Homework: GPS

Table of Contents

Instructions

For this assignment we will writing a program that will use an adjacency list to build City and Map objects with the end of finding the shortest route to a destination from a starting place of our choosing. For this assignment the data is provided to you along with the code contained within the `if __name__ == "__main__"`. We will also assume that your script is named `gps.py`.

Additionally we will be using an algorithm named `Breadth First Search`. BFS is a well known algorithm used to search tree data structures. We start with a root node, then we look at all nodes in the next level down before moving to the subsequent level. I provide pseudocode for this algorithm. If you would like to know more, please take a look at this MIT: Lecture 13: Breadth-First Search (BFS). BFS is a very well known algorithm popularly used in coding interviews. If you are interested in potentially being a Software Engineer in the future, I highly recommend familiarity with BFS.

Your script should contain two classes, `City` and `Map`, and at least the following functions: `bfs`, `main` and `parse_args` that exist outside of any class. At the end of the script should be an `if __name__ == "__main__":` statement. Specifications for each of these required program elements are given below. You may write additional classes, methods, and/or functions if you wish.

`City class`

Functionality

This class holds data representing a City. We assume that Cities have names and neighbors. We also assume that the Neighbor relationships have associated values such as the distance between cities, and the interstates that connect them.

Attributes

- name: the input string that contains the name of the city.
- neighbors (dict): Starts off as an empty string that we will populate with the `add_neighbor()` method later. The keys of this dictionary will be other `City` objects that are connected to this instance of `City`. The values of those keys are tuples where the first item is the distance between the cities(int), and the interstate(str) that connects them.
  - Ex. {'Washington' : (45, '95')}

Methods

__init__()

- **Parameters**
  - self
  - Name(str) - The name of the City.
- **Functionality**
  - Set the `name` argument to an attribute.
  - Set the `neighbors` attribute to an empty dictionary.

__repr__()

- **Parameters**
  - self
- **Functionality**
  - Return the `name` attribute of the instance.

add_neighbor()

- **Parameters**
  - self
  - neighbor(City) - The `City` object that will be connected to this instance (and vice versa).
  - distance(str) - The distance between the two cities.
  - interstate(str) - The interstate number that connects the two cities.
- **Functionality**
  - If the neighbor City being passed in is not present within the `neighbors` attribute of this instance as a key.
    - Create a key/value pair in the `neighbors` attribute such that the key value is the neighbor

City being passed in and the value is a tuple containing the distance between the cities and the interstate that connects them (as a string).

        - Ex. {'Washington' : (45, '95')}
    - If the current instance is not in the neighbor City's `neighbors` attribute as a key:
        - Create a key/value pair in the `neighbors` attribute of the neighbor City being passed in such that the key value is the current instance and the value is a tuple containing the distance between the cities and the interstate that connects them (as a string).
            - Ex. {'Washington' : (45, '95')}

Map class

Functionality

This class stores the map data as a form of Graph where each node in the Graph is a city, and the edges are represented by the relationships that the cities have to each other.

Attributes

- cities: a list of all of the unique city objects that make up the Graph structure.

Methods

__init__()

- **Parameters**
    - self
    - relationships(dict) - A dictionary where the keys are individual cities and the values are a list of tuples where the
        - first element in the tuple is a string representing a City that is connected to the key (city),
        - the second element is the distance between the cities
        - and the third element is the interstate that connects them.

NOTE	This structure is common in the creation of Graph. This data structure is commonly known as an adjacency list (though it may be known by other terms). You do not need to know about social Graphs for this assignment but its helpful to know that in graphs nodes are 'objects' that are connected to other nodes via edges. These edges may be directed (meaning they have direction), and weight (some value that represents an attribute to the edge). Our graph is an undirected weighted graph (sorta).

