Assignment 1

An Experiment to Compare Algorithms for Parcel Delivery

Due date: Tuesday, March 2, 2021 before 1:00 pm sharp, Toronto time.

You may complete this assignment individually or with one partner.

Learning Goals

By the end of this assignment you be able to:

- read complex code you didn't write and understand its design and implementation, including:
  - reading the class and method docstrings carefully (including attributes, representation invariants, preconditions, etc.)
  - determining relationships between classes, by applying your knowledge of composition and inheritance
- complete a partial implementation of a class, including:
  - reading the representation invariants to enforce important facts about implementation decisions
  - reading the preconditions to factor in assumptions that they permit
  - writing the required methods according to their docstrings
  - use inheritance to define a subclass of an abstract parent class
- design a class, given its name and purpose, including
  - designing an appropriate interface
  - weighing some options and choosing an appropriate data structure to implement a class
  - recording important facts about an implementation decision using representation invariants
- make reasonable decisions about which classes should be responsible for what
- use an ADT to solve a problem, without having to think about how it is implemented
- perform unit testing on a program with many interacting classes

Please read this handout carefully and ask questions if there are any steps you do not understand.

The assignment involves code that is larger and more complex than Assignment 0. If you find it difficult to understand at first, that is normal – we are stretching you to do more challenging things. Expect that you will need to read carefully, and that you will need to go back over things multiple times.

We will guide you through a sequence of tasks, in order to gradually build the pieces of your implementation. Note that the pieces are laid out in logical order, not in order of difficulty.

Coding Guidelines

These guidelines are designed to help you write well-designed code that maintaints the interfaces we have defined (and thus will be able to pass our test cases).

For class `DistanceMap`, all attributes must be private.

For all other classes except `Parcel` and `Truck`, You must NOT:

- change the interface (parameters, parameter type annotations, or return types) to any of the methods you have been given in the starter code. Note: in the scheduler subclasses, you are permitted to change the interface for the inherited initializer, if needed.
- change the type annotations of any public or private attributes you have been given in the starter code.
- create any new public attributes.
- create any new public methods except to override the schedule method that your classes Greedy and Random inherit from their abstract parent.
- add any more import statements to your code.

You will be designing classes `Parcel` and `Truck`, so we make these exceptions:

- You will be designing the attributes for these classes `Parcel` and `Truck`, and are allowed to make some or all of them public. Use your judgment.
- You may add public methods to `Parcel` and `Truck`.

You may:

- remove unused imports from the Typing module. (We have included those that we think you might want to use in `DistanceMape`, `Parcel` and `Truck`.)
- create new private helper methods for the classes you have been given.
  - if you do create new private methods, you must provide type annotations for every parameter and return value. You must also write a full docstring for such methods, as described in the Function Design Recipe.
- create new private attributes for the classes you have been given.
  - if you do create new private attributes you must give them a type annotation and include a description of them in the class's docstring as described in the Class Design Recipe.
  - Exception: In class `PriorityQueue` we have defined all the attributes needed. Do not define any new attributes, even private.

You may assume that all arguments passed to a method or function will satisfy its preconditions.

All code that you write should follow the Function Design Recipe and the Class Design Recipe.

Introduction

Consider what happens when a delivery company like FedEx or Purolator receives a plane load of parcels at Pearson Airport to be delivered by truck to cities all over southern Ontario. They have to schedule the parcels for delivery by assigning parcels to trucks and determining the route each truck will take to make its deliveries.

Depending on how well these decisions are made, trucks may be well-packed and have short, efficient routes, or trucks may not be fully filled and may have to travel unnecessary distances.

For this assignment, you will write code to try out different algorithms to perform parcel scheduling and compare their performance.

Problem description

Be sure to read through this section carefully. Your Python code must accurately model all of the details described here.

The parcel delivery domain

Each parcel has a *source* and a *destination*, which are the name of the city the parcel came from and the name of the city where it must be delivered to, respectively. Each parcel also has a *volume*, which is a positive integer, measured in units of cubic centimetres (cc).

Each truck can store multiple parcels, but has a *volume capacity*, which is also a positive integer and is in units of cc. The sum of the volumes of the parcels on a truck cannot exceed its volume capacity. Each truck also has a *route*, which is an ordered list of city names that it is scheduled to travel through.

