# Queueing theory assignment: Simulation with event stacks

EBB074A05

Nicky D. van Foreest Nicky

2022:01:19

## 1 General info

This file contains the code and the results that go with the YouTube movies mentioned in the README file in this directory.

You should read the relevant section of my queueing book to understand what it going on. Most of it is very easy, but without background it will be quite cryptic (I believe).

I included a number of exercises to help you think about the code. Keep your answers short; you don't have to win the Nobel prize on literature.

The idea of the code is that you run it yourself as you read through the tutorial.

## 2 General info

This file contains the code and the results that go with this youtube movie:

We will discuss *heap queues* and *classes*. With these we will build a very powerful simulation environment. You should memorize in particular that we implement an *event stack* with a heap queue.

### 2.1 TODO Set theme and font size

Set the theme and font size so that it is easier to read on youbute

```
(load-theme 'material-light t)
(set-face-attribute 'default nil :height 240)
```

By the way, this is emacs lisp; you cannot run it in python.

### 2.2 Load standard modules

I need a new modules `heapq`.

Python Code

```python
from heapq import heappop, heappush
import numpy as np
from scipy.stats import expon, uniform

import matplotlib.pylab as plt
import seaborn as sns; sns.set()

np.random.seed(3)
```

## 3 Sorting with heap queues

Here is a simple example of using heaps to sort data

We are going to sort a bunch of students by age. The heap uses the first element of the *tuple* of a student's age and name to sort the students.

**Ex 3.1.** Look up on the web: what is the difference between a tuple and a list? When to use one or the other? (As always, keep your answer brief.)

Let's put a number of students on the stack.

Python Code
```python
stack = []

heappush(stack, (25, "Cynthia"))
heappush(stack, (21, "James"))
heappush(stack, (21, "James"))
heappush(stack, (25, "Cynthia"))

print(stack)
```

Python Code
```python
age, name = heappop(stack)
print(stack)
```

Python Code
```python
age, name = heappop(stack)
print(stack)
```

Python Code
```python
heappush(stack, (20, "Pete"))
heappush(stack, (18, "Clair"))
heappush(stack, (14, "Jim"))

print(stack)
```

With popping, we take an entry from the top of the stack.

Python Code
```python
age, name = heappop(stack)
print(stack)
```

Python Code
```python
heappush(stack, (15, "John"))
print(stack)
```

**Ex 3.2.** Add an extra attribute to the tuple, for instant a student's height, and make a stack by which you can sort by height. Include your code.

You must have seen during the youtube video that by putting the same element more than once on the stack resulted in a longer, and unsorted, stack. (This came to a surprise to me!) Here I made a simple mistake. A heap queue is not necessarily sorted. However, when *popping* from the stack, it is guaranteed that the elements are popped in sorted order. In other words, it is irrelevant how the stack is sorted, as long as the elements come out in a sorted way. Let's check this.

Python Code

```python
stack = []

heappush(stack, (25, "Cynthia"))
heappush(stack, (21, "James"))
heappush(stack, (21, "James"))
heappush(stack, (25, "Cynthia"))

print(stack)
```

Apparently, the stack does not appear to be sorted. In the code below I pop the elements of the stack, with `heappop`, and append the popped items to the list `res`. You should know that anytime we pop from the stack, the stack becomes one element shorter. So the code below moves elements from `stack` to `res`. Any pass through the while loop removes an element of the stack, and the loop keeps running until the stack is empty. (To help you think about this, this is an ideal exam question.)

Python Code

```python
res = []

while stack:
    res.append(heappop(stack))

print(res)
```

Indeed, `res` *is* sorted.

## 4 Classes

In a class we organize information.

Python Code

```python
class Student:
    def __init__(self, name, age, phone):
        self.name = name
        self.age = age
        self.phone = phone

    def __repr__(self):
        return f"{self.name}, {self.age}, {self.phone}"
```

**Ex 4.1.** After you have read this section on classes, extend the class such that we can give the student also a surname. Include your code.

**Ex 4.2.** Look up on the web: what does the repr method do?

Making an *object* of a *class* is called *instantiation*.

Python Code

```python
hank = Student("Hank", "21", "Huawei")
print(hank)
```

Let's add some more students and put them in a list.

Python Code

```python
students = [
    Student("Joseph", "18", "Motorola"),
    Student("Maria", "21", "Huawei"),
    Student("Natasha", "20", "Apple"),
    Student("Chris", "25", "Nexus"),
]
print(students)
```

**Ex 4.3.** Make two more students, e.g., take your phones, ages and names. (And if you don't like to spread such details, just lie about your age :-) ) Then add these students to the stack. Show your code and the output.

With heaps we can sort the students in any sequence we like. Let's sort them by phone brand.

