2DI66 Advanced Simulation - Assignment 2
Group 47

Robin Jonker
1011291
r.m.jonker@student.tue.nl

Muhammed Karakurt
1565583
m.karakurt@student.tue.nl

March 14, 2021

Introduction

The problem is about a complex queuing system involving multiple stations and a moving server with different server discipline options. The server handles customers at the visited station and after some criteria are met the server switches to the next station with deterministic switch-over times. These criteria depend on server disciplines which determine how many customers the server would handle during each visit. Exhaustive service discipline means that when a server visits a station, it stays there until no customers are there for it to handle. K-limited service discipline is the same as exhaustive service, but no more than k_i customers can be served during one visit period to station i. Gated service means that when the server arrives at station i, it counts how many customers are present at that station at that particular moment, and then only those customers will be served during that visiting period.

The nature of the problem is stochastic because customers arrive with the Poisson process, internal arrivals and external departures have some probability distribution and the service times are assumed to be exponentially distributed.

The objective of the problem is to find about the performance of the queuing system with some defined performance measures for different service disciplines. These measures are the steady-state waiting times for each station (W_i), the steady-state queue length of each station (Q_i), the sojourn time of a customer that leaves the system after a visit to station i (T_i) and the cycle times for each station which means the time between two visit beginnings of station i (C_i).

Finding the Simulation Parameters

Required Number of Runs

Aiming to get an accuracy of 0.5 for the 95% confidence intervals for the average output variables, we used the inequality 1 with $\varepsilon = 0.5$ and $Z_{\alpha/2} = 1.96$ to determine the number of runs required to reach the desired level of accuracy. Since we don't know the standard deviation for any of the output variables beforehand we performed a short, initial simulation with small number of runs (3 runs). Then we calculated the standard deviations of all output variables for all stations and estimated σ as the mean of these standard deviations for each server discipline. After that we plugged in the estimated σ to the inequality 1 and the results are as follows:

- $\hat{\sigma} = 0.7422$ and $n > 8.4632 \Rightarrow n = 9$ for the exhaustive service discipline
- $\hat{\sigma} = 1.3583$ and $n > 28.3504 \Rightarrow n = 29$ for the k-limited service discipline
- $\hat{\sigma} = 0.3249$ and $n > 1.6213098 \Rightarrow n = 2$ for the gated service discipline

Thus, we needed different number of runs for each server discipline of the simulation to get the aimed accuracy.

$$n > \left(\frac{Z_{\alpha/2} \cdot \hat{\sigma}}{\varepsilon}\right)^2 \tag{1}$$

Total Simulation Length and Warm-Up Length

To determine the total simulation length and warm-up length we have plotted the two of the performance measures (Q_i and W_i) with time for all different service disciplines. We have decided to comment on the average queue length(Q_i) graph for the gated service discipline. The conclusions driven from that graph is valid for all service disciplines.

As can be seen in Figure 4 in appendix A.1, the performance measure have spikes until approximately 5000 which could distort the output variables. Also, the performance measures don't change significantly after around 40,000 which means the simulation reaches the steady-state around that time. In light of these findings, we decided to set the maximum simulation time to 40,000 and the warm-up period to 5000. Output variables won't be recorded until the simulation time reaches the warm-up time (5000) and the simulation will go on until it reaches maximum time (40,000).

The Code

Our Simulator was created using an object-oriented approach in Python. We have the following eight classes: Simulator, Network, Server, Station, Queue, Customer, Event and FES. This section will go over each class and explain their functionality. The source code can be found in Appendix C. We will provide pseudocode for the relevant methods. With the implementation described in this Section, we can do one run of the simulation with a max_time of 40000 seconds in around 2 seconds. Thus, to simulate the number of runs per service discipline as specified in the previous Section, we needed less than 2 minutes in total.

Simulator is the core of our program. When it is run, the `main` method of the Simulator is called with a filename of the input file. The pseudocode of the main method can be found in Algorithm 1. On line 2, the `read_input` method is called, which is simple a method to read the input txt file, splitting it into different lines (and each lines into words). Each word is then appended to a list of a certain input variable based on the current line number. This method returns the number of stations and a list of the lambdas, the expected service times, the expected switch times, the list of k-limited values and a 2d list of probabilities (including the probability to leave the system from a certain station).

On line 5 of Algorithm 1, a **Network** object is created. It needs several variables: the serv_discipline, nr_stations, starting_station and the lists of input variables. On object creation, it uses these input variables to initialize several parts of the Network:

- Several instance variables: current_time = 0, warm_up_time = 5000, scheduled_first_departure = False.
- a **Server** object.
- a **FES** object.
- *nr_stations* of **Station** objects, each with their own **Queue** object. These stations are stored in a list in the Network.
- The first arrival **Event** is scheduled for each Station.

