INST0060 Investigation report: Hospital Queuing problem

Group T1*

University College London, WC1E 6BT

May 1, 2020

Abstract

Queuing systems are a common subject of study in reinforcement learning. Here, we simulate a hospital with doctors and patients, each with their own characteristics, and use reinforcement learning to optimize the allocation of arriving patients to a queue.

1 Introduction

1.1 Background

In recent times, hospital queuing times have become a concern. Because hospital represent clusters of vulnerable individuals, it can be of vital interest to limit the number of people standing in line or in a waiting room.

According to a 2015 study, on average Americans spent 84 minutes per visit in the hospital during the 2003-2010 period. [3]

A long-term goal to accommodate the increasingly large flow of patients would be to improve healthcare infrastructures. Yet, because staffing and finance issues are so context-sensitive in many countries, it is not possible to directly ascertain which strategy to follow. However, a more efficient allocation of pre-existing resources can be beneficial in many cases. Furthermore, some decisions in such complex systems could potentially be automated for higher reactivity and better performance.

Reinforcement learning is a form of goal-directed learning with interaction; an agent learns how to respond to situations with certain actions so as to maximize a reward. [5] In a sense, a reinforcement learning agent explores and exploits past experience.

Queuing problems have been optimised using reinforcement learning in the past (such as traffic intersections [4]), and hospitals have been modelled statistically as well. [1, 2]

In this work, we use reinforcement learning to allocate patients to queues as efficiently as possible. First, we will describe the system as modelled (with simplifying assumptions) and the methods used to learn policies. Then, we will analyse the resulting policies in simple cases, to gauge the ability of learners compared to systematic or random policies and to human intuition. We will also reflect on our experience of the project, discussing the various difficulties encountered and what steps were taken in response.

2 Methods

In the first instance, we tried modelling the hospital system as a Markov decision process (or MDP) in continuous time, in order to maintain proximity with both a real hospital system and the material we took away from the INST0060 lectures. However, we quickly realised that we were not equipped to simulate a system in continuous time, and that the complexity of the system (even with our simplifying assumptions) prevented us from using an MDP or table-lookup approach. Hence, we decided to approximate a hospital system by a discrete-time process with a patient arrival every timestep and developed featurisations that allowed for the use of function approximation.

2.1 Model characteristics

The hospital contains servers (doctors) of different types, which correspond to their abilities. Similarly, jobs (patients) have different priorities that warrant different services. For example, a doctor of type 1 can treat a patient with priority 1 or lower, but a doctor of type 0 *cannot* treat a patient with priority 1 or higher.

In general, if the highest doctor type is N at initialisation, then it is expected that there be at least one doctor of each type $n \in \{0, 1, \dots, N\}$. Each *type of* doctor has an associated queue, hence thereafter we call "queue n" the unique queue which doctors of type n draw from. Newly arrived patients have needs sampled from $\{0, 1, \dots, N\}$ according to a

(0bp)= (yellow!15!white); color(6bp)=(yellow); color(50bp)=(black!70!green)]]]]

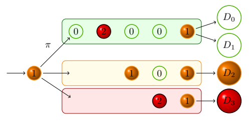

Figure 1: A hospital with 4 doctors: two of type 0 (D_0 and D_1, green), one of type 1 (D_2, orange) and one of type 2 (D_3, red). A newly arrived patient with priority 1 is being allocated to a queue according to policy π. Note the priority 2 patient mistakenly sent to queue 0; they will not be sent away until they have reached the end of the queue.

known distribution and are allocated to a given queue in $\mathcal{A} = \{0, 1, \dots, N\}$ by an agent following a policy π. Note that, once allocated, a patient may not change queues, though they *may be allocated to a doctor who cannot treat them* (see Figure 1). Once they are received by a doctor, if there is a discrepancy, the patient is sent away.

The state space is considered discrete, for simplicity. At each time step:

- a new patient arrives, with priority level sampled at random;
- one or more doctors might be finished treating a patient; when that is the case, the “cured” patient leaves and is replaced by the next-in-line.

Each individual doctor has their own efficiency, which is modelled by the probability of being done with a patient at each timestep.

Hence, if a skilled doctor happens to also be very fast, it might be beneficial to allocate low priority patients to their queue, to deplete the size of queues more quickly; however, this runs the risk of making patients with high priority wait longer.

