Review lecture 30

The Final exam is Tuesday (May 2) from 8:00 - 9:50 AM in Wright 10. It will be closed-book and notes, except for **three** 8" x 11" sheets of paper containing any notes that you want. (Plus, the Python Summary Handout) About 75% of the test will cover the following topics (and maybe more) since the second mid-term test, and the remaining 25% will be comprehensive (mostly big-oh analysis and general questions about stacks, queues, priority queues/heaps, lists, and recursion).

Chapter 6: Trees
Terminology: node, edge, root, child, parent, siblings, leaf, interior node, branch, descendant, ancestor, path, path length, depth/level, height, subtree
General and binary tree recursive definitions
Tree shapes and their heights: full binary tree, balanced binary tree, complete binary tree
Applications: parse tree, heaps, binary search trees, expression trees
Traversals: inorder, preorder, postorder
Binary search tree ADT: interface, implementation, big-oh of operations
Balanced binary search trees: AVL tree ADT: interface, implementation, big-oh of operations

File Structures - Lecture 25 handout:
http://www.cs.uni.edu/~fienup/cs1520s17/lectures/lec24_questions.pdf
We talked about how the in memory data structures need to be adapted for slow disks.
From this discussion you should understand the general concepts of Magnetic disks:

- layout (surfaces, tracks/cylinders, sectors, R/W heads)
- access time components (seek time - moving the R/W heads over the correct track, rotational delay - disk spins to R/W head, data transfer time - reading/writing of sector as it spins under the R/W head)

Hash Table as a useful file structure
B+ trees as a useful file structure - see web resources:

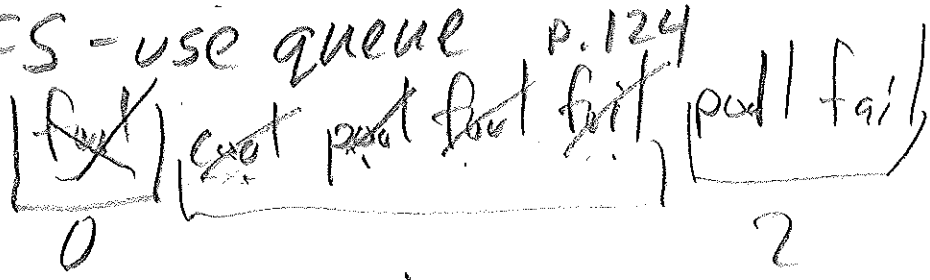
http://www.sci.unich.it/~acciaro/bpiutrees.pdf

http://en.wikipedia.org/wiki/B%2B_tree

http://www.ceng.metu.edu.tr/~karagoz/ceng302/302-B+tree-ind-hash.pdf

Chapter 7: Graphs
Terminology: vertex/vertices, edge, path, cycle, directed graph, undirected graph
Graph implementations: adjacency matrix and adjacency list
Graph traversals/searches: Depth-First Search (DFS) and Breadth-First Search (BFS)
General Idea of the following algorithms: topological sort, Dijkstra's algorithm (single-source, shortest path), Prim's algorithm (determines the minimum-spanning tree), TSP (Traveling-Saleperson Problem)
Approximation algorithm to solve TSP, general idea of backtracking and best-first search branch-and-bound.
You should understand the graph implementations and algorithms listed above. You should be able to trace the algorithms on a given graph.

BFS - use queue p. 124



1

DFS - use stack or recursion and run-time stack

p. 126

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 tranform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