Each parcel has a unique ID, that is, no two parcels can have the same ID. Each truck also has a unique ID, that is, no two trucks can have the same ID.

Depot

There is a special city that all parcels and trucks start from, and all trucks return to at the end of their route. We'll refer to this city as the *depot*. (You can imagine all parcels have been shipped from their source city to the depot.) Our algorithms will schedule delivery of parcels from the depot to their destinations.

You may assume that no parcels have the depot as their destination.

There is only one depot.

Truck routes

All trucks are initially empty and only have the depot on their route (since it is their initial location). A truck's route is determined as follows: When a parcel is scheduled to be delivered by a truck, that parcel's destination is added to the end of the truck's route, unless that city is already the last destination on the truck's route.

Example: Consider a truck at the start of the simulation, and suppose that the depot is Toronto. The truck's route at that point is just Toronto. Now suppose parcels are packed onto that truck in this order:

- a parcel going to Windsor. The truck's route is now Toronto, Windsor.
- another parcel going to Windsor. The truck's route is unchanged.
- a parcel going to London. The truck's route is now Toronto, Windsor, London.
- a parcel going to Windsor. The truck's route becomes Toronto, Windsor, London, Windsor. (Yes, this is a silly route, so the order in which we pack parcels onto trucks is going to matter.)

Whatever its route, at the end, a truck must return directly to the depot.

Scheduling parcels

You will implement two different algorithms for choosing which parcels go onto which trucks, and in what order. As we saw above, this will determine the routes of all the trucks.

(1) Random algorithm

The *random algorithm* will go through the parcels in random order. For each parcel, it will schedule it onto a randomly chosen truck (from among those trucks that have capacity to add that parcel). Because of this randomness, each time you run your random algorithm on a given problem, it may generate a different solution.

(2) Greedy algorithm

The *greedy algorithm* tries to be more strategic. Like the random algorithm, it processes parcels one at a time, picking a truck for each, but it tries to pick the "best" truck it can for each parcel. Our greedy algorithm is quite short-sighted: it makes each choice without looking ahead to possible consequences of the choice (that's why we call it "greedy").

The greedy algorithm has two configurable features: the order in which parcels are considered, and how a truck is chosen for each parcel. These are described below.

Parcel order

There are four possible orders that the algorithm could use to process the parcels:

- In order by parcel volume, either smallest to largest (non-decreasing) or largest to smallest (non-increasing).

- In order by parcel destination, either smallest to largest (non-decreasing) or largest to smallest (non-increasing). Since destinations are strings, larger and smaller is determined by comparing strings (city names) alphabetically.

Ties are broken using the order in which the parcels are read from our data file (see below).

Truck choice

When the greedy algorithm processes a parcel, it must choose which truck to assign it to. The algorithm first does the following to compute the *eligible trucks*:

1. It only considers trucks that have enough unused volume to add the parcel.
2. Among these trucks, if there are any that already have the parcel's destination at the end of their route, only those trucks are considered. Otherwise, all trucks that have enough unused volume are considered.

Given the eligible trucks, the algorithm can be configured one of two ways to make a choice:

- choose the eligible truck with the most available space, or
- choose the eligible truck with the least available space

Ties are broken using the order in which the trucks are read from our data file. If there are no eligible trucks, then the parcel is not scheduled onto any truck.

Observations about the Greedy Algorithm

Since there are four options for parcel priority and two options for truck choice, our greedy algorithm can be configured eight different ways in total.

Notice that there is no randomness in the greedy algorithm; it is completely "deterministic". This means that no matter how many times you run your greedy algorithm on a given problem, it will always generate the same solution.

Putting it all together

For this assignment, your program will create an experiment by reading in some parcel, truck, and configuration data from a set of files. Your code will be able to apply the random algorithm or any of the variations of the greedy algorithm to a scheduling problem, and then report on statistics of interest, such as the average distance travelled by the trucks. This would allow the delivery company to compare the algorithms and decide which is best, given the company's priorities.

Starter code

Download **this zip file (https://q.utoronto.ca/courses/204422/files/12593206/download)** ↓ **(https://q.utoronto.ca/courses/204422/files/12593206/download?download_frd=1)** containing the starter

files for this assignment. Extract its contents into `assignments/a1` in the csc148 folder that you set up by following the **Software Guide (https://q.utoronto.ca/courses/204422/pages/software-guide)** .