- **Functionality**
    - Set the `cities` attribute to an empty list.
    - For each key in the `relationships` argument being passed in:
        - If the key (**which is a string that represents a city**) is not found in any of the `City`

objects currently within the `cities` attribute:

    - Create a `City` object with that name (**the key**) and append that city to the `cities` attribute.
- Identify which one of the `City` objects within the `cities` attribute has the name that is the same as the key that we are currently working with.
    - In essence, find the corresponding `City` object by its name.
    - IMPORTANT: You need to only get the index position of that object for now, you will need it for a later step.
- For each tuple in the value corresponding to this key: (NOTE: Another way of saying this is, for each neighbor to the city that we are currently working with) (HINT: Sequence unpacking is your friend)
    - If the connecting city ('neighbor string' so to speak) is not found in any of the `City` objects currently within the `cities` attribute: (NOTE: Similar to what we did earlier)
        - Create a `City` object with that name (**the neighbor string**) and append that city to the `cities` attribute.
    - Identify which one of the `City` objects within the `cities` attribute has the name that is the same as the neighbor string that we are currently working with.
        - In essence, find the corresponding `City` object by its name.
        - IMPORTANT: You need to only get the index position of that object for now, you will need it for a later step.
    - Using the previously identified index positions, use the `add_neighbor()` method of the `City` object to connect each city together.
        - NOTE: You dont need to do any checking to see if they are already connected as neighbors, the `add_neighbor()` method should already do this for you.

__repr__()

- Parameters
    - Self
- Functionality
    - Return the string representation of the `cities` attribute.

`bfs()`

Functionality

This function will implement the bfs (Breadth First Search) algorithm to find the shortest paths between the two nodes in a graph structure. In this case we need an implementation for an undirected, unweighted graph.

NOTE	But Prof Cruz said earlier in these instructions that this graph is weighted, what gives? Well yes, the graph is weighted, but a BFS implementation for a weighed graph is slightly more complex and I would rather keep it simple.

NOTE	You are not expected to know BFS by heart. Instead follow this pseudocode below carefully.

Parameters

- Graph(Map) - A map object representing the graph that we will be traversing.
- Start(str) - The start city in a roadtrip for example.
- Goal(str) - The destination city in a roadtrip for example.

Returns

A list of strings (cities) that we will visit on the shortest path between the start and goal cities.

Pseudocode

- Create an empty list named `explored`. This will be the list that contains all the cities that we have visited.
- Create a list within a list. The inner list should contain the `start` city to start and nothing else. Name this list of lists `queue`.
- While `queue` is not empty.
  - Pop the first element (much like a regular `queue`) and save it. This is a path.
  - Identify the last node in this path and save it (DO NOT POP IT!)
  - If the node that we just identified is not currently present in explored:
    - Using the graph that was passed in, find the node who's name matches the current value of node. Save the `Neighbors` attribute of this object. This attribute should be a dictionary.
      - NOTE: You have done this before, think back to how this was done in the `Graph` class.
    - For each key in the dictionary representing the neighbors to this node:
      - Explicitly convert the path we have been working with to a list and save this value to a variable named `new_path`. This is a suggested new path that may or may not lead to the shortest path.
      - To the `new_path` append the neighbor (which is the current key we are iterating over).
      - Append the `new_path` to the `queue`.
      - If the string representation of neighbor (which is the current key we are iterating over) is the same value as the `goal` argument then we have reached our destination.
        - Return a list containing the string representation of each element in the `new_path`. (NOTE: List comprehensions are your friend)
    - Append the node to `explored`.
- If after going through the nodes, we can find no path then we must print a message stating so.

- Return `None`

`main()`

Functionality

This function will create a Map object with the connections data being passed in. It will then use `bfs()` to find the path between a start City and a destination City. It will parse the returned value and instruct the user on where they should drive given a start node and an end node.

Parameters

- start(str) - The start city in a roadtrip for example.
- destination(str) - The destination city in a roadtrip for example.
- Graph(dict) - A dictionary representing an adjecency list of cities and the cities to which they connect.

Returns

A string that contains all of the same contents that we have printed out to the console/terminal.

