COSC122 2018 Semester 2
INTRODUCTION TO ALGORITHMS

Assignment 3: Queueing Cars

7% of overall grade
Due Friday 19 October 2018, 11:55pm

1 Overview

1.1 Introduction

Thanks to advances in computer and network technologies we live in an age of increasingly pervasive surveillance. As with most things, there are pros and cons to such widespread surveillance but we will leave that philosophical discussion to your Philosophy courses and/or COSC101. This assignment isn't going to take sides in the debate—our aim is simply to give you some insight into how data from such surveillance systems might be used efficiently to meet whatever good/evil ends that one might have for the data.

1.2 Due date

Your assignment code should be submitted by Friday 19 October 2018, 11:55pm. The drop dead date is 7 days later, and any late assignments will incur a 15% absolute penalty. That is your mark will be the raw mark less 15 marks (given the assignment is marked out of 100).

1.3 Submission

Submit via the quiz server. The submission page will be open closer to the due date.

1.4 Implementation

All the files you need for this assignment are on the quiz server. You must not import any additional standard libraries unless explicitly given permission within the task outline. For each section there is a file with a function for you to fill in. You need to complete these functions, but you can also write other functions that these functions call if you wish. All submitted code needs to pass PYLINT program checking (it will make you proud of your stylish code).

1.5 Getting help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work. If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues, but **never** post your own program code to Learn (or make it available publicly on the internet via sites like GitHub)[1]. Remember that the main point of this assignment is for you to exercise what you have learned from lectures and labs, so make sure you do the relevant labs first, so don't cheat yourself of the learning by having someone else write material for you.

2 Merging is Heaps of fun

2.1 General Overview

This assignment consists of two main parts. The first is to find the items in a list that are unique to that list and the second is the implementation of a priority queue using a d-heap. A general overview is given here. It is followed by a description of the provided tests and files and then the detailed description of each task.

- The first task, as a warm up, is to implement a function that will return the list of number plates that are unique to a given list. The thin veil of utility for this scenario is that the Police want a list of cars that were sighted at one location but not at another—for some important investigation.
- Then the focus switches to an application for heaps and the rest of the tasks will work through the implementation of a priority queue (with a min-d-heap). The priority queue will store Cars based on their `Priority`. The `Priority` for a car is a function of its distance from the Police Station (or donut shop) and the dodginess of the car's owner. Police want to visit the closest/dodgiest cars first so you will be implementing the priority queue using a min-d-heap.

3 Supplied Code and Data

3.1 Provided Skeleton Code

The two modules you will need to complete are `unique.py` and `car_queue.py`. A brief overview of those modules is given here and more detail is provided in the task descriptions in section 4.

3.1.1 The `unique.py` module

You will need to complete the `unique_plates` function in this module as per the task description. This module also contains some examples of testing code to get

[1] Check out section 3 of UC's *Academic Integrity Guidance* document. Making your code available to the whole class is certainly not discouraging others from copying.

your started with your testing. The tests for this module will be provided in the `test_unique` module as outlined below.

Note: the `unique_plates` function will be dealing with lists of `NumberPlate` objects.

3.1.2 The `car_queue.py` module

This module contains skeletons of the `CarHeapQeueue` and `EditableCarHeapQeueue`. We have provided the `__init__` methods and various other basics such as `__str__` and `__repr__`, etc... You will need to complete various methods in the `CarHeapQeueue` and `EditableCarHeapQeueue` classes module as per the task descriptions. This module also contains some examples of testing code to get your started with your testing. The tests for this module will be provided in the `test_car_queue` module as outlined below.

Notes:

- The heaps need to be min-d-heaps, where d is the maximum number of children that a node can have (we will generally refer to d as the number of children or `num_children`) and the number of children can be specified when a heap/queue is initialised—see the `__init__` method definition.
- The first/min node in the heap should be stored at index 0. This is different from the labs but it will make dealing with larger numbers of children a lot easier. Make sure you draw out various d-heaps and work out the general formula for finding the parent/children of a given node. You should make sure you test your queues/heaps with various numbers of children, including 1 child—which makes the queue rather inefficient but is an interesting case in that it performs in a linear fashion.

