C200 Programming Assignment №10
Graphs, Stacks, & Queues

Professor M.M. Dalkilic
Computer Science
School of Informatics, Computing, and Engineering

Indiana University, Bloomington, IN, USA

April 19, 2025

Introduction

Due Date: 5:00 PM, Friday, April 25, 2025. This is the last assignment of the semester. As always this assignment will need to be submitted to Gradescope **and it will also need to be pushed to Github**. If you do not push to Github, you will **receive a penalty for the assignment**. **We do not accept late submissions**. Any form of cheating or plagiarism will be dealt with seriously. If this results in a grade change, we are compelled by the University to send a formal notification to the Dean, please review the policy on academic misconduct in the syllabus.

As always, all the work should be with you and your partner; but *both* of you should contribute. Please remember:

- You will be submitting on Gradescope and pushing to Github
- Your highest score of **10 submissions** will be used for your final score.
- Any function that is defined requires a docstring, please refer to lab 03 for what a docstring should have.
- Do **NOT** change the function names, or the parameters.
- Finally, manual grading will be done, so if you receive a 50 out of 100 on the autograder, it means the lowest grade you can receive is 50. The rest will come from manual grading of the problems.

If your timestamp is 5:01PM or later, the homework will not be graded. So **do not wait until 4:59 PM** to submit. This also includes pushing to Github, so make sure to also plan accordingly. If you have questions or problems with Gradescope, Git and/or Github, please visit office hours or make a post on Inscribe ahead of time. Since you are working in pairs, your paired partner is shown on canvas. Esure that you are using only features that we have discussed in lectures and labs.

Github and Gradescope

Download the a10.py and a10.pdf files from the Assignment10 page on Canvas and copy the files under the Assignment10 folder in your github repository (this repository should already be cloned in your laptop). Start working on the assignment, and remember to commit and push frequently.

As mentioned in the previous assignment, we will be requiring you to push your code to Github repository. **If you do not** we will be penalizing you for the assignment. Again, the reason is because we want to see your progress, it will also help you in future courses that you will take (yes a lot of courses use Github), and it will also help you in your future career.

You will still **be required** to also submit your assignment to Gradescope. **Do not** use the Github submission feature on Gradescope. Since we are using Github enterprise (University Github), there's no way for you to link your Github account to Gradescope, since it's not public.

Problem 1: Breath First Search (BFS)

During lab you learned about the stack and queue data structures. When it comes to traversing a graph, we use two main algorithms: Depth First Search (DFS) and Breath First Search (BFS). While DFS uses a stack, BFS on other hand uses a queue. A queue looks something like this:

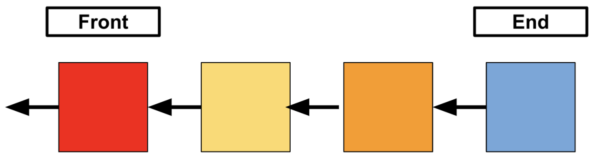

Figure 1: Queue

For this problem, you will implement both the BFS algorithm and the queue structure. You will be given the almost the same graph structure that was used in lab. To visually see how the BFS algorithm works, look at the following graph below (figure 2), starting at node 1:

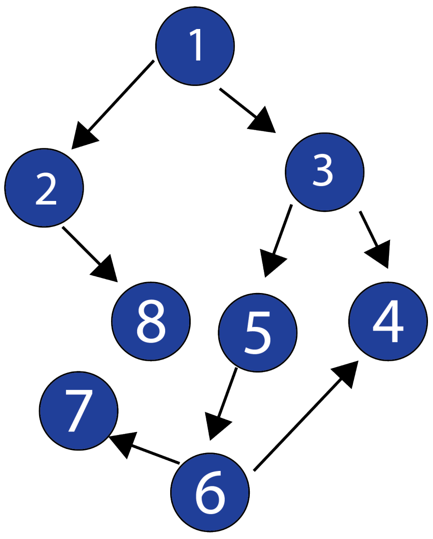

Figure 2: Graph

So how would the BFS algorithm work? For example we have a u, unvisited queue, where $u = \{1, 2, 3, 4, 5, 6, 7, 8\}$. If we take 5 and do BFS(5), then we will get a list of visited nodes, $v = \{5, 6, 7, 4\}$ or $v = \{5, 6, 4, 7\}$, either solution is correct as long as we have visited the apporpriate nodes. This is how it would look like:

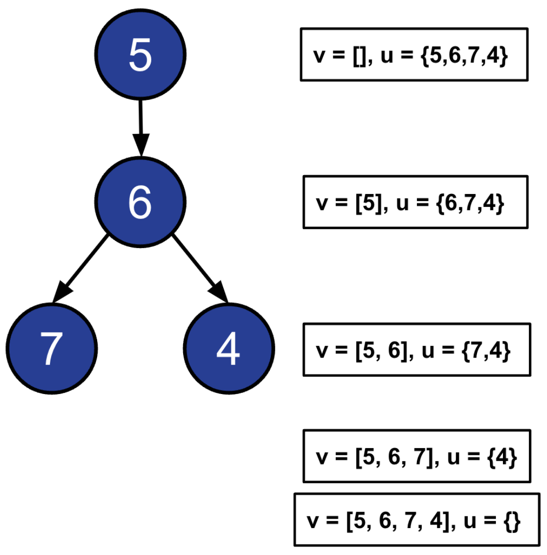

Figure 3: BFS

The following code:

```
nodes = [1,2,3,4,5,6,7,8]
g = Graph(nodes)
elst = [(1,2),(1,3),(2,8),(3,5),(3,4),(5,6),(6,7),(6,4)]
for e in elst:
    g.add_edge(e)

q = Queue()
q.enqueue(5)
print(g.bfs(q))
```

will produce the following output:

```
[5, 6, 7, 4]
```

Problem 1: BFS

- Reviewing the lecture slides will help you VERY much to understand this problem.
- You **must** implement the queue structure first before you implement the BFS algorithm. Do **not** use the queue module from python
- BFS **must** be done in recursion.
- Include a docstring for each function implemented.

Problem 2: Haversine Distance

Finding the distance between two points $p_1 = (x_1, y_1), p_2 = (x_2, y_2)$, $d(p_1, p_2)$ in a 2D plane is straightforward:

$$d(p_1, p_2) = [(x_1 - x_2)^2 + (y_1 - y_2)^2]^{1/2} \tag{1}$$

When calculating the distance on the Earth, however, we have to take into account Earth's shape. To that end, we can use the Haversine formula, named by Inman for the function:

$$h(\theta) = \sin^2\left(\frac{\theta}{2}\right) \tag{2}$$

called the *haversine*. If p_1, p_2 are on a Earth (sphere) we have:

$$h\left(\frac{d}{r}\right) = h(y_2 - y_1) + \cos(x_1)\cos(y_1)h(x_2 - x_1) \tag{3}$$

where d is the distance between the points for a sphere whose radius is r. To solve for d we take the inverse sin and muliply by r:

$$d = 2r \arcsin\left[h\left(\frac{d}{r}\right)^{1/2}\right] \tag{4}$$

NOTE that `[]` in `arcsin` can be treated as parentheses, we just use brackets for readability. Assume the points p_1, p_2 are latitude, longitude pairs. These pairs describe a point through intersection: The latitude runs parallel to the equator and longitude runs perpendicular to the equator. They intersect at a point. We now show how to calculate distance between two points on the Earth using $p_1 = (lat_1, lon_1), p_2 = (lat_2, lon_2)$ latitude and longitude:

$$\begin{aligned}
hd(p1, p2) &= \qquad (5)\\
lat_d &= \frac{lat_2 - lat_1}{2} \qquad (6)\\
lon_d &= \frac{lon_2 - lon_1}{2} \qquad (7)\\
d_0 &= \sin(lat_d)^2 + \cos(lat_1)\cos(lat_2)\sin(lon_d)^2 \qquad (8)\\
r &= 3961 \text{ mi.} \qquad (9)\\
d_1 &= 2r\arcsin(d_0^{1/2}) \qquad (10)
\end{aligned}$$

Where d_1 is the distance between the two points. When implementing in Python, you'll have to make sure to convert latitude and longitude to radians. You can use math's radian function. The function arcsin is implemented in math as `asin()`.

The following code:

```python
#Lindley Hall
#south side of campus
l1 = (39.165341,-86.523588)

#Luddy Hall
#on northside of campus
l2 = (39.172725,-86.523295)
print("haversine", hd(l1,l2), "mi")
```

will produce the following output:

```
haversine 0.5107158183712371 mi
```

Problem 2: Haversine

- Complete the function `hd`
- You are allowed to use the `radians` function in the `math` package
- Here is a URL that provides a different, but equivalent formula for finding distance `https://andrew.hedges.name/experiments/haversine/`.
- Include a docstring for the function implemented.

Problem 3: Queue It!