Algorithm 1: main(input_file)

```
Result: Simulation has been run nr_runs times, and the resulting statistics have been computed
1  Define simulation parameters (nr_runs, max_time, warm_up_time and service discipline);
2  Call read_input(input_file) method to collect lists for each of the input variables;
3  for i ← 0 to nr_runs by 1 do
4      Select random starting station;
5      Create a Network object based on input variables;
6      Call simulate_network(network, max_time) to simulate the created network;
7      Store the average queue length and mean cycle, sojourn and waiting times for this run;
8  end
9  for i ← 0 to nr_stations by 1 do
10     For the current station, compute the average of the statistics per run;
11     Compute the standard deviation of the statistics;
12     Compute the 95% confidence intervals of the statistics;
13     Print the results for the current station;
14 end
15 for i ← 0 to nr_stations by 1 do
16     For the current station, compute the average of the statistics per run;
17 end
```

On line 6 of Algorithm 1, the simulate_network method is called for the created **Network**. This method runs simulates the network up to max_time, returning the relevant statistics when it finishes. The pseudocode of the simulate_network method can be found in Algorithm 3. This pseudocode can be split into several sections as follows: after line 1 initializes the result statistic lists, lines 2-45 contain the main while loop for our simulation. In every loop, we get the newest event and update the time. In lines 7-11 we handle arrivals, and in lines 12-28 we handle departures. Lines 29-44 contain the inner while loop that checks if the server should switch to the next station. The checks for this are done in the `should_switch_station()` method in the Network object. It should be noted we have several if statements with 'if current_time > warm_up_time' before updating statistics, which is done to only start collecting statistics in the steady state. Finally, line 46 returns the collected statistics to the Simulator.

The Network object contains several important methods: schedule_departure and schedule_arrival, which are methods to respectively create a departure or an arrival event in the FES, and handle_arrival, a method that handles an arrival event by creating a Customer and adding it to the queue specified by event.queue. These three methods are all very short and self-explanatory, so we have chosen to omit the pseudocode.

The final method in the Network object is the should_switch_station method. It is used as the guard of the inner while loop of Algorithm 3, on line 29. The pseudocode for this method can be found in Algorithm 2. The method checks all relevant switch conditions for the selected service strategy, and if those are triggered, it returns true. For all strategies this involves checking if the queue is empty (as this would always mean the server has to switch.

For k-limited and gated strategies there are secondary conditions which can force the server to switch, which are checked using the methods k_i_customers_served and gated_target_customers_served in the **Server** object. The Server object keeps track of the current station of the server, but also of the number of customers served at the current station (in served_at_current_station) and, for gated strategy, the target number of customers to handle at the current station (in gated_target). k_i_customers_served checks if served_at_current_station is greater or equal than the k_i parameter for the current station, while gated_target_customers_served checks if served_at_current_station is greater or equal than the gated_target. The final method in the Server object is the move_station method, which simply sets the current_station to (current_station + 1) % nr_stations, and resets served_at_current_station to 0. As each of these methods in the Server object are only one or two lines long, we have again chosen to omit the pseudocode for these methods.

The remaining object classes are all very simple. The **Station** class simply keeps track of the arrival rate, the service rate, the list of transition probabilities and its own Queue object. The **Queue** class (based on the SwitchingServerQueue example provided by the course) simply keeps track of a list of Customers, and has methods to get the size of the queue, to add customers to the queue and to pop customers from the queue. The **Customer** class stores the time a customer arrives in the system, and the original station that the Customer entered the system in.

The final two object classes are the **Event** and the **FES** classes. The Event class tracks the type of the event (arrival or departure), the time the event will take place and the station for which the event will take place. The randomly generated interarrival time (for arrivals) or service time (for departures) are also stored in the Event class. The FES class (also based on the SwitchingServerQueue example provided by the course) uses a heapq to track the scheduled Events in order of arrival time. Using the add() and next() methods, events can be either added or popped from the heapq. There are also method to check if the heapq is empty, to get the length of the heapq and to peek at the first event in the heapq without popping it.