3.2 Other Modules and Classes

3.2.1 The `classes3.py` module

Continuing the famous series of useful assignment classes, `classes3.py` contains the definitions of the data classes you will be using for this assignment.

The very popular `NumberPlate` class returns for the third edition of the assignments and you should be familiar with its ability to act like a `str` while keeping track of comparisons in the background.

Making their first appearance in an assignment are the `Priority` and `Car` classes. They are described below.

`Priority` : For now, you may consider a `Priority` to be an `int`, in that all the common comparisons (`==`, `<`, `>`, `<=`, `>=`, `!=`) are available (eg, if `a` and `b` are `Priority`'s then expressions like `a < b` and `a == b`, etc, will work as expected). The difference is that we keep track of these comparisons in the background, via the `StatCounter` class.

`Car` : `Car` objects are designed to record information about cars. That is, they contain the following instance variables, `plate`, `priority`, `location`, `distance_to_station`, and `dodgy_factor`. The distance to the Police station is calculated when a `Car` object is initialised and is, for simplicity, just the Cartesian distance from the location to the Police Station (which is assumed to be at grid location (0,0)). The priority for a car is also calculated at initialisation and is a function of the

distance and the dodgyness. The closer the `Car` is to the Police station or the more dodgy the owner, the lower the Priority score. You can think of low priority scores being closer or more important.

You should read through the class definitions in `classes3.py` to make sure you fully understand the objects you will be using for this assignment.

In the `unique_plates` function you should be counting how many `NumberPlate` comparisons are made and returning the value. NOTE: In the heap questions you will be counting the number of `Priority` comparisons—make sure you remember to switch focus when you move on to those questions.

3.2.2 The `utilities.py` module

As in the previous assignments, the `utilities.py` module contains functions for reading test data from test files and making simple lists of data. Files containing test data are in the folder `test_data` to make it easier to test your own code.

The most useful functions in `utilities.py` are outlined below.

`read_unique_test_data` Returns three lists of `NumberPlates` reflecting the data in the file. The first two lists are the lists that are to be supplied to the `unique_plates` function for testing. The third list is list of `NumberPlates` that is expected to be returned by the `unique_plates` function. Check out the example in the module.

`plates_from_strings` Returns a list of `NumberPlates` based on the strings provided. The strings must be valid number plates character sequences. An example call would be `plates_from_strings(['AAA111', 'BBB111', 'CCC111'])`.

`read_heap_test_data` This function returns two lists. The first is a list of cars to initialise a queue with. The second is a list of (command, data) tuples as outlined in section 3.3.3 below. You can use these lists to do some simple testing yourself but you will probably want to use the `run_heap_tests` function to run full tests.

`verify_heapness` Given a heap this function will True if the heap is valid, otherwise it returns False.

`verify_indices` Given an `EditableCarHeapQueue` this function will check that the indices for each number plate (stored in the `_indices` dictionary) point to the correct indices in the heap's data list. That is, the index for a given plate does actually point to the node that contains the `Car` with the given plate value.

`run_heap_tests` Takes a `filename`, `priority_queue` type and `number_of_children` as it's main parameters. It then loads the data from the file, creates a queue of the given type with the given number of children and the initial data from the file. Then it runs through all the commands in the file in sequence—checking that the queue remains a valid heap throughout and that the expected values are returned when dequeuing etc... This is very similar to the function that is used in the test file, it just has more output for you to read.

3.2.3 The `stats.py` module

The `stats.py` module contains another class that you may find useful: `StatCounter`. This class holds all the comparison information that you might need to check your code against. We also use it to verify that you do perform the correct number of comparisons.

`StatCounter` : In order to count how many comparisons your code makes, we provide a `StatCounter` class. Note that because of the way it is implemented, it will not behave like a regular class. This class is only for testing purposes, and not in your final code. You should only use the `StatCounter.get_count(counter_name)` method for testing and should remove all calls to it from the code you submit to the Quiz Server. You will, of course, need to remove any imports related to `stats` from your code—otherwise *pylint* might get angry with an unused import.