Pseudocode

- Create a `Map` object using the `connections` attribute.
- Get the cities that you will traverse (for example on a roadtrip) given a start city and a destination city using `bfs()`. I will call this the 'instructions'
  - NOTE: This should include the start and destination cities.
- Try to perform the following
  - Declare an empty string
    - This string will contain the same content that will be printed.
  - For each of the cities(strings) in the 'instructions':
    - If the current element is the first element:
      - Print a message with the following syntax:
        - "Starting at {starting city}"
      - Add this same printed string to the string declared earlier
    - If the current element is anything before the last element:
      - Identify the next city in the list (where we have to go next, this value should be a string)
      - Using the map object that you created earlier, find the city who's name is the same as the current city we are iterating over. Using this City object, identify the neighbors to this city (dictionary). To make our lives easier in the next step, you will also need to convert this dictionary's keys to strings.
      - Get the value who's key is the next city within the recently identified neighbors

dictionary. (See previous step)

- This is a tuple where the first element is an int that represent the distance between the two cities, and the second element is the interstate that connects them.

- Print a message with the following syntax:
  - "Drive {distance_to_drive} miles on {interstate} towards {next_city}, then"
- Add this same printed string to the string declared earlier

- If the current element is the last element:
  - Print a message with the following syntax:
    - "You will arrive at your destination"
  - Add this same printed string to the string declared earlier
- Return the string declared earlier.

- In the case of an error, quit the program

IMPORTANT	You will most likely need to use the enumerate function in this function. Enumerate helps keep track of the element that you are on during a for loop along with that element's index at the same time!