Algorithm 2: should_switch_station()

```
Result: Returns if the server should switch stations, by checking the conditions for the selected strategy
if server.current_station.queue is empty then
    return True (as the server should always switch if the current queue is empty)
if strategy is k-limited then
    if strategy is k-limited and server.k_i_customers_served then
        return True (as the server has served the k customers needed for k-limited at the current station)
else if strategy is gated and server.gated_target_customers_served then
    return True (as the server has served the customers that were present when the server entered the station)
return False (server should not switch if none of the above conditions are met)
```

```
Algorithm 3: simulate_network(network, max_time)
Result: Given network has been simulated for max_time, and the resulting statistics are returned
Lists for all result statistics are initialized;
while current_time < max_time do
    Get next event from the FES;
    Update the current_time based on the time of the event;
    if current_time > warm_up_time then
        Update cumulative_queue_lengths statistic of all stations;
    if event is an arrival event then
        Call network.handle_arrival(event);
        if network.scheduled_first_departure is False and arrival event is for current station of the server then
            Call network.schedule_departure(current_station);
            Set scheduled_first_departure to True;
    else if event is a departure event then
        Get the Station of the departure event and pop the first Customer from its Queue;
        Increase served_at_current_station variable of Server object by 1;
        Select a random destination for the customer based on the given probabilities;
        if current_time > warm_up_time then
            Update waiting_times statistic for current station;
        if customer is leaving the system then
            if current_time > warm_up_time then
                Update sojourn_times statistic for current station;
        else
            Add the customer to the destination queue (as internal arrivals are instant);
        if strategy is exhaustive and current_station.queue is not empty then
            Call network.schedule_departure(current_station);
        else if strategy is k-limited and current_station.queue is not empty and not network.server.k_i_customers_served
         then
            Call network.schedule_departure(current_station);
        else if strategy is gated and not network.server.gated_target_customers_served then
            Call network.schedule_departure(current_station);
    while network.should_switch_station do
        Compute time that server will be done switching stations;
        while If event.time of event in front of fes < time_done_switching do
            Handle all events that happen before Server is done switching;
        end
        Move server to current_station + 1 mod nr_stations;
        Reset network.served_at_current_station to 0;
        if strategy is gated and current_station.queue is not empty then
            network.server.gated_target = current_station.queue.size;
        Update current time to time_done_switching;
        if current_time > warm_up_time then
            Update cycle_times statistic for new station;
        if new_station.queue is not empty then
            Call network.schedule_departure(new_station);
            Set network.scheduled_first_departure to True;
    end
end
Return the collected statistics (waiting times, cumulative queue lengths, sojourn times and cycle times)
```

Performance Measures

In this Section, we report the result of the simulation using the input values shown in Tables 7 and 8 in appendix B.

Exhaustive Service Discipline

Station	$\mathbb{E}[W_i]$	$\mathbb{V}[W_i]$	$\mathbb{E}[Q_i]$	$\mathbb{V}[Q_i]$	$\mathbb{E}[C_i]$	$\mathbb{V}[C_i]$	$\mathbb{E}[T_i]$	$\mathbb{V}[T_i]$
1	9.3667	0.0623	1.3135	0.0035	23.1848	0.1981	56.2006	5.4093
2	9.4895	0.0544	1.6685	0.0064	23.1858	0.1998	46.7438	6.1138
3	11.3948	0.0502	1.1356	0.0023	23.1850	0.1995	64.2431	4.2911
4	10.1658	0.0642	1.7000	0.0057	23.1846	0.1995	48.5513	2.3605
5	12.3903	0.0965	1.5106	0.0041	23.1830	0.1991	66.2227	9.4409
6	11.6592	0.0893	1.5452	0.0051	23.1831	0.1978	55.9826	3.9545

Table 1: Performance Measures for Exhaustive Service

	Waiting Time			Queue Length			Cycle Time			Sojourn Time		
Station	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound
1	9.2036	9.3667	9.5297	1.2748	1.3135	1.3522	22.8940	23.1848	23.4756	54.6811	56.2006	57.7201
2	9.3371	9.4895	9.6419	1.6161	1.6685	1.7208	22.8937	23.1858	23.4778	45.1283	46.7438	48.3592
3	11.2484	11.3948	11.5412	1.1043	1.1356	1.1669	22.8932	23.1850	23.4769	62.8897	64.2431	65.5964
4	10.0002	10.1658	10.3313	1.6505	1.7000	1.7496	22.8928	23.1846	23.4764	47.5475	48.5513	49.5551
5	12.1873	12.3903	12.5933	1.4687	1.5106	1.5524	22.8915	23.1830	23.4744	64.2152	66.2227	68.2301
6	11.4640	11.6592	11.8545	1.4986	1.5452	1.5917	22.8925	23.1831	23.4736	54.6834	55.9826	57.2818