Python Code

```python
stack = []

for s in students:
    heappush(stack, (s.phone, s))

res = []  # get a sorted list  students in order of their name.
while stack:
    res.append(heappop(stack))

print(res)
```

Note once more that we can provide a *key* as a first element in a tuple, and then insert the entire tuple into a heap. In the rest of the tuple we can put anything we like. Like this, we associate things (here student objects) to keys.

**Ex 4.4.** Change the code so that you can sort the students by age. Show the line(s) of your code to achieve this. Then sort by name.

## 5 Sorting jobs with a heap queue

How can use heap queues in the simulation of queueing systems? To see, think of time as a sequence of events in which things happen.Then, in a queueing system, two things can happen: a job can arrive or a job leaves. So, by specifying the arrival times of jobs, and their departure times, and storing these times in a heap queue, we use the heap queue to sort all these times. Then we jump from event to event, and this is nothing but the queueing simulation!. In other words, with a heap queue, we can let the heap queue do all the work of tracking time. You'll see below we can nearly forget about it. Once you get the idea (which takes some time), you'll see how neat is all is.

### 5.1 A decent job class

We store the arrival, service, and departure time as job *attributes*. We also store the queue length at arrival times to gather statistics at the end. We can then compute the job sojourn time by means of *class methods*.

Let's start from scratch again with the code so that you have a simple starting point.

Python Code

```python
class Job:
    def __init__(self):
        self.arrival_time = 0
        self.service_time = 0
        self.departure_time = 0
        self.queue_length_at_arrival = 0

    def sojourn_time(self):
        return self.departure_time - self.arrival_time

    def waiting_time(self):
        return self.sojourn_time() - self.service_time

    def service_start(self):
        return self.departure_time - self.service_time

    def __repr__(self):
        return f"{self.arrival_time}, {self.service_time}, {self.service_start()}, \
                 {self.departure_time}\n"

    def __le__(self, other):
        # this is necessary to sort jobs when they have the same arrival times.
        return self.id <= other.id
```

**Ex 5.1.** Explain the waiting time and sojourn functions.

Let's make a few jobs and store them in a heap queue. For later purposes, we have to add labels to indicate what type of event we are dealing with, an arrival or a departure.

Python Code

```
ARRIVAL, DEPARTURE = 0, 1

events = []  # event stack, global
num_jobs = 5

time = 0
for i in range(num_jobs):
    time += 3
    job = Job()
    job.arrival_time = time
    job.service_time = 5
    heappush(events, (job.arrival_time, job, ARRIVAL))

while events:
    time, job, typ = heappop(events)
    print(job)
```

# 6 A very powerful GG1 server simulator

## 6.1 The class definition

We need two states to indicate whether the server is busy or idle.

Python Code

```
IDLE, BUSY = 0, 1
```

Python Code

```
class GG1:
    def __init__(self, F, G, num_jobs):
```

```python
        self.F = F # interarrival time distribution
        self.G = G # service time distribution
        self.num_jobs = num_jobs
        self.queue = []
        self.served_jobs = []  # assemble statistics
        self.state = IDLE

    def make_jobs(self):
        time = 0
        for i in range(num_jobs):
            time += self.F.rvs()
            job = Job()
            job.arrival_time = time
            job.service_time = self.G.rvs()
            heappush(events, (job.arrival_time, job, ARRIVAL))

    def run(self):
        while events:  # not empty
            time, job, typ = heappop(events)

            if typ == ARRIVAL:
                self.handle_arrival(time, job)
            else:
                self.handle_departure(time, job)

    def handle_arrival(self, time, job):
        job.queue_length_at_arrival = len(self.queue)
        if self.state == IDLE:
            self.state = BUSY
            self.start_service(time, job)
        else:
            self.put_job_in_queue(job)

    def start_service(self, time, job):
        job.departure_time = time + job.service_time
        heappush(events, (job.departure_time, job, DEPARTURE))

    def put_job_in_queue(self, job):
        heappush(self.queue, (job.arrival_time, job))

    def handle_departure(self, time, job):
        if self.queue:  # not empty
            _, next_job = heappop(self.queue)
            self.start_service(time, next_job)
        else:
            self.state = IDLE
        self.served_jobs.append(job)
```

**Ex 6.1.** Explain how the handle departure method works.

**Ex 6.2.** Explain how the run method works.

## 6.2 Generating jobs

Python Code

```python
labda = 2.0
mu = 3.0
rho = labda / mu
```

```python
F = expon(scale=1.0 / labda)  # interarrival time distribution
G = expon(scale=1.0 / mu)  # service time distribution
num_jobs = 100

events = []

gg1 = GG1(F, G, num_jobs)
gg1.make_jobs()
```

**Ex 6.3.** Suppose we would write

Python Code
```python
gg1 = GG1(G, F, num_jobs)
```

What are then the arrival rate and service rate?

To view the contents of the first couple of events I tried this: `print(events[:5])`, but that failed. After a bit of searching on the web I found the following.

