ENGN 2020:

Homework #6

Brown University School of Engineering

Assigned: March 18, 2019, Due: April 1, 2019

Problem 1

K23-2-1 and K23-2-2. Also label all verteces 0, 1, 2, $\cdots$, according to their distance from s. A purely graphical solution is fine.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Problem 2

K23-2-13. Solve the problem first as written. Then add an edge between nodes 1 and 6 of length 4, and solve again.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Problem 3

You will build an object-oriented version of Moore's algorithm in this problem. A useful strategy can be to think about how your data should be structured, then focus on writing the methods you will need to modify that data. Then, when you go to write the algorithm it should be rather straightforward. Recall that there were two basic types of data we used in the chalkboard implementation of Moore's algorithm: a *queue* and a *step table*. These are shown in Table 1.

Table 1: Moore's algorithm as we might do it on a computer, with a queue and a step table. This shows the state of the objects after 4 steps. Note that the items crossed off the queue have been processed and would be removed from the queue.

Queue

Node	Distance
~~I~~	~~0~~
~~G~~	~~1~~
~~K~~	~~2~~
~~J~~	~~2~~
H	2
A	3
E	3
F	3

Step table

Node	Distance	New neighbors
I	0	(G, K)
G	1	(J)
K	1	(H)
J	2	(A, E, F)

Part a. The queue is your list of what vertex to process next: you simply append new items to the end and "pop" the next item to process from the beginning.[1] Build a `Queue` class that behaves as follows, taking in the `Vertex` object you created last week:

```
>>> queue = Queue()
>>> queue.append(vertex=Vertex('I'), distance=0)
>>> queue.next()
{'distance': 0, 'vertex': <__main__.Vertex object at 0x7fcacb59f4a8>}
>>> queue.append(vertex=Vertex('G'), distance=1)
>>> queue.append(vertex=Vertex('K'), distance=2)
>>> queue.next()
{'distance': 1, 'vertex': <__main__.Vertex object at 0x7fcacb59f518>}
```

Submission

Label: hw6_3a
Points: 0.5
Class initialization variables: (none)
Class method(s): `append` (which takes in `vertex` (an object) and `distance` (an integer)), `next` (which takes no arguments and returns a dictiary with keys `vertex` and `distance`)

(Note that the above is only a crude check that your class works; you should make sure it does what you expect before moving on.)

Part b. The step table contains the list of processed verteces, their distances from the origin, and the new neighbors that they have discovered. When we add items to the step table, it should first check to see if the neighbors discovered are actually new. Use the below skeleton of a `StepTable`

[1] In principle, we could do this directly with the "append" and "pop" commands of a python list, but for the sake of practice we'll build our own class. Your class can internally use a python list and its append and pop methods.

class and write its `get_new_neighbors` method such that the `append` method works as you expect.

File attached here.

```python
class StepTable:
    """Container to remember the steps taken by Moore's algorithm.
    "data" is a list of steps, where each step contains the vertex
    object, the distance from the origin, and the new neighbors
    encountered of that vertex. E.g.,

    data = [ (<vertex object>, 0, ['B', 'D']),
             (<vertex object>, 1, ['C']),
             (<vertex object>, 1, ['F']),
             ...]
    """

    def __init__(self, data=None):
        if data is None:
            self.data = []
        else:
            self.data = data

    def append(self, vertex, distance):
        """Adds the given vertex object to the step table.
        Distance is the distance from the origin."""
        neighbors = vertex.get_neighbors()
        new_neighbors = self.get_new_neighbors(neighbors)
        self.data.append((vertex, distance, new_neighbors))

    def get_new_neighbors(self, neighbors):
        """Compares the vertex names in the list "neighbors"
        to the neighbors already contained in steptable.
        Returns a (shorter) list of names of neighbors that have
        not yet been discovered.
        """
        # Write me!

    def get_reverse_path(self, vertex_name):
        """Starting at the vertex named "vertex_name", traces
        backwards through the step table to find the shortest
        distance to the origin. Note that this should only be
        called *after* vertex_name has been discovered."""
        # Write me!

    def print(self):
        """Attempts to pretty print the contents of the
        step table."""
        for row in self.data:
            print('{:10s} {:3d} {:s}'.format(row[0].name, row[1], str(row[2])))
```

Paper Submission

(Turn in a PDF to Canvas.)

Points: 0.5

Part c. After the step table is complete (or you have found the desired end vertex), you need a way to trace back its path to the origin. For example, in Table 1, if E was the desired end vertex, we can look at the table to trace it back to J, to G, then to I, resulting in a path I-G-J-E. Write the method `get_reverse_path` that takes in the name of the desired vertex (*e.g.*, `'E'`) and returns the shortest path to this vertex as a list (*e.g.*, `['I', 'G', 'J', 'E']`.)

Paper Submission

(Turn in a PDF to Canvas.)
Points: 0.5

Part d. Now that you have created your data structures and the methods you will need, it should be relatively straightforward to implement Moore's algorithm. Use your `Vertex`, `Queue`, and `StepTable` classes to write this code, and demonstrate that it works for arbitrary start and end points using the links dictionary from the previous problem set.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 0.5

Part e. Use your algorithm, along with code you wrote last week, to demonstrate that you can pick two websites from the Brown Engineering domain and find the shortest distance between them (within the Brown Engineering domain), if it exists. Turn in your complete code and at least two examples showing pages separated by at least 5 clicks.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Problem 4

K23-3-4 and K23-3-5. Your solutions should be a row-by-row table as we produced in class; *e.g.*, for problem 5:

Step	1	2	3	4	5
0	0	"2"	"∞"	"5"	"∞"
⋮					

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Problem 5

K23-4-4 and K23-4-5.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 1

Problem 6

Write a python program to implement Prim's algorithm as described in Table 23.6 of Kreyszig. The structure of your function or class is up to you, but you must turn in your original code. Use your program to solve Problems 6 through 9 in K23-5.

Paper Submission

(Turn in a PDF to Canvas.)
Points: 3