The module that drives the whole program is `experiment`. It contains class `SchedulingExperiment`.

A `SchedulingExperiment` can be created based on a dictionary that specifies the location of necessary data files as well as an algorithm configuration. Then the experiment can be run and statistics can be generated (and optionally reported).

The `experiment` module contains a main block that sets up and runs a single experiment. It can be used as a quick check to see if your experiment code can run without errors; it does not check the correctness of the resuts. For testing the correctness of your code, you should add unit tests to `a1_test.py` and run that.

An experiment can be run in verbose mode. In that case, it should print step-by-step details regarding the scheduling algorithm as it runs. You may find this helpful for debugging. The specific verbose output doesn't matter, as we will never test your code in verbose mode.

You are free to change the name of the configuration file that is read in the main block of module `experiment`, because we will not test your code by *running* module `experiment`. Our tests will *import* the `experiment` module and use class `SchedulingExperiment`.

Format of the Configuration Dictionary

The configuration dictionary stores the configuration of the scheduling experiment as a set of key-value pairs:

- depot_location: the name of the city where each parcels is, and from which it must be delivered to its destination. Also the city where all trucks start at and return to.
- parcel_file: the name of a file containing parcel data.
- truck_file: the name of a file containing truck data.
- map_file: the name of a file containing distance data.
- algorithm: either 'random' or 'greedy'. If 'random', none of the remaining keys are required in the configuration file (and if they are present, they will be ignored).
- parcel_priority: either 'volume' or 'destination'.
- parcel_order: either
  - 'non-decreasing' (meaning we process parcels in order from smallest to largest), or
  - 'non-increasing' (meaning we process parcels in order from largest to smallest).
- truck_order: either
  - 'non-decreasing' (meaning we choose the eligible truck with the least available space, and as go through the parcels we will choose trucks with greater available space), or
  - 'non-increasing' (meaning we choose the eligible truck with the most available space, and as we go through the parcels we will choose trucks with less available space).

- verbose: either true or false. Note the lack of quote marks; the code we've written to read in the configuration dictionary can read such values directly into a boolean.

Format of the Data Files

The parcel file has one parcel per line, with the format:

```
<parcel_id>, <source>, <destination>, <parcel_volume>
```

Parcel IDs may occur in any order and need not be consecutive, but no parcel ID occurs more than once in the file.

The truck file has one truck per line, with the format:

```
<truck_id>, <truck_volume>
```

Truck IDs may occur in any order and need not be consecutive, but no truck ID occurs more than once in the file.

The map file has one distance fact per line, with the format:

```
<city1>, <city2>, <distance1> [, <distance2> ]
```

The distance from city1 to city2 is distance1, and the distance from city2 to city1 is distance2. (The two might differ if there is unusual geography, one-way roads, or construction.) The square brackets indicate that distance2 can be omitted to indicate that the distance is the same going from city2 to city1 as it is going from city1 to city2. All distances are integers. There is no duplicate information or contradictory information in the file.

The starter files include an example of each of these data files, as well as a program called `generator.py` that can generate new, random, parcel and truck files. You may find this helpful for testing your code. Note the places in `generator.py` that allow you to control what it generates.

Assuming valid inputs

The functions that read from data files or configuration files all have strong preconditions allowing the code to assume inputs are valid and exactly as described in the handout.

Each method that we have specified states whatever preconditions you may assume hold. The docstrings should make clear what your code must do. Notice that there is little to no focus on validating data.

Tasks

The next part of this handout guides you through the code as you work on the different parts of this assignment.

Task 0: Prepare

Start the assignment by doing the following:

1. Read this handout to learn about the problem domain and what you will do for the assignment.
2. Do the **Assignment 1 quiz (https://q.utoronto.ca/courses/204422/quizzes/152382)** on Quercus to get your head into the starter code.
3. Ensure that you have a precise understanding of the random and greedy algorithms described below. To assist you, we have created a detailed example of the greedy algorithm in action: **greedyscheduler_example.html.pdf (https://q.utoronto.ca/courses/204422/files/12431252?wrap=1)** ↓ **(https://q.utoronto.ca/courses/204422/files/12431252/download?download_frd=1)**

Task 1: Distance Map

Requires:

- knowing how to design and implement a class (week 2).