> ⚠ **Important:** You cannot rely on `StatCounter` (or more specifically, the methods and instance variables in `StatCounter`) being available to you on the quiz server at the time of submission, so do not use it in your final code.

3.3 Testing your code

Off-line tests are available in the `test_unique.py` and `test_car_queue.py` modules and you should use them to check your code before submission (the submission quiz won't fully mark your code until submissions close). The provided tests use the Python `unittest` framework. You are not expected to know how these tests work; you just need to know that the tests will check that your code finds the list of stolen number plates that were sighted, and that the number of comparisons that your code makes is appropriate.

You should also be doing your own testing, eg, of various edge cases such as when an input list is empty or when no number plates are sighted etc...

3.3.1 Provided tests

The test modules provide a number of tests to perform on your code. You can adjust which tests you are running by commenting/uncommenting the appropriate lines in the `all_tests_suite` function (much the same as in the previous assignments). Each test has a name indicating what test data it is using, what it is testing for, and a contained class indicating which algorithm is being tested. In the case of a test case failing, the test case will print which assertion failed or which exception was raised.

As with previous assignments, these tests aren't going to be very helpful for debugging your code. You will want to write your own tests using small amounts of data. This will let you get a feel for how your classes are storing data and whether or not they are operating as expected (because you will be able to work out how they should be working by hand).

> ⚠ **Important:** In addition to the provided tests you are expected to do your own testing of your code. This should include testing the trivial cases such as empty lists and lists of differing sizes.

3.3.2 Test data for *unique*

The files provided for testing the `unique_plates` function contain three lists of number plate strings. The first two are the two lists that we want to compare to find the unique list of plates that are in the second list. The third list is the expected list of unique plates, ie, to compare with what your function actually returns. You can assume that neither input list has any repeated plates within itself.

The files are named in the form `unique-n1-n2-nexp-seed.txt` where:

- `n1` is the number of plates in the first list.
- `n2` is the number of plates in the second list.
- `nexp` is the number of unique plates in the second list. That is the number of plates in the second list that don't appear in the first list.
- `seed` is the random seed used when generating the data. You don't need to worry about this.

See section 3.2.2 for more information on how to use the `read_unique_test_data` function to read data from the test files.

3.3.3 Test data for Car queues

The files provided for testing the `car_queue` classes contain:

- An initial list of cars to load into a queue/heap when initialising the queue/heap.
- A list of (command, data) pairs that are to be carried out in the order they appear in the file.

The following commands appear in the files:

enqueue is followed by the details for a car to be enqueued.

dequeue is followed by the plate of a car that is expected after the dequeue.

update is followed by the car to update.

remove is followed by the plate of the car to remove from the queue.

The files are named in the format: `n_imports-n_enq-n_deq-n_upd-n_rem-seed.txt` where:

- `n_imports` is the number of cars in the list to import initially
- `n_enq` is the number of *enqueue* instructions in the command list.
- `n_deq` is the number of *dequeue* instructions in the command list.
- `n_upd` is the number of *update* instructions in the command list.
- `n_upd` is the number of *remove* instructions in the command list.
- `seed` is the random seed used when generating the data. You don't need to worry about this.

For example, a file named `1000-50-1000-20-0-a.txt` would have a list of 1000 cars to initialise the queue with. It would then have a instruction list that contained a random mix of instructions. There would be 50 enqueues, 1000 dequeues, 20 updates and 0 removes.

You can use the `read_heap_test_data` function (in the `utilities` module to load heap test files. See section 3.2.2 for more information on this function. Alternatively you can use `run_heap_tests` function to load and run test files. `run_heap_tests` will create a queue using the import data and then run through the commands in sequence. For example:

```
>>> from utilities import *
>>> from car_queue import *
>>> heap = run_heap_tests('./test_data/3-2-0-0-0-a.txt', CarHeapQueue, 2)
Enqueueing Car('MYB544', (-6800, 15), 87, 884)
Enqueueing Car('NPD009', (-5058, -6143), 80, 1591)
```

You should open some of the test data files and make sure you understand the format—remember to open them in Wing or a suitably smart text editor. If you are using windows then try Notepad++ because the normal Notepad is pretty horrible [2]. If you are using Linux (or a Mac) then almost any text editor will do.

