ECS 32B – Introduction to Data Structures
Final Project

Due: Friday, March 19, 2020, 8:30pm PT

Overview

The final project is to simulate a delivery truck. Each truck starts out empty but with a map that has the addresses of the post service offices and of the possible places it might need to deliver to. The truck is supposed to pick up all packages from all of the post service offices and deliver all the pakcages (but not necessarilty in that order). The truck also knows what packages there are in each post service office.

Each package has an address and a name to which the package should be delivered. The goal is to pick up and deliver all packages. By the end of Part 2, the fastest groups will gain extra credit.

The final project is broken into two parts. Part 1 is to set up the basic data structures. The description of Part 1 is in Section . Part 2 is to find the fastest route of delivering all packages given a map and the pakcages' information. The details of Part 2 will come later.

Part 2 also has an extra credit opportunity. The groups that can finish delivering the pakcages early will earn some extra credit, depending on your ranking (not cumulative):

- Top 2: 2%
- Top 5: 1.5%
- Top 10: 1%
- Top 15: 0.5%

The exact measure of performance will come later in Part 2's description. The basic operations that would probably be calculated towards the final running time include

- picking up (or adding) a package;
- delivering a pakcage (including finding the package in the truck first);
- calculating which route to take;
- driving from one location to the next.

Part 2 is supposed to be more time consuming than Part 1 and you might also make changes in Part 1 to make Part 2 more efficient. So Part 1 is worth 7% and Part 2 is 15%. (The final project as a whole is 22%.)

Part 1

The template for Part 1 is provided in "part1.py". It contains two classes: `Package` and `Truck`. You are to finish defining the `Truck` class.

Package

The `Package` class has only one method, the constructor. Each package is initialized with a unique identifier `id` and has four attributes:

- `id` (a string with only numeric characters) for the package's unique identifier.
- `address` (a string) for the address to which the package is supposed to be delivered.
- `office` (a string) for the address of a post service office: if a package is not collected, then the attribute refers to the office where the package is; otherwise, it refers to from which office the package is collected.
- `ownerName` (a string) for the name to which the package is supposed to be delivered, and
- `delivered` (a Boolean) for indicating if the package is delivered or not.
- `collected` (a Boolean) for indicating if the package is collected by a truck or not.

You are free to define accessors and mutators, or you may change the attributes directly. The `Package` class will not be graded.

Truck

The focus of Part 1 is to finish defining the `Truck` class. The template contains its basic attributes and methods. You are free to add more.

In the template, a truck has three attributes:

- `id` (a string with only numeric characters): the unique identifier of a truck.
- `size` (a non-negative integer): the size of the truck, i.e., the maximum number of packages can be stored in a truck (assuming all packages are of the same volume).
- `location` (a string): the current location of a truck.
- `packages`: the packages that have been loaded in a truck.

You get to decide the type of the `packages` attribute. It can be a Python list or any of the data structures we have covered (or will cover) in this class.

The methods you need to define include the followings.

- `__init__(self, id, n, loc)`: Each truck is initialized with a unique identifier `id`, a size `n` and a location `loc`. No matter what type you choose for the collection of packages in a truck, it should be initalized to be empty.
- `collectPackage(self, pk)`: collect a package `pk` (an instance of `Package`), i.e., add to `pk` to `self.packages`. The truck and `pk` must be at the same post service office for the truck to be able to collect the package.

- `deliverOnePackage(self, pk)`: deliver one pckage `pk`.
  The truck has to be at the address specified by `pk.address` to be able to deliver the package.
- `deliverPackages(self)`: deliver all packages that are supposed to be delivered to truck's current location.
- `removePackage(self, pk)`: remove a package `pk` from the truck and put it back to a post service office without delivering it. The truck must be at a post-service office for this to work and the removed package will be at the post-service office afterwards.
- `driveTo(self, loc)`: drive from current location to location `loc`.
- `getPackagesIds(self)`: return a Python list of the ids of the packages in `self.packages`. This is for testing. The order of the ids does not matter.

Note

- Do not change the names of the attributes and methods in either class.
- Feel free to add more methods and attributes if needed.

Part 2

Part 2 has two tasks. The first task is to implement some standard graph algorithms with some modifications. The second part is more complicated and the winners get extra credits. **Each group must submit only one version and list their members in the submission. Gradescope lets you add your members.**

In both tasks, we say a map is an **undirected and weighted graph** G, represented as a Python list of tuples for the edges and their weights in the graph. Each tuple (u, v, w) in G means there is an edge from vertex u to vertex v such that the weight of the edge is w. Since the graph is undirected, you also need to consider the edge from v to u with weight w.

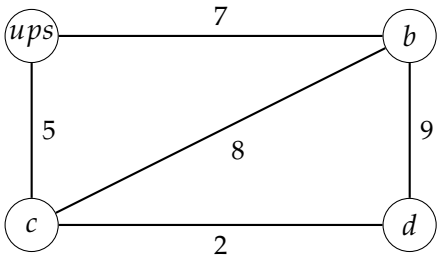
Given a path $[v_1, v_2, ..., v_n]$ from location v_1 to location v_n, we say the **weight of the path** or the **distance from v_1 to v_n taking the path** is the sum of the weight of all the edges on the path.

(Note that this is not the standard representation of a weighted graph. You might find converting the input graph into a adjacency list of adjacency matrix helpful.)