Queues are all around us and are used in many applications. Whether it's a queue for a line at a store, for a printer, or even for a computer program. They are really good for storing data that is waiting to be processed, espcially when it comes to music! Suppose we have the following data:

```
data = {"Beatles": [("Hey Jude", 100, "Pop"), ("Across the Universe", ↩
    200, 'Rock')],
        "Queen": [("Bohemian Rhapsody", 300, "Pop"), ("Don't Stop Me ↩
           Now", 150, "Rock")],
        "Miles Davis": [("So What", 400, "Jazz"), ("Freddie Freeloader↩
           ", 350, "Jazz")],}

genres = ["Pop", "Rock", "Jazz"]
```

In this data, we have the artists as keys and the values being the songs, active listeners, and the genre of the song. At the same time, we have a list of our favorite genres in order from most favorite being Pop, and least being Jazz. We will need to create a queue that will play the songs based on the order of our favorite genres, which are given in the list named genres, and then sort from the most active listeners to the least active listeners for each genre.

The following code:

```
    genres = ["Pop", "Rock", "Jazz"]
    bandAndSong = {"Beatles": [("Hey Jude", 100, "Pop"), ("Across the ↩
       Universe", 200, 'Rock')],
                   "Queen": [("Bohemian Rhapsody", 300, "Pop"), ("Don'↩
                      t Stop Me Now", 150, "Rock")],
                    "Miles Davis": [("So What", 400, "Jazz"), ("↩
                       Freddie Freeloader", 350, "Jazz")],}
    q = Queue()
    songQueue = queue_it(bandAndSong, genres, q)
    print(songQueue)
    print(f"\nFirst Song: {songQueue.dequeue()}")
    print(f"Second Song: {songQueue.dequeue()}")
    print(f"Third Song: {songQueue.dequeue()}")
```

will produce the following output:

```
    [['Queen', ('Bohemian Rhapsody', 300, 'Pop')], ['Beatles', ('Hey ↩
       Jude', 100, 'Pop')], ['Beatles', ('Across the Universe', 200, '↩
       Rock')], ['Queen', ("Don't Stop Me Now", 150, 'Rock')], ['Miles↩
        Davis', ('So What', 400, 'Jazz')], ['Miles Davis', ('Freddie ↩
    Freeloader', 350, 'Jazz')]]
First Song: ['Queen', ('Bohemian Rhapsody', 300, 'Pop')]
Second Song: ['Beatles', ('Hey Jude', 100, 'Pop')]
Third Song: ['Beatles', ('Across the Universe', 200, 'Rock')]
```

Problem 3: Queue It!

- For testing purposes, we will use the same queue structure that you will implement in problem 1. So make sure you implement the queue structure first.
- You **cannot** use the queue module from python, again use the one from Problem 1
- You are allowed to use the list.sort() function, and also highly recommended to use it.
- The Queue structure must be returned
- Include a docstring for the function implemented.

Problem 4: The Editor

Having an editor is very useful for programming, text editing, and other tasks. Yet, how does one implement an editor? The idea behind an editor is to use a stack(s) to help the editor out. The stack(s) will hold the text that is being added, deleted, or modified, for the `Stack` you **must use the one from lab**. Since building a whole editor is a bit too much, we will focus simply on typing, undoing and redoing. Suppose we have the following entries:

- TYPE a
- TYPE b
- TYPE c
- UNDO
- UNDO
- REDO

So we will need to TYPE "a", then "b", then "c". After that we will need to UNDO twice on the last two commands. Finally we will need to REDO the last command. The following code:

```python
print(editor(["TYPE a", "TYPE b", "TYPE c", "UNDO", "UNDO", "REDO"↩
    ]))
print(editor(["TYPE x", "TYPE y", "UNDO", "TYPE z", "REDO"]))
print(editor(["UNDO", "REDO", "TYPE a"]))
print(editor(["TYPE h", "TYPE i", "UNDO", "REDO", "TYPE !"]))
```

produces:

```
ab
xz
a
hi!
```

Problem 4: The Editor

- Implement the function.
- You must use the stack structure created in lab to implement the editor.
- Do **not** use the stack module from python, again you must use the stack structure created in lab.
- Only string operation that we have discussed in class can be used.
- Include a docstring for the function implemented.

Problem 5: Distance between a point and line

In a 2D Euclidean plane, the formula for determining the distance between a point and line is:

$$d(point, line) = \frac{|ax' + by' + c|}{\sqrt{a^2 + b^2}} \tag{11}$$

$$point = (x', y') \tag{12}$$

$$line = (ax + by + c) \quad \text{for variables } a, b, c \tag{13}$$

The following code:

```python
line = (4,6,-26)
point = (2,-4)
print(d(point,line))
```

produces:

```
5.824
```

Problem 5: Distance between a point and line

- Complete the function.
- Round the output to three decimal places.
- Include a docstring for the function implemented.