Table 2: Confidence Intervals of Performance Measures for Exhaustive Service

For the exhaustive service discipline the performance measures in Table 1 demonstrate valuable insights. First we can see that stations' λ and $\mathbb{E}[B]$ values directly affect the steady-state waiting time, queue length and sojourn time which is expected. As λ decreases (holding $\mathbb{E}[B]$ constant) which means the arrivals are more frequent and we have the same service rate, we observe higher waiting times and higher sojourn times for that station's customers. We can see this comparison very clearly when we take a look at stations 4 and 5, they have the same service rates and different λ values. However, the steady-state waiting time and the queue length isn't always correlated, this might be because of the unbalanced transition probabilities. Another insight is that for every station, steady-state sojourn time may be a good indicator for the number of stations that a customer visits before departing from that station to outside of the system. Lastly, we have observed that the mean cycle times converge approximately to the same value in the steady state for all stations which we will elaborate in the theoretical validation part.

Validation of Theoretical Results

We have used the theoretical findings by Sidi, Levy, and Fuhrmann [1].These theoretical results are calculated for exhaustive and gated service strategies. For the validation, first we have used equation 2 to solve the system of linear equations and find γ values for each station. Then we have calculated the network utilisation using the equation 3 and the mean cycle time using equation 4.

$$\gamma_i = \lambda_i + \sum_{j=1}^{6} \gamma_j \cdot p_{ji} \quad \forall i \in \{1, 2, 3, 4, 5, 6\} \tag{2}$$

$$\rho = \sum_{j=1}^{6} \gamma_i \cdot \mathbb{E}[B_i] \tag{3}$$

$$\mathbb{E}[C_i] = \frac{r}{1-\rho}, \quad r = \sum_{i=1}^{6} \mathbb{E}[R_i] \tag{4}$$

With our input values theoretical network utilisation is calculated as **0.6283** and the mean cycle time is calculated as **22.5989** and our stochastic simulation results for mean cycle times are between **23.1830** and **23.1856**. Furthermore when we take a look at the confidence intervals for the mean cycle times they are between **22.8915** and **23.4778**. Although there is a very slight discrepancy between the simulation's results and the theoretical results, this discrepancy is negligible in a practical sense. In conclusion, this validates our simulation's results.

To validate the stability condition of the system we tweaked the input parameters and the resulting theoretical network utilisation is calculated as **1.1267**. Then we plotted one of the performance measures (Q_i) with time and from the figure 5 in appendix A.2 we can see that the output variable doesn't converge to some value. This validates that for exhaustive service discipline if the theoretical network utilisation is bigger than 1 the system becomes unstable and all the performance measures go to infinity.

Gated Service Discipline

Station	$\mathbb{E}[W_i]$	$\mathbb{V}[W_i]$	$\mathbb{E}[Q_i]$	$\mathbb{V}[Q_i]$	$\mathbb{E}[C_i]$	$\mathbb{V}[C_i]$	$\mathbb{E}[T_i]$	$\mathbb{V}[T_i]$
1	12.7488	0.1424	1.7610	0.0083	23.2754	0.2261	75.0439	8.5223
2	14.5077	0.2168	2.4546	0.0184	23.2757	0.2264	65.1046	7.0411
3	14.1034	0.1365	1.3759	0.0046	23.2782	0.2256	85.5598	7.7267
4	15.5113	0.1783	2.4947	0.0155	23.2782	0.2256	67.1396	6.6281
5	15.9475	0.2642	1.8753	0.0116	23.2781	0.2244	87.9783	9.8784
6	14.3174	0.2036	1.8715	0.0103	23.2753	0.2265	74.3325	5.0432

Table 3: Performance Measures for Gated Service

	Waiting Time			Queue Length			Cycle Time			Sojourn Time		
Station	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound
1	12.5149	12.7488	12.9827	1.7045	1.7610	1.8175	22.9806	23.2754	23.5701	73.2345	75.0439	76.8533
2	14.2191	14.5077	14.7962	2.3704	2.4546	2.5387	22.9809	23.2757	23.5706	63.4599	65.1046	66.7492
3	13.8745	14.1034	14.3324	1.3340	1.3759	1.4178	22.9837	23.2782	23.5726	83.8369	85.5598	87.2826
4	15.2496	15.5113	15.7729	2.4175	2.4947	2.5719	22.9838	23.2782	23.5726	65.5439	67.1396	68.7353
5	15.6289	15.9475	16.2661	1.8085	1.8753	1.9420	22.9845	23.2781	23.5717	86.0303	87.9783	89.9264
6	14.0378	14.3174	14.5971	1.8085	1.8715	1.9344	22.9804	23.2753	23.5703	72.9406	74.3325	75.7244

Table 4: Confidence Intervals of Performance Measures for Gated Service

For the gated service discipline the performance measures in Table 3 demonstrate the same insights that is stated in previous section for exhaustive service discipline.

