? b Why this city?

It is the closest city to 'a'

b) What city would you connect next to expand your partial road network?

next closest to anything in partial road system

(priority queue / min. heap)

c) What would be some characteristics of the resulting "graph" after all the cities are connected?

"connected" graph with no cycles ⇒ "spanning tree"

d) Does your algorithm come up with the overall best (globally optimal) result?

bestest / min. spanning tree - MST

2. Prim's algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a *greedy algorithm*. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO 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).

a) What greedy criteria does Prim's algorithm use to select the next vertex and edge to the partial minimum spanning tree? Vertex closest to any vertex in partial MST

b) Consider the textbook's Prim's Algorithm code (Listing 7.12 p. 346) which is incorrect.

```
def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert) \
                  + currentVert.getDistance()
          if v in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

c) What is wrong with the code? (Fix the above code.)

3. To avoid "massive" changes to the `BinHeap` class, it can store `PriorityQueueEntry` objects:

```
class PriorityQueueEntry:
    def __init__(self,x,y):
        self.key = x
        self.val = y

    def getKey(self):
        return self.key

    def getValue(self):
        return self.val

    def setValue(self, newValue):
        self.val = newValue
```

```
    def __lt__(self,other):
        return self.key < other.key

    def __gt__(self,other):
        return self.key > other.key

    def __eq__(self, other):
        return self.val == other.val

    def __hash__(self):
        return self.key
```

a) Update the above Prim's algorithm code to use `PriorityQueueEntry` objects.

b) Why do the `__lt__` and `__gt__` methods compare `key` attributes, but `__eq__` compare `val` attributes?