In this study, we restrict ourselves one doctor per type, as this still offers ample flexibility. The general case could be the subject of future work. Similarly, though this study started with one queue *per doctor* rather than per *type* of doctor, we simplified the system after discussions with the module administrators and TAs.

One of the challenges associated with learning a policy to set balanced priorities between curing patients and keeping the hospital from getting too full. Because of the behaviour of misallocated patients (see Figure 1), it can be more efficient for the agent to just misallocate patients from the start so as to keep the hospital empty.

The agent’s actions tend to have very delayed consequences, which makes it difficult for it to link rewards and policy. Further, originally the penalty for misallocation was set to trigger only when patients reached the doctor. This mechanic was modified midway through the project to investigate its effect, and this will be the subject of Experiment 3.

2.2 Learning

Two well-known algorithms of RL are the Q-learning (QL) and the SARSA algorithms. [5] SARSA is an on-policy algorithm, meaning the agent learns the value of the state-action pair on the basis of the performed action. By contrast, QL is an off-policy algorithm, where the state-action-value function (or q-function) is estimated based on a greedy policy, which is not necessarily the currently implemented policy. Generally, SARSA tends to exhibit faster convergence, while the performance of Q-learned policies tend to be higher. However, in SARSA the learner can easily get stuck in local minima, and QL tends to take longer to converge.

2.3 Featurisation

Even with the limited complexity of its modelling, the number of individual states in the hospital is extremely large, since each state includes the number of people in each queue, but also the types of patients and their waiting times. In fact, it is countably infinite if the hospital occupancy in not bounded (which it is during training), so that a simpler representation (a simple vector) must be used for the agent to learn efficiently.

While at the beginning a few real numbers were used (such as the average number of patients per queue), later on an effort was made to include more information (real numbers, but converted to discrete values) and finally, one-hot vectors. This last attempt was the most successful and will be applied throughout this work; the impact of featurisations will be the subject of experiment 1.

In this work, the systems of interest will be:

Experiment 1: Two doctors (one of type 0 and one of type 1) with one featurisation and one learning algorithm. The

Experiment 2: We analyse the behaviour of the agent as the frequency of urgent patients increases.

3 Results

3.1 Experiment 1: Featurisations (code in `feat_exp.py`)

In this experiment, we compare the performance of different featurisations. The hospital has six doctors, of types 0 through 5. Here, doctors 0,1,4 are inefficient (0.1), doctors 3,5 are efficient(0.4 and 0.5, respectively) and doctor 2 is highly efficient(0.9). The system is deliberately complex, so that patient allocation is not trivial, but still intuitive for an observer. Patients of different types arrive with equal probability. Here we use the SARSA algorithm, but this is not a siginificant choice since our main concern are the featurisations. Furthermore, the learning process consists of 100 episodes with 100 steps each. The learning process is repeated 100 times for each feature so that we can get consistent data. This is essential since our model incorporates a certain level of randomness.

The most important characteristic of the current model, apart from the definition of the used featurisation, is the reward system. This system has undergone many iterations before its current state and is perhaps not perfect. Nevertheless, it produces stable and logical results, as far as our data shows. If a patient is allocated to a queue of a doctor who cannot treat them then a penalty proportional to the the type of the patient is immediately recognized, but the patient continues to wait in the queue. The only other time when a reward is assigned is when a patient reaches a doctor. No matter whether the patient can or cannot be treated by the doctor a penalty directly proportional to the waited time and patient's type is recognized. However, if the patient can be treated then a further constant positive reward is recognized as well. The fact that a penalty is immediately received if a patient is sent to an inappropriate queue is a strong indicator for our better featurisations and they would not really allow this to happen. The constant positive reward for a treated patient mitigates misallocation of patients as well. Before it was introduced the system was occasionally willing to assign patient to wrong (but empty) queues and in that way get rid of them faster and with small penalty.

Finally, it is time to take the features into consideration. We have created a variety of featurisations and they can all be found in `learning.py` file. They generally fall into two categories: one-hot vectors (featurisations 7-12) and non-one-hot vectors (featurisations 1-6). We note that feature 1 is the best performer from the non-one-hot ones and still it appears to be just as good as the simple naive policy. Given this, the data shown in the bar plot below implies that one-hot vectors are the better choice(the upper bound of the total reward is 10 000 in this experiment).