Python Code
```python
from itertools import islice

print(list(islice(events, 0, 5)))
```

**Ex 6.4.** Read on the web what `islice` does, and explain it in your own words.

**Ex 6.5.** Explain also why we need to turn the output of `islice` into a `list`. (This requires that you read and think about what a generator is.)

**Ex 6.6.** Explain the contents of the above table; what do the rows represent? Why is the content of column 5 negative?

## 6.3 Run the simulation

Now run the simulation.

Python Code
```python
gg1.run()
```

Print the first 5 jobs (not all, because when we simulate a 1000 or so jobs, the document explodes).

Python Code
```python
print(gg1.served_jobs[:5])
```

**Ex 6.7.** Explain the first two lines. Are the results in line with what you expect how a $G/G/1$ FIFO queue should behave? If so, why (not)?

## 6.4 Analyze the results, statistics

Here is a plot.

Python Code
```python
plt.clf()
plt.plot(sojourn)
plt.savefig('sojourn0.png')
'sojourn0.png'
```

**Ex 6.8.** Change the seed of the random number generator, choose your favorite number of jobs (something positive, reasonably small). make your own plot and include it in your document.

And some statistics. For instance the sojourn times.

Python Code
```python
sojourn = np.zeros(len(gg1.served_jobs))
for i, job in enumerate(gg1.served_jobs):
    sojourn[i] = job.sojourn_time()

print(sojourn.mean(), sojourn.std(), sojourn.max())
```

Python Code
```python
wait = sojourn.mean() - G.mean()
print(wait)
```

## 6.5 Comparison with Sakasegawa's formula

We can compare the results of our simulation with the theoretical values of the stationary state.

Python Code
```python
def sakasegawa(F, G, c):
    labda = 1.0 / F.mean()
    ES = G.mean()
    rho = labda * ES / c
    EWQ_1 = rho ** (np.sqrt(2 * (c + 1)) - 1) / (c * (1 - rho)) * ES
    ca2 = F.var() * labda * labda
    ce2 = G.var() / ES / ES
    return (ca2 + ce2) / 2 * EWQ_1


average_wait = sakasegawa(F, G, c=1)
print(average_wait, average_wait + G.mean())
```

**Ex 6.9.** Compare the result of Sakasegawa's formula and the results of your simulation (with about 100 jobs). You should remark that the difference in the waiting time is quite large. Do we simulate the $M/M/1$ queue here? Is Sakasegawa's result is exact for the $M/M/1$ queue?

**Ex 6.10.** Run an example with more jobs, 1000 or so. Is now the average sojourn time in the simulation nearer to the theoretical result?

**Ex 6.11.** Another idea is this. We start with an empty system, and that is a very particular state. Suppose we would start with 10 jobs in queue. Do a simulation for 100 jobs, but with 10 jobs in queue at the start. What is then the average sojourn time? Hint, below I include some code to put 10 jobs in queue right at the start. There is subtlety. Af first I set the arrival time to 0 for all these jobs, but then the heapq algorithm doensn't know how to sort the jobs. To repair this problem, I just take very small arrival times.

Python Code
```python
eps = 0.0001
for i in range(10):
    job = Job()
    job.arrival_time = i*eps
    job.service_time = G.rvs()
    heappush(events, (job.arrival_time, job, ARRIVAL))
```

## 6.6 Further experiments

**Ex 6.12.** Change the service distribution to the uniform distribution $U[a, b]$. Take $a$ and $b$ reasonable, so that the load remains below 1. Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times.

**Ex 6.13.** Change the arrival distribution to a uniform distribution, and take also the service distribution to be uniformly distributed. Take some sensitble values for $\lambda$ and $\mathsf{E}[S]$, e.g., $\lambda = 1$ and $\mathsf{E}[S] = 0.4$. Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times.

## 7 SPTF scheduling

Suppose we prefer to serve the shortest job in queue; it can be proven that this scheduling rule minimizes the queue length. We can inherit all of our `GG1` queue, except the scheduling rule. For this we need to change just one line!

Python Code

```python
class SPTF_queue(GG1):
    def put_job_in_queue(self, job):
        heappush(self.queue, (job.service_time, job))
```

And that's it! Now run it.

Python Code

```python
sptf = SPTF_queue(F, G, num_jobs)
sptf.make_jobs()
sptf.run()
print(sptf.served_jobs[:5])
```

## 8 LIFO scheduling

Last-in-First-Out is also trivial.

Python Code

```python
class LIFO_queue(GG1):
    def put_job_in_queue(self, job):
        heappush(self.queue, (-job.arrival_time, job))
```

**Ex 8.1.** Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times. Comment on your findings.

## 9 Serve longest job first

**Ex 9.1.** Update the code of the SPTF queue such that the longest job is selected from the queue, rather than the shortest. (Hint, put a minus sign at the right position when adding a job to the queue heap.) Simulate it. Make a graph of the waiting times and the sojourn times. Comment on your findings.

## 10 Hints