In file `distance_map.py`, use the Class Design Recipe to define a class called `DistanceMap` that lets client code store and look up the distance between any two cities. Your program will use an instance of this class to store the information from a map data file.

If a distance from city A to city B has been stored in your distance map, then it must be capable of reporting the distance from city A to city B and of reporting the distance from city B to city A (which could be the same or different).

Your class *must* provide a method called `distance` that takes two strings (the names of two cities) and returns an int which is the distance from the first city to the second, or -1 if the distance is not stored in the distance map. The doctest examples in class `Fleet` depend on there also being a method called `add_distance`. Make sure that it exists and is consistent with those doctests.

The choice of data structure is up to you. For something so simple, there are surprisingly many choices. Just pick something reasonable, and be sure to define representation invariants that document any important facts about your data structure.

You may find it helpful to use a default parameter somewhere in this class. Here is an example of how they work. This function takes two parameters, but if the caller sends only one argument, it uses the default value 1 for the second argument.

```
def increase_items(lst: List[int], n: int=1) -> None:
    """Mutate lst by adding n to each item.

    >>> grades = [80, 76, 88]
    >>> increase_items(grades, 5)
    >>> grades == [85, 81, 93]
    True
    >>> increase_items(grades)
    >>> grades == [86, 82, 94]
```

```
    True
    """
    for i in range(len(lst)):
        lst[i] += n
```

Task 2: Modelling the Domain

Requires:

- knowing how to design and implement a class (week 2).
- knowing how to identify appropriate classes from a problem description (week 2).

Your next task is, in file `domain.py` to define the classes necessary to represent the entities in the experiment: classes `Parcel`, `Truck`, and `Fleet`.

We have designed the interface for class `Fleet`. As you might imagine, it keeps track of its trucks, but it also offers methods that report statistics about things such as how full its trucks are, on average. (These statistics are used when we create and run instances of the `Experiment` class.)

Class `Fleet` is a client of classes `Parcel` and `Truck`, so use class `Fleet` to figure out what services it will need from classes `Parcel` and `Truck`. (A little bit of that is already dictated by the doctest examples in class `Fleet`.) Then use the Class Design Recipe to help you design classes `Parcel` and `Truck` to provide those services.

Start each of these two classes simply, by focusing on the data you know it must store and any operations that you are certain it must provide. Very likely, as you start to use these classes in later steps, you will add new operations or make other changes. This is appropriate and a natural part of the design process.

Once you have classes `Parcel` and `Truck` completed and well tested, you can move on to implement class `Fleet`.

Remember to do all of your work for this task in the file `domain.py`.

Task 3: Priority Queue

Requires:

- knowing how to implement a class (week 2).
- understanding inheritance (week 3).

In the greedy scheduling algorithm, you will need to consider the parcels one at a time. While we could use a Python list, it has no *efficient* way to give us items in order according to our priority (e.g. by volume in non-decreasing order).

Instead we will use something called a **PriorityQueue**. Like a list, it is also a kind of container that allows you to add and remove items. However, a priority queue removes items in a very specific

order: it always removes the item with the highest "priority". If there is a tie for highest priority, it chooses among the tied items the one that was inserted first. (See the docstrings in class `PriorityQueue` for examples.)

A priority queue can prioritize its entries in different ways. For example, we might want to make a priority queue of strings in which shorter strings have higher priority. Or we might want to make a priority queue of employees in which employees with the least seniority have highest priority. Our `PrirorityQueue` class lets client code define what the priority is to be by passing to the initializer a function that can compare two items. Cool! See **section 1.4 of the Lecture Notes (https://www.teach.cs.toronto.edu/~csc148h/winter/notes/python-recap/type_annotations.html)** for information about the type annotation `Callable`, which is used in that class.

A PriorityQueue supports the following actions:

- determine whether the priority queue is empty
- insert an item with a given priority
- remove the item with the highest priority

File `container.py` contains the general `Container` class, as well as a partially-complete `PriorityQueue` class. `PriorityQueue` is a child class of `Container`.