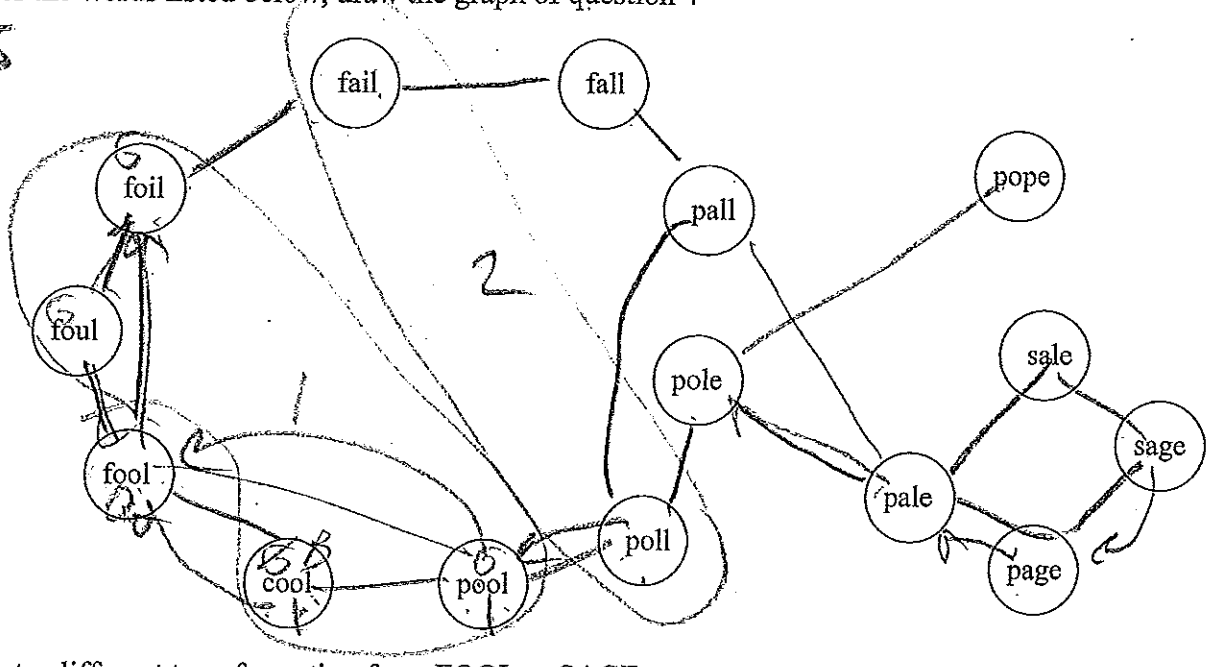
We can use a graph algorithm to solve this problem by constructing a graph such that

- a word represents a vertex
- an edge represents?

- a word ladder transformation from one word to another represents?

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

BFS



a) List a different transformation from FOOL to SAGE

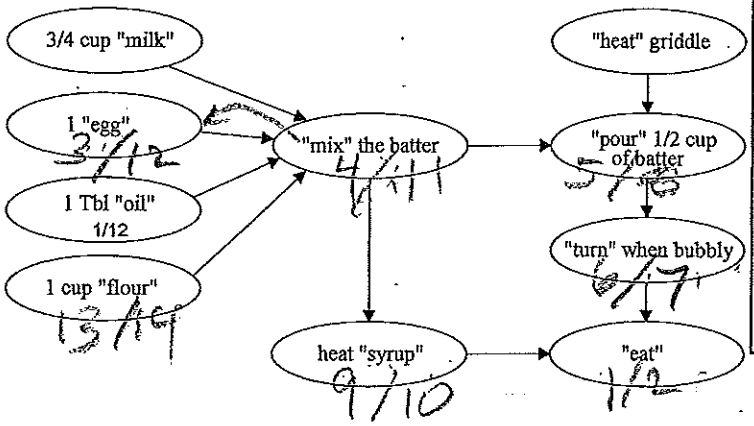
b) If we wanted to find the shortest transformation from FOOL to SAGE, what does that represent in the graph?

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?

Part B: 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) Run the `lab11/make_pancake_DFS.py` program. Write on the above graph the `discovery` and `finish` attributes (e.g., 1 / 12 of "oil") assigned to each vertex by executing the `dfs` method..