The test files are generated in a quasi-random fashion and the seed suffix indicates the random seed that was used when generating the data—you don't need to worry about this. But, you do need to worry about the fact that we can easily generate different random files to test with.

4 Tasks [100 Marks Total]

For task 1 you will be writing the `unique_plates` function in the `unique.py` module. In this function you will count the number of `NumberPlate` comparisons.

For tasks 2 through 4, you will be working in the `CarHeapQueue` class in the `car_heap.py` module. For task 5, you will be working in the `EditableCarHeapQueue` class in the `car_heap.py` module. Do not forget to increment the `self.comparisons` counter whenever your code compares two `Priority` objects.

4.1 Finding unique plates [20 Marks]

You need to complete the function `unique_plates(plate_list1, plate_list2)` in `unique.py`. The function takes two *sorted* lists of `NumberPlates` and returns a list containing `NumberPlates` that contains plates that only appear in `plate_list2`, in the same order as they appear in `plate_list2`, and an `int` for the number of `NumberPlate` comparisons that were made. That is, the function should return a tuple like `(result_list, comparisons)`.

The function should take into account that both lists are already in ascending order, that is, it should use a method that is similar to that of *mergesort* or the *common items* function that you wrote in Lab 6.2. This means, for example, you won't need to use any for loops.

You can test your function with the `test_unique` module. But you should, of course, write some smaller more informative tests yourself, to help with debugging.

Note, there are three basic variations for the main loop for the method you should be using for this function. Only one of the variations will match the expected counts in the simple test cases provided. It is up to you to figure out which variation it is. For

[2] There will soon be an update to Windoze 10 that finally gives Notepad the ability to deal with line endings properly, like every other text editor on the planet. Yay!

inspiration think about the two variations of binary search that you saw earlier in the course.

Note: For the remaining tasks you will be writing code in the the `car_queue.py` module and doing final testing with the `test_car_queue.py` module.

4.2 Enqueue and sifting up [20 Marks]

A heap has two fundamental operations: sift-up, and sift-down. For this task, you are to implement the sift-up operation, and use to it *enqueue* Cars. A sift-up operation proceeds by determining the parent index `p` of the sifted index `i`. If the data stored at `i` has a lower priority value than the data stored at `p`, then they need to be swapped (remember you are implementing a *min-heap*). Otherwise, no swap is needed and the sifting up is complete. Repeat this process until the data that started in position `i` is in the right place (this is called *restoring the heap invariant*), ie, doesn't need to swap with it's parent. Use this description to complete the method `_sift_up`.

Remember that the heap/queue initialiser has a parameter for the number of children the heap will have (ie, where the number of children is the *d* in *d-heap*). You can access the number of children (or d factor) via `self.num_children`.

Important Note: You should store the first item/node in the heap at index 0 in the `_data` list. This is different to what you did in the heaps lab but it will make it easier to manage larger *d* values (ie, larger numbers of children).

4.2.1 Helper methods

You won't use all these helper methods for enqueue/sift-up but it makes sense to get them all working so they are ready to use in later tasks. Writing these helper methods now will help you get a feel for your d-heap. Obviously, we recommend that you use these methods as they make your code more adaptable and will be useful later when you implement the `EditableCarHeapQueue`. More importantly, the unit tests will expect these functions to be defined and working.

The helpers you should implement in the `CarHeapQeueue` class are:

- `_index_of_parent`
- `_swap`
- `_indices_of_children`
- `_index_of_min_child`

4.2.2 The `_sift_up` and `enqueue` methods

You should use `_index_of_parent` and `_swap` in your implementation of the `_sift_up` method. Once you have implemented `_sift_up`, you will be able to implement the `enqueue` method. This should be a very short method, relying almost entirely on `_sift_up`. *We recommend that you implement `_sift_up` recursively.*