Template

```
"""Create routes between cities on a map."""
import sys
import argparse

# Your implementation of City, Map, bfs, and main go here.

def parse_args(args_list):
    """Takes a list of strings from the command prompt and passes them through as
arguments

    Args:
        args_list (list) : the list of strings from the command prompt
    Returns:
        args (ArgumentParser)
    """

    parser = argparse.ArgumentParser()

    parser.add_argument('--starting_city', type = str, help = 'The starting city in a
route.')
    parser.add_argument('--destination_city', type = str, help = 'The destination city
in a route.')

    args = parser.parse_args(args_list)
```

```
    return args

if __name__ == "__main__":

    connections = {
        "Baltimore": [("Washington", 39, "95"), ("Philadelphia", 106, "95")],
        "Washington": [("Baltimore", 39, "95"), ("Fredericksburg", 53, "95"),
("Bedford", 137, "70")],
        "Fredericksburg": [("Washington", 53, "95"), ("Richmond", 60, "95")],
        "Richmond": [("Charlottesville", 71, "64"), ("Williamsburg", 51, "64"),
("Durham", 151, "85")],
        "Durham": [("Richmond", 151, "85"), ("Raleigh", 29, "40"), ("Greensboro", 54,
"40")],
        "Raleigh": [("Durham", 29, "40"), ("Wilmington", 129, "40"), ("Richmond", 171,
"95")],
        "Greensboro": [("Charlotte", 92, "85"), ("Durham", 54, "40"), ("Ashville",
173, "40")],
        "Ashville": [("Greensboro", 173, "40"), ("Charlotte", 130, "40"), (
"Knoxville", 116, "40"), ("Atlanta", 208, "85")],
        "Charlotte": [("Atlanta", 245, "85"), ("Ashville", 130, "40"), ("Greensboro",
92, "85")],
        "Jacksonville": [("Atlanta", 346, "75"), ("Tallahassee", 164, "10"), ("Daytona
Beach", 86, "95")],
        "Daytona Beach": [("Orlando", 56, "4"), ("Miami", 95, "268")],
        "Orlando": [("Tampa", 94, "4"), ("Daytona Beach", 56, "4")],
        "Tampa": [("Miami", 281, "75"), ("Orlando", 94, "4"), ("Atlanta", 456, "75"),
("Tallahassee", 243, "98")],
        "Atlanta": [("Charlotte", 245, "85"), ("Ashville", 208, "85"), ("Chattanooga",
118, "75"), ("Macon", 83, "75"), ("Tampa", 456, "75"), ("Jacksonville", 346, "75"),
("Tallahassee", 273, "27") ],
        "Chattanooga": [("Atlanta", 118, "75"), ("Knoxville", 112, "75"), (
"Nashville", 134, "24"), ("Birmingham", 148, "59")],
        "Knoxville": [("Chattanooga", 112,"75"), ("Lexington", 172, "75"),
("Nashville", 180, "40"), ("Ashville", 116, "40")],
        "Nashville": [("Knoxville", 180, "40"), ("Chattanooga", 134, "24"),
("Birmingam", 191, "65"), ("Memphis", 212, "40"), ("Louisville", 176, "65")],
        "Louisville": [("Nashville", 176, "65"), ("Cincinnati", 100, "71"),
("Indianapolis", 114, "65"), ("St. Louis", 260, "64"), ("Lexington", 78, "64") ],
        "Cincinnati": [("Louisville", 100, "71"), ("Indianapolis,", 112, "74"),
("Columbus", 107, "71"), ("Lexington", 83, "75"), ("Detroit", 263, "75")],
        "Columbus": [("Cincinnati", 107, "71"), ("Indianapolis", 176, "70"),
("Cleveland", 143, "71"), ("Pittsburgh", 185, "70")],
        "Detroit": [("Cincinnati", 263, "75"), ("Chicago", 283, "94"), ("Mississauga",
218, "401")],
        "Cleveland":[("Chicago", 344, "80"), ("Columbus", 143, "71"), ("Youngstown",
75, "80"), ("Buffalo", 194, "90")],
        "Youngstown":[("Pittsburgh", 67, "76")],
        "Indianapolis": [("Columbus", 175, "70"), ("Cincinnati", 112, "74"), ("St.
Louis", 242, "70"), ("Chicago", 183, "65"), ("Louisville", 114, "65"), ("Mississauga",
498, "401")],
```

```
        "Pittsburg": [("Columbus", 185, "70"), ("Youngstown", 67, "76"),
("Philadelphia", 304, "76"), ("New York", 391, "76"), ("Bedford", 107, "76")],
        "Bedford": [("Pittsburg", 107, "76")], #COMEBACK
        "Chicago": [("Indianapolis", 182, "65"), ("St. Louis", 297, "55"),
("Milwaukee", 92, "94"), ("Detroit", 282, "94"), ("Cleveland", 344, "90")],
        "New York": [("Philadelphia", 95, "95"), ("Albany", 156, "87"), ("Scranton",
121, "80"), ("Providence,", 95, "181"), ("Pittsburgh", 389, "76")],
        "Scranton": [("Syracuse", 130, "81")],
        "Philadelphia": [("Washington", 139, "95"), ("Pittsburgh", 305, "76"),
("Baltimore", 101, "95"), ("New York", 95, "95")]
    }

    args = parse_args(sys.argv[1:])
    main(args.starting_city, args.destination_city, connections)
```

Running your program

Your program is designed to run from the terminal. To run it, open a terminal and ensure you are in the directory where your script is saved.

The program takes at two command-line argument: the name of the starting city(string) and the name of the destination city(string). Below are some examples of how to use the program. The examples assume you are using a Unix based OS (macOS, Linux) and your program is called `gps.py`. If you are using Windows, replace `python3` with `python`.

Example 1

Input:

```
python3 gps.py --starting_city Baltimore --destination_city 'Daytona Beach'
```

Output:

```
Starting at Baltimore
Drive 39 miles on 95 towards Washington, then
Drive 53 miles on 95 towards Fredericksburg, then
Drive 60 miles on 95 towards Richmond, then
Drive 151 miles on 85 towards Durham, then
Drive 54 miles on 40 towards Greensboro, then
Drive 92 miles on 85 towards Charlotte, then
Drive 245 miles on 85 towards Atlanta, then
Drive 346 miles on 75 towards Jacksonville, then
Drive 86 miles on 95 towards Daytona Beach, then
You will arrive at your destination
```

Example 2

Input:

```
python3 gps.py --starting_city Washington --destination_city Richmond
```

Output:

```
Starting at Washington
Drive 53 miles on 95 towards Fredericksburg, then
Drive 60 miles on 95 towards Richmond, then
You will arrive at your destination
```