Homework 9

Due Friday, July 30 at 12:59 PM

Fun Sea Creature Fact: A shrimp's heart is in its head!

Installing and Handing In

Accept the GitHub assignment **here**. Homework is handed in through Gradescope. For written portion, submit a PDF and match the questions accordingly. For the coding portion, submit ALL of the .py files.

Written Problems

Problem 9.1

Negative weights

Purpose: Critically thinking about Dijkstra's algorithm
Resources: Shortest Paths Lecture (June 24)

The proof of Dijkstra's algorithm relies on all the weights being non-negative. However, you've heard rumors that there's a new version of Dijkstra's algorithm that computes the shortest path *and* accounts for negative weights.

This new algorithms calls the weight of the most negative (i.e. smallest) edge, m. The algorithm then begins by adding $|m|$ (absolute value of m) to the weight of each edge, thus making all weights non-negative. It then uses the original Dijkstra's algorithm to find the shortest path.

Does this algorithm still find the shortest path of the original graph (before $|m|$ is added)? Why or why not? (Note: if you conclude the new algorithm doesn't work, instead of explaining why not, you may give a counterexample).

Problem 9.2

Connecting Aquariums

Purpose: Critically thinking about graphical representations
Resources: Intro to Graphs Lecture (June 22), Shortest Paths Lecture (June 24), MSTs Lecture (June 29), and More MSTs Lecture (July 1)

The New National Aquarium's location is referred to as location Q. All of the other aquariums need to be reachable from this starting point, Q. The aquarium locations are a set of points P_i $(i = 1 \dots n)$. *Note: Q is not included in the set of points P_i.* An aquarium's location can be connected to the starting point directly by a single path or by a chain of paths. This means location 5 could be

connected by a path to location 2, which could then be connected by a path to the starting point Q; now locations 2 and 5 are both connected to the starting point as required. Each path is a distance, so it will always be positive. Also note that if a path directly connects two aquarium locations the path will be a straight line.

Assume that you're given the locations of the points P_i and the starting point, Q, and that $d(A, B)$ is a constant-time function that gives the straight-line distance between A and B, (straight-line distance means that if one were to draw a line between A and B, without regarding paths, the length of this line would be $d(A, B)$). Your goal is to find the sequence of paths that cover the least total distance and connect all aquariums to the starting point.

Describe how to model this problem as a graph problem, and then give a brief paragraph description of your solution. If your solution uses an algorithm from class (this is not required), no need to explain the intricacies of the algorithm, instead focus on why this algorithm solves this problem. Also be sure to include what the running time of your algorithm is in terms of n.

Python Problems

Problem 9.3

Functional Programming Practice

Purpose: Brief practice with functional programming to prep you for upper-level courses
Resources: Functional Programming Lecture (July 20) and Section 8

Overview

In `functional.py` fill in the corresponding stencil functions to solve each of the below problems using **ONLY** python's built in `map` or `reduce` (it is up to you to decide which one is appropriate for the problem).

Details

- You should not be using any other function calls, and you should also not need to use any for loops.
- The functions you pass into `map` or `reduce` must be anonymous functions.
- The solution to each one is a single line of code (besides the error handling).
- If you want to return a list of what map returns, you cannot call `return map(blah blah...)` because `map` returns a map object (which is an iterator). Instead, you must cast this to a list by calling `return list(map(blah blah...))`

Input/Output

Each problems' input and output are outlined below. Remember that you can assume your function will be tested on anything that fits within the below definition for Input, but will **not** be tested on inputs that don't fit within the below input definition.

Testing

Write your test cases in `functional_test.py`. A few examples have already been filled in for you. **DO NOT** write your tests within the example test functions we provide! Our scripts will skip the test functions we provide, so write your own functions to test your code thoroughly.

Part 1: apply_all

Input

- A list of unary (one argument) functions and a number
- This list can be empty
- Both the list of functions, and the number, can also be `None`. Note: the list's items will not be `None`

Output

- If the input list is not `None`: A new list with each of those functions applied to that number. In other words, if you pass in the list of functions `[f(x), g(x), h(x)]` and the number `n`, `apply_all` should produce `[f(n), g(n), h(n)]`.
- If the input *list* is `None`: raise an `InvalidInputException`.