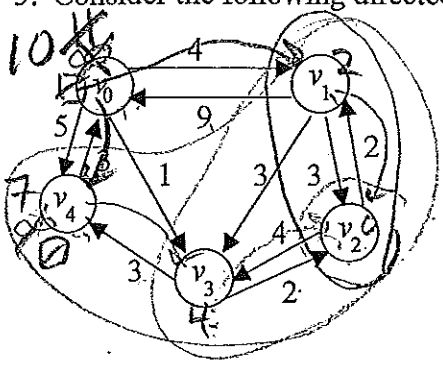
b) A *topological sort* algorithm can use the dfs `finish` attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour ½ cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. Outline the steps to perform a topological sort from the `finish` attributes.

After you have answered the above questions, raise your hand and explain your answers.

EXTRA CREDIT:
Add code to the end of the `made_pancake_DFS.py` program to print the topological sort for making pancakes.

Note Start at V_2

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?

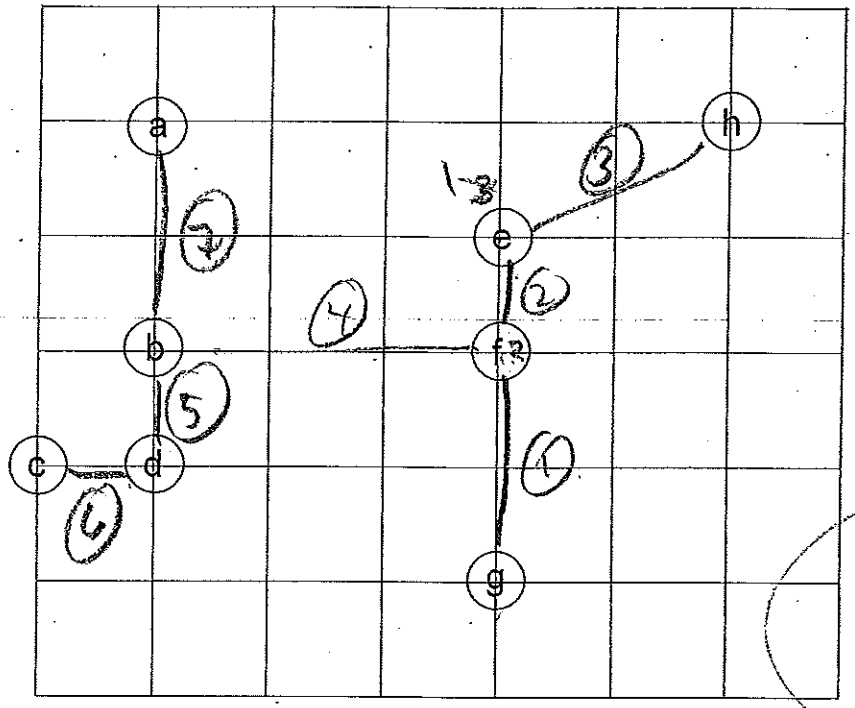
v_0,

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?

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? Why this city?

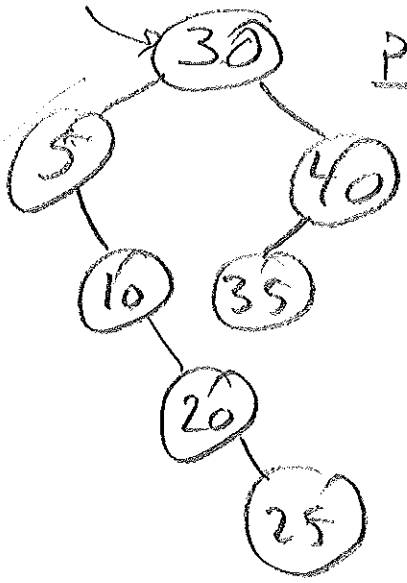
b) What city would you connect next to expand your partial road network?

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

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

put values:

BST 30, 5, 40, 10, 20, 25, 35



Preorder

30, 5, 10, 20, 25, 40, 35

AVL

put values, 30 5 40 10, 20 25 35