`Feature 1` description, discussion TBI

`Feature 7` description, discussion TBI

`Feature 12` description, discussion TBI

Will add a plot for the reward evolution of all the features perhaps

Also some plot about allocation/misallocation of patients

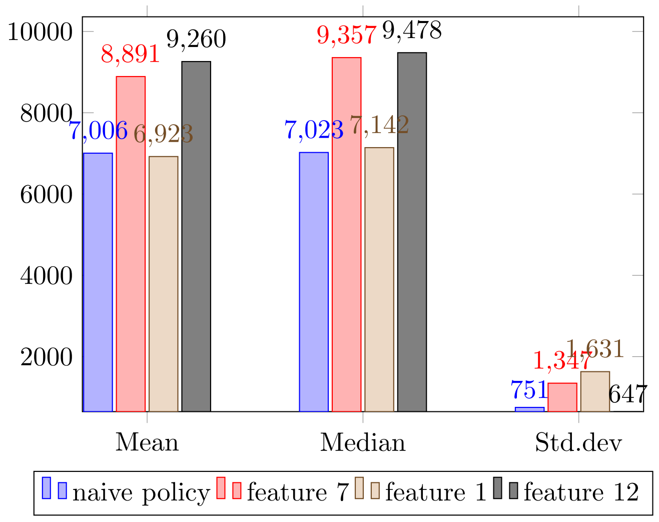

Figure 2: Caption

3.2 Experiment 2: Model behaviour (code in `fastdoc_exp.py`)

In this experiment, we try to characterise the behaviour of the agent in ambiguous situations. The hospital has four doctors, of types 0 through 3. Doctor 3 (the most highly skilled) is very efficient (80% chance of treating a patient at each time step), while lower skilled doctors are equally slow (40%).

We set the featurisation to `feature_12`, the learning algorithm to SARSA and the reward system to penalise misallocations early and not penalise when occupancy is reached, as these seem to yield better performance overall. These assumptions are the object of experiments 1 and 3. New patients are equally likely to be of type 0, 1 or 2. We vary the probability of type 3 patient arrivals during training, from 10% to 90%. When there are few urgent patients, we would expect the agent to allocate many low priority patients to doctor 3 because it treats more quickly. However, when there are many, allocating low-priority patients to doctor 3 causes type 3 patients to wait longer, which is detrimental to the reward.

For each data point, we train an agent on a hospital with a given arrival probability and then apply the learned policy to a hospital with *equiprobable* arrivals so that policies can be compared on equal footing. We record the proportion of patients of type 3 allocated to queue 3, the *overall* proportion of patients allocated to queue 3, and the average time waited by a patient that has seen a doctor.

We start by running a "short-term" trial, in which for each probability, the simulation (training and testing) was carried out 300 times; for each simulation the agent was trained for 50 episodes with 500 steps each.The results are shown in Figure 3.

We mentioned earlier that we should compare policies on an equal footing. However, this is not necessarily possible considering the agent might not have had equal access to each state in each circumstance: when type 3 patients arrive 90% of the time, the agent gets little experience with treating lower priority patients. In this case, since patients of type 3 can only be put in queue 3 (any other decision would result in a misallocation penalty), the agent learns to put any arriving patient in queue 3, irrespective of their type: this is reflected in Figure 3.

Yet, the behaviour for low probability agrees with our predictions.

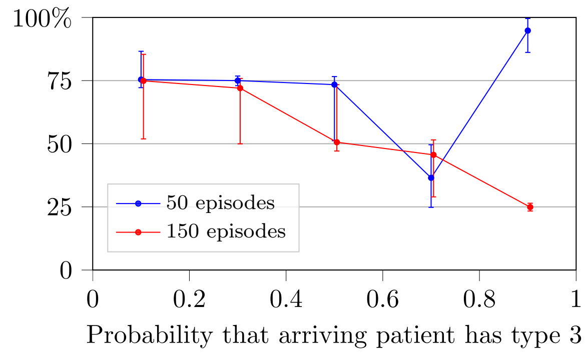

Figure 3: Proportion of patients allocated to queue 3 as priorities change. Error bars show 20th and 80th percentile (to represent the high skew). Points are artificially offset for readability.

In response to this extraneous, though predictable behaviour, we also ran a "long-term" trial, with the same parameters and training lasting 150 episodes instead of 50. This way, the model can gather more meaningful experience with each type of patient during training, even when arrivals of some patient type are sparse. For each point, the simulation was carried out 100 times (which took around 4 processor hours on a modern laptop). The result is in Figure 3.