You must complete the `add()` method according to its docstring. It is just one method, but you will have to read carefully and think clearly to complete it.

(Notice that we're using a sorted list to implement this class; in later courses, you'll learn about a much more efficient implementation called **heaps (https://en.wikipedia.org/wiki/Heap_%28data_structure%29)**.)

Task 4: The Scheduling Algorithms

Requires: same as Task 3, and also a solid understanding of the scheduling algorithms described above.

In the file `scheduler.py`, we have provided you with a class called **Scheduler**. It is an abstract class: the `schedule` method is not implemented, so no instances of the class should be made. Its purpose is to define what *any* sort of scheduler must be able to do.

Your task is to implement two subclasses of Scheduler called **RandomScheduler** and **GreedyScheduler**, which implement the scheduling algorithms defined above. This is the heart of the code.

Your code in this module will make use of the various classes you have completed in the earlier steps. For example, the `schedule` method takes a list of `Parcel` objects and a list of `Truck` objects.

You'll find use for class `PriorityQueue` in your `GreedyScheduler`, since it needs to get the parcels into the desired order (as specified in the configuration dictionary). A priority queue is perfect for this.

When you create an instance of `PriorityQueue`, it needs to be told what function you want it to use to dictate which of two things has higher priority. You will need to define a (very simple) function for each of the four kinds of parcel priority you may have to deal with, described above. Define these four functions at the module level (outside of any class) and name each with a leading underscore to indicate that it is private. When your greedy schedule needs to put the parcels in order, it can create a priority queue that uses the appropriate one of these priority functions. (It will look at the configuration dictionary to determine which.)

You may need to add one or two attributes to your scheduler classes. If you do, make them private. You may also need to define an initializer for one of your child scheduler classes that has a different parameter list than the one it inherits. This is permitted. Just be sure not to change the interface of any other methods in the starter code.

Task 5: Experiment

Requires: same as Task 3

You are ready to complete the code that runs the whole experiment! Take a look at the file `experiment.py` and complete the code.

When calculating the statistics, you can assume there is always at least one parcel and one truck, and that at least one truck will be used. You may also assume that the map file contains every distance that you will need in order to compute the statistics.

Extra modules we are providing

There are several modules that we are providing for you:

- `explore`: Compares all algorithms on a single problem. Now that you have all the code written, try running this and see how the algorithms measure up against each other.
- `a1_test`: Contains a sample configuration dictionary from which it generates 6 pytest test cases – one for each statistic that your code should generate. (Yes, this code creates pytest code. Very cool.) You will not hand in this module, but it is the place where you should put your tests of the overall `SchedulingExperiment` class. To add a new test case, just follow the pattern that we've shown.
- `generator`: Generates random parcel and truck data and writes each to a file.

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the Python style guidelines that PyCharm points out. Fix them!

- In each module, run the provided `python_ta.check_all()` code to check for errors. Fix them! PythonTA is also included in the tests we provided on MarkUs. The full feedback will be in a file added to your submission called pyta_feedback.txt.
- Make sure that the code you wrote follows the conventions of the Function Design Recipe and the Class Design Recipe, including conventions relating to function, method, and class docstrings.
- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as calls to the `print` function. (Better yet, use `print` only when in verbose mode.)
- Remove any `pass` statement where you have added the necessary code.
- Remove the word “TODO” wherever/whenever you have completed the task.
- Take pride in your gorgeous code!

Submission instructions

Submit the following files on MarkUs: `container.py`, `distance_map.py`, `domain.py`, `experiment.py` and `scheduler.py`. We strongly recommend that you submit early and often. We will grade the latest version you submit within the permitted submission period.

Be sure to run the tests we’ve provided within MarkUs one last time before the due date. This will make sure that you didn’t accidentally submit the wrong version of your code, or worse yet, the starter code!

How your assignment will be marked

There will be no marks associated with defining your own test cases with pytest or hypothesis.

The marking scheme will be approximately as follows:

- pyTA: 10 marks, with 1 mark deducted for each occurrence of each pyTA error.
- following the coding guidelines above: 5
- self-tests, provided in MarkUs: 20
- `Parcel` and `Truck` classes: will not be tested directly, other than for aspects that are proscribed by the doctest examples in class `Fleet`
- hidden tests: 65