Example

`apply_all([lambda x: x+1, lambda x: x+2, lambda x: x+3], 4)` → `[5, 6, 7]`

Part 2: compose

Input

- A list of unary (one argument) functions and a number
- This list can be empty
- Both the list of functions, and the number, can also be `None`. Note: the list's items will not be `None`

Output

- **Input** A list of unary (one argument) functions and a number. Both the list of functions, and the number, can also be `None`. Note: the list items will not be `None`

- If the input list is not `None`: a single number that is the composition of all of the functions in the list applied to `n`. The inner-most function in the composition will be the first function in the input list, and the outer-most function in the composition will be the last function in the input list. In other words, if you pass in the list of fuctions `[f(x), g(x), h(x)]` and the number `n`, `compose` should produce `h(g(f(n)))`.
- If the input *list* is `None`: raise an `InvalidInputException`.

Example
`compose([lambda x: x+1, lambda x: x+2, lambda x: x+3], 4)` → `10`.

Part 3: list_compose_steps

Input

- A list of unary (one argument) functions and a number
- This list can be empty
- Both the list of functions, and the number, can also be `None`. Note: the list's items will not be `None`

Output

- **Input** A list of unary (one argument) functions and a number. Both the list of functions, and the number, can also be `None`. Note: the list items will not be `None`
- If the input list is not `None`: a list with each of the intermediate values produced by `compose` In other words, if you pass in the list of functions `[f(x), g(x), h(x)]` and the number `n`, `list_compose_steps` should produce `[n, f(n), g(f(n)), h(g(f(n)))]`
- If the input *list* is `None`: raise an `InvalidInputException`.

Example
`list_compose_steps([lambda x: x+1, lambda x: x+2, lambda x: x+3], 4)` → `[4, 5, 7, 10]`.

Additional Details Please note that using `append` will not work in your lambda expression because `append` doesn't return anything. To append some number x to a list, you should write `list+[x]` instead of `list.append(x)`.

Problem 9.4

Dijkstra's Algorithm

Purpose: Implementing Dijkstra's algorithm
Resources: Shortest Paths Lecture (June 24)

Overview

In the file `dijkstra.py` implement dijkstra's algorithm so that your function finds the shortest paths from a given vertex to all other vertices.

Input/Output

You can assume your function will be tested on anything that fits within the below definition for Input, but will **not** be tested on inputs that don't fit within the below input definition.

Input:

- `g` and `src`.

  `g` is a connected, undirected, graph with positive (0 included) edge weights. You can assume that there will be no two edges between the same vertices. This graph will be an instance of `MyGraph`, which is declared in `mygraph.py`.

  `src` is a vertex. Note that `src` is not guaranteed to exist within `g`.

- `g` or `src`, or both can be `None`

Output:

- A **tree** that contains the shortest paths from `src` to all other vertices. This tree will be an instance of `MyGraph`, which is declared in `mygraph.py`. Note, since the output should have the properties of a tree, your output must not contain any cycles.
- If either `g` or `src` are `None` throw an `InvalidInputException`
- If `src` is the only node in the graph, meaning there are no shortest paths, return just the `src` node.
- If `src` is not contained within `g`, throw an `InvalidInputException`

Details

- You may not manipulate or destroy the graph. Decorations are okay, but do not add or remove vertices or edges.
- To get the weight between two edges call `MyEdge.element()`, where `MyEdge` is the edge in question.

- To figure out how to make your own tree/graph, look in `mygraph.py`. Within the `MyGraph` class are all the methods you can call on an instance of `MyGraph`.
- We have imported `heappriorityqueue` in case you decide this data structure would be useful. If you intend to create a `heappriorityqueue` you would initialize it by calling `q = HeapPriorityQueue()`.

Testing

Write your tests in `dijkstra_test.py`. Remember to think about how to test an implementation if there are multiple shortest paths.