Assumptions:

- The map is connected, i.e., there is at least one path between any two locations.
- The weigth for every edge is non-negative.

Example: The following graph is represented as `[(ups,b,7), (ups,c,5), (b,c,8), (b,d,9), (c,d,2)]`. The path from *ups* to *b* then to *d* is `[ups, b, d]`.



Task 1 (75 points)

Download the part2Task1.py file.

Given a map on which exaclty one of the locations is a post service office, implement **BFS**, **DFS** and the **Dijkstra's** algorithms with the post service office being the source. The implement of each algorithm is worth 25 points.

Specifically, write three functions `bfs`, `dfs`, and `dijkstra`. The input for each of the functions includes

- the map `G`;
- the post service office `office`.

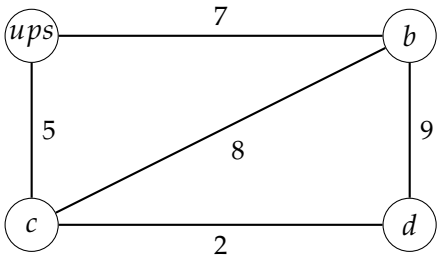
The output is a Python dictionary called `path` with keys being locations in G such that the value `path[loc]` for *loc* is the path from `office` to `loc` computed from the corresponding algorithm. Also, `path[office] = [office]`.

- For BFS and DFS, the path from `office` to each location can be reconstructed from the order of the locations visited.
- For Dijkstra's algorithm, the path from `office` to each location `loc` is the shortest path from `office` to `loc`.

Note:

- When there are more than one locations to choose from, choose them alphabetically.
- For this task, we do not care about the packages. All we need to do is traverse all the locations. You do not need the `Package` or `Truck` class.

Example:
Given the following map with *ups* being the only post service office,



- BFS: The first level only contains *ups*. The second level contains *b* and *c* with predecessor *ups*. The third level contains *d* with predecessor *b*, since *b* is enqueued before *c*. Therefore, the output is

```
path = {ups: [ups],
    b: [ups, b],
```

```
    c: [ups, c],
    d: [ups, b, d]}
```

- DFS: The output is

```
path = {ups: [ups],
    b: [ups, c, d, b],
    c: [ups, c],
    d: [ups, c, d]}
```

- Dijkstra's: Starting from *ups*, the shortest path we take from *ups* to each of the other locations is computed as:

```
path = {ups: [ups],
    b: [ups, b],
    c: [ups, c],
    d: [ups, c, d]}
```

Task 2 (25 points)

Download the part2Task2.py.

Given a map with multiple post service offices, deliver all packages in all of the offices to their correct addresses.

Specifically, write a function `deliveryService`. The function has three inputs:

- a map `G`;
- a truck `truck`;
- a Python list `packages` that includes all the packages involved, i.e., all the packages in all of the offices.

The output is a pair `(deliveredTo, stops)`, where

- `deliveredTo` is a Python dictionary with keys being package ids (NOT packages) and values being addresses, such that `deliveredTo[id] = addr` means the package with id `id` is delivered to `addr`. (This part of the output is used in testing if a package has been delivered to its correct address.)
- `stops` is a Python list that contains all the stops the truck made in order. For example, if the truck is initlized at location *addr*1, drives to *ups*1, then drives to *addr*2, then `stops = [addr1, ups1, addr2]`. (This part of the output is used in the extra credit part to compute the mileage the truck has driven.)

Restrictions:

- If the truck is at `loc1`, it can only drive to `loc2` if `loc1` and `loc2` are connected on the map.
- Do not change
  - anything in the input map;
  - the delivery address and id of a package,
  - the office of a package without using the `removePackage` function properly;
  - the location of a truck without using the `driveTo` function properly.
- Do not delete items from the `packages` parameter.

Extra Credit for Task 2

For those who implement Task 2 correctly, i.e., receive all 25 points of Task 2, there is an extra credit opportunity. The amount of extra credit you receive depends on your group's ranking (listed in the Overview section).

The ranking takes into count two factors:

1. **The actual running time of your code.** This part of the ranking is shown on Gradescope. The fastest group is ranked at the top. **Don't forget to add your group members to the submission and give your group a unique name.**
2. **The total mileage the truck drives.** Unfortunately, I couldn't figure out how to display the mileage as part of the ranking on Gradescope. TAs will run your programs locally to get the mileage using the second output of your deliveryService function.

Design Your Own Project

You may also design your own project. If you wish to do so, you must talk to me first about your proposed project to make sure it is accetable. If you did your own project without talking to me, you will not be graded properly on Gradescope and hence no points.

The project must also contain two parts:

1. The first part demonstrates the usage of at least one of the linear data structures we have learned in this class, including stack, queue, deque, linked list, and hash table.
2. The second part uses either trees or graphs (we haven't learned yet).

These two parts do not have to be related in any way. They are due at the same time on the same day as the rest of the class.

In addition, you will need to write a **report**. The report should summarize what the project is, what functionalities you want to achieve and what problem(s) to solve. If the two parts of the project are not related, you will need to write two reports, one for each part.

You will also need to write your own **test cases** for both parts. You can email me early on and I will create a separate submission locker for you on Gradescope so that you can test your code.

For Part 2, you can also add additional functionalities as **extra credit** (worth 1% toward the course grade).

If your group has more than three people, you need to state in your reports(s) the extra work handled by the additional teammates.