After completing this task, you should be able to use the heap as follows:

```
from utilities import *
from car_queue import *
heap = run_heap_tests('./test_data/0-5-0-0-0-a.txt', CarHeapQueue, 2)
Enqueueing Car('STJ358', (-5862, 3545), 97, 205)
Enqueueing Car('DOOO30', (-2700, -5302), 18, 4878)
Enqueueing Car('UAD178', (-2589, -8645), 46, 4873)
Enqueueing Car('XMO888', (-9392, 2082), 9, 8754)
```

```
Enqueueing Car('LHN782', (-6099, -312), 73, 1648)
print(heap)
----------------------------------------
CarHeapQueue:
----------------------------------------
0-> Car('STJ358', (-5862, 3545), 97, 205)
1-> Car('LHN782', (-6099, -312), 73, 1648)
2-> Car('UAD178', (-2589, -8645), 46, 4873)
3-> Car('XMO888', (-9392, 2082), 9, 8754)
4-> Car('DOO030', (-2700, -5302), 18, 4878)
----------------------------------------
verify_heapness(heap)
True
```

You should check out the `run_simple_car_heap_tests` function in the `car_queue` module for some simple testing to get your started with your testing.

4.3 Sifting down and dequeue [25 Marks]

The `_sift_down` method is the opposite of the `_sift_up` method. Rather than comparing the priority at index `i` with the parent, we compare the priority at `i` with the minimum child priority. To get the index of the child with the minimum priority you should get the list of child indices and then use `_index_of_min_child` method. Once you have the index of the minimum child you can check whether the `Car` at `i` should swap with that at `min_child_index`—that is, does the data at position `min_child_index` have a lower priority value than that at position `i`. Repeat this procedure until the heap invariant is restored.

Once you have implemented `_sift_down`, please implement the `dequeue` method. Remember the the method for dequeing is to transplant the value from the last node in the heap into the first node, remove the last node, then sift-down from the first node. Therefor the enqueue method should be relatively short, leaving most of the work to `_sift_down`. The `dequeue` method should `raise IndexError("Can't dequeue from empty queue!")` if dequeue is called on an empty queue.
When your enqueue method is working you will be able to run tests on files that include dequeues, eg, the following test run uses a file that contains three enqueues and three deuqueues.

```
>>> from utilities import *
>>> from car_queue import *
>>> heap = run_heap_tests('./test_data/0-3-3-0-0-a.txt', CarHeapQueue, 2)
Enqueueing Car('SDQ593', (-398, -9422), 75, 2357)
Enqueueing Car('NQK252', (-3913, 7441), 1, 8323)
Enqueueing Car('YKY312', (-5427, -7374), 72, 2563)
Dequeued SDQ593, which is right
Dequeued YKY312, which is right
Dequeued NQK252, which is right
>>> print(heap)
----------------------------------------
CarHeapQueue:
----------------------------------------
  Empty
----------------------------------------
```

This is not the end, Fun continues on the next . . .

4.4 Building the queue in O(n) time [15 Marks]

By this point, you should have a fully functioning priority queue, based around the heap data structure. If you look at how we build the heap currently, you will see that any existing data imported is simply enqueued one by one. Because when importing information we have the data all at once from the beginning, it turns out we are able to build the heap in less than $O(n \log n)$ time—we can build the heap in $O(n)$ time, in fact, using a technique called *fast heapify*. For this task, you are expected to implement the `fast_heapify` method in the `CarHeapQueue` class. It should not take more than a few lines.

As a hint, consider that the *slow* heapify uses enqueue, which in turn relies on the `_sift_up` method. Why might this not be the best option? Think about how many items will be sifted up by the maximum number of levels and how many will be sifted up by the minimum number of levels. What other options are there?

> ⚠ **Important:** The fast heapify task appears simple, but is conceptually difficult. You do not need to rigorously prove that your implementation runs in $O(n)$ time, but you should try to convince yourself why it does. Your method should work for any d-heap but it's easiest to think about it in terms of a simple 2-heap. The concept will be the same for any d-factor. Allow sufficient time to solve this problem.

This is not the end, Fun continues on the next ...

4.5 Updating Cars in the queue [20 Marks]

The status of cars is continually changing so the Police want a fast way to update cars that are already in the queue/heap. However, if we were to use our current priority queue implementation, a naïve approach would require us to dequeue Cars until the Car with the plate we want is dequeued, and then re-enqueue all the Cars we just dequeued along with the updated Car. This would take $O(n \log n)$ time which, while not terrible, is certainly not as efficient as it could be.