It would seem that in general the agent is more cautious about allocating patients to queue 3, particularly when urgent patients are frequent (the probability is about 25%, which corresponds to only the type 3 patients being put in queue 3). This is

3.3 Experiment 3: Penalty check (code in `rewards_exp.py`)

Experiment 3 tests and compares the efficiency of queues with different combinations of two penalties, i.e. the penalty for misallocation (`earlyRewards`) and the penalty for reaching the occupancy of the hospital (`capacity_penalty`). When the former one is true, the penalties, proportional to time waited, are issued directly and immediately when a patient is sent to a specific queue incorrectly before the patient reaches the doctor. When the later one is true, it means that a penalty for inefficiency, which is extremely large, should be taken into account when the maximum capacity of the hospital is reached either at or before the episode terminates.

Figure 4 depicts the distribution of the frequency rate of misallocation with capacity = 100, steps =100, episodes =10. It can be thus inferred that SARSA contains a higher rate of misallocation than Q-Learning does, indicating that Q-Learning might have a better performance in improving the efficiency of queues. The standard deviations of SARSA are relatively large, which also shows that the output values of SARSA are spread out over a wider range.

This experiment also compares the results when we apply two different rewards. Firstly, we create early reward in our model. When it is true, the rewards are allocated directly when the patient is sent to a specific queue, which represents that the patients in hospital are allocated to each queue timely. Its rewards are calculated proportionally according to the waiting time of patients. Besides, there is a penalty for full capacity of hospital. Capacity penalty equals True means when the capacity is reached not only the episode terminates but also the maximum capacity of hospital. In `hospital.py` file, we create a function to compare the length of all the queues to the occupancy of hospital. Large number of negative reward is accumulated if the former is bigger. In experiment 3, early rewards and capacity penalty are divided into four categories. As showed in Figure 3, TT, TF, FT and FF stand for true or false of early rewards and capacity penalty respectively. For example, TF means early rewards equals true and capacity penalty equals false.

Figure 3 summarizes the performance of Sarsa and Q learning with early reward and capacity penalty. X-axis can be explained as their status of applying penalties and y-axis is the occurrence frequency of misallocation of patients in hospital. Black line in the middle of each bar represents standard deviation of input values. Overall, Sarsa algorithm has higher probability to misallocate patients in the wrong queues than Q learning. In addition, large frequency appears when we only apply capacity penalty or not request both early reward and capacity penalty.

4 Discussion

5 Conclusion

Declaration

This document represents the group report submission from the named authors for the project assign-

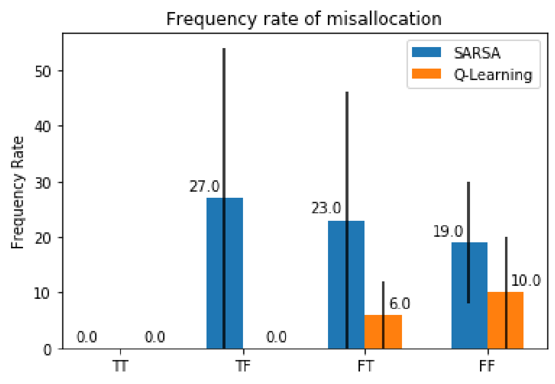

Figure 4: The frequency rate of misallocation

ment of module: Foundations of Machine Learning and Data Science (INST0060), 2019-20. In submitting this document, the authors certify that all submissions for this project are a fair representation of their own work and satisfy the UCL regulations on plagiarism.

References

[1] Jiekun Feng and Pengyi Shi. Steady-state diffusion approximations for discrete-time queue in hospital inpatient flow management. *Naval Research Logistics (NRL)*, 65(1):26–65, 2018.

[2] Matthew S. Hagen, Jeffrey K. Jopling, Timothy G. Buchman, and Eva K. Lee. Priority queuing models for hospital intensive care units and impacts to severe case patients. *AMIA Annual Symposium proceedings Archive*, 2013:841–850, 2013.

[3] Kristin N. Ray, Amalavoyal V. Chari, John Engberg, Marnie Bertolet, and Ateev Mehrotra. Opportunity costs of ambulatory medical care in the United States. *The American Journal of Managed Care*, 21(8):567–574, 2015.

[4] Azura Che Soh, Mohammad Hamiruce Marhaban, Marzuki Khalid, and Rubiyah Yusof. Modelling and optimisation of a traffic intersection based on queue theory and Markov decision control methods. In *First Asia International Conference on Modelling & Simulation (AMS'07)*, pages 478–483. IEEE, 2007.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.