A better algorithm would to be to perform a linear search through the underlying array, and when the Car with the required plate is found, replace the Car with the updated Car and then sift up/down, as appropriate. The final sifting is $O(\log n)$, but the initial search is $O(n)$—hence, this algorithm runs in $O(n)$ time. Better, but not the best it could be.

For this task, you will be implementing an `EditableCarHeapQueue`, which enables arbitrary car updating (by plate) in $O(\log n)$ time using a new `update` method. This method should expect a Car object (`updated_car`) and it should locate the Car in the queue that has the same plate as `updated_car`. It should then replace the given Car with the `updated_car` and sift up/down as needed.

As with most efficiency improvements, we have traded space for time: you will need to store the index of every Car in a Python dictionary within the `EditableCarHeapQueue` object, ie, as `self._indices`. The dictionary will be keyed by `plate` and the value will be the plate's index in the `_data` list. For example, if the car with the plate `ABA323` is at index 43 in `_data` then `self._indices['ABA323']` should return 43 and `self._data[43]` should return a `Car` with the `NumberPlate` `ABA322`.

Using the dictionary to look-up the index of a Car will therefore take $O(1)$ time. This means that the update operation will take $O(1)$ to find the car to update plus $O(\log n)$ for the subsequent sifting of the replacement—hence the overall update is done in $O(\log n)$ time. The wrinkle in the plan is that you need to constantly keep track of where each Car is stored, even when sifting. For this reason, you should provide updated versions of the `enqueue`, `dequeue`, and `_swap` methods which ensure that the new `self._indices` dictionary is kept up to date with the correct index for any Cars that are moved around. Note, if your `_sift_up` and $_sift_down$ methods use the `_swap` appropriately then they will use the updated `_swap` and therefore will keep the indices updated.

The example below demonstrates the relationship between the `_data` contents and the `_indices` for a heap

```
>>> heap = run_heap_tests('./test_data/0-10-0-0-0-a.txt', EditableCarHeapQueue, 2, verbose=False)
>>> print(heap)
----------------------------------------
EditableCarHeapQueue:
----------------------------------------
0-> Car('GRN365', (264, -205), 4, 320)
1-> Car('VXD992', (1142, -1252), 69, 525)
2-> Car('CWF176', (-4677, 3575), 80, 1177)
3-> Car('MKL432', (-7183, 9267), 87, 1524)
4-> Car('EVH799', (-3377, 9281), 75, 2469)
5-> Car('SGB531', (-4278, 1966), 11, 4190)
6-> Car('RQF183', (-726, 5712), 19, 4663)
7-> Car('SUH894', (6128, -5529), 32, 5612)
8-> Car('PSO464', (-1942, -9444), 77, 2217)
9-> Car('OOV465', (2567, 4799), 41, 3211)
----------------------------------------
>>> print(heap._indices)
{'SUH894': 7, 'MKL432': 3, 'SGB531': 5, 'VXD992': 1, 'EVH799': 4, 'CWF176': 2, 'RQF183': 6, 'GRN365': 0, 'PSO464': 8,
    'OOV465': 9}
>>> print(verify_indices(heap))
True
```

> ⚠ **Important:** Do not reinvent the wheel. Use `super()` to your advantage so that you can reuse code you have already written in Tasks 4.2–4.4.

4.6 Removing cars from any position in the queue [0 Marks]

Sometimes the status of cars changes so that they are no longer interesting to the police. In this case, it is useful to be able to remove cars from the queue. The method for this will be very similar to the update method but it needs to remove the old Car rather than update it. Removing the old Car will be similar to the dequeue operation except you will need to dequeue from a node is at an arbitrary index in the heap, rather than it always being the root node that is being removed/popped.

We won't be testing your `remove` method but we leave it here as a fun exercise for students who are interested.

5 Updates and clarifications

Please keep an eye on the new forums and he assignment section of he quiz server page for any updates or clarifications that might come out during the course of the assignment.

— Have fun —