Validation of Theoretical Results

Our stochastic simulation results of mean cycle times for gated service discipline are between **23.2754** and **23.2782**. Furthermore when we take a look at the confidence intervals for the mean cycle times they are between **22.9804** and **23.5726**. Although there is a very slight discrepancy between the simulation's results and the theoretical results, this discrepancy is again negligible in a practical sense. In conclusion, this validates our simulation's results.

To validate the stability condition of the system we again tweaked the input parameters and the resulting theoretical network utilisation is calculated as **1.0787**. Then we plotted one of the performance measures (Q_i) with time and from the figure 6 in appendix A.2 we can see that the output variable again doesn't converge to some value. This validates that for gated service discipline if the theoretical network utilisation is bigger than 1 the system becomes unstable and all the performance measures go to infinity.

K-Limited Service Discipline

Station	$\mathbb{E}[W_i]$	$\mathbb{V}[W_i]$	$\mathbb{E}[Q_i]$	$\mathbb{V}[Q_i]$	$\mathbb{E}[C_i]$	$\mathbb{V}[C_i]$	$\mathbb{E}[T_i]$	$\mathbb{V}[T_i]$
1	13.0488	3.2699	1.7903	0.0895	23.2269	0.5061	79.7971	86.0961
2	14.1259	2.9464	2.3987	0.1118	23.2262	0.5062	67.5380	56.2125
3	15.7814	2.6449	1.5476	0.0417	23.2266	0.5067	91.5766	82.4864
4	13.9183	2.0501	2.2422	0.0812	23.2252	0.5071	69.9287	56.6482
5	21.2337	8.7871	2.4588	0.1535	23.2268	0.5050	98.8400	118.9670
6	17.2306	5.2066	2.2268	0.1256	23.2269	0.5061	81.9116	86.6128

Table 5: Performance Measures for K-Limited Service

	Waiting Time			Queue Length			Cycle Time			Sojourn Time		
Station	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound	Lower Bound	Mean	Upper Bound
1	12.3906	13.0488	13.7069	1.6814	1.7903	1.8992	22.9679	23.2269	23.4858	76.4200	79.7971	83.1743
2	13.5011	14.1259	14.7506	2.2771	2.3987	2.5204	22.9672	23.2262	23.4852	64.8092	67.5380	70.2669
3	15.1895	15.7814	16.3733	1.4733	1.5476	1.6220	22.9675	23.2266	23.4857	88.2710	91.5766	94.8822
4	13.3972	13.9183	14.4395	2.1384	2.2422	2.3459	22.9660	23.2252	23.4844	67.1893	69.9287	72.6680
5	20.1548	21.2337	22.3126	2.3162	2.4588	2.6014	22.9681	23.2268	23.4854	94.8702	98.8400	102.8098
6	16.4001	17.2306	18.0611	2.0978	2.2268	2.3558	22.9680	23.2269	23.4858	78.5243	81.9116	85.2988

Table 6: Confidence Intervals of Performance Measures for K-Limited Service

For the k-limited service discipline the performance measures in Table 5 demonstrate the same insights as all other service disciplines.

Comparison with Theoretical Results

Our stochastic simulation results of mean cycle times for gated service discipline are between **23.2252** and **23.2269**. Furthermore when we take a look at the confidence intervals for the mean cycle times they are between **22.9660** and **23.4858**. Although there is a very slight discrepancy between the simulation's results and the theoretical results, this discrepancy is again negligible in a practical sense.

To check for the stability condition of the system we again tweaked the input parameters and the resulting theoretical network utilisation is calculated as **1.2413**. Then we plotted one of the performance measures (Q_i) with time and from the figure 7 in appendix A.2 we can see that the output variable doesn't converge to some value. This means even though it is not theoretically shown that a moving server queuing system with k-limited service discipline has a stability condition, with our simulation we have reached the same conclusion as other service disciplines. In conclusion, for k-limited service discipline the theoretical result for mean cycle time is very close to the simulation's results and the theoretical results also seems to be valid for k-limited service.

Comparing the Three Service Disciplines

To get better insight in the performance of the three service disciplines, we created grouped bar charts from the results for the mean W_i (Figure 1, mean Q_i (Figure 2) and mean T_i (Figure 3). We have chosen to omit a grouped bar chart for the mean C_i, as for all service disciplines and all stations, the C_i is around 23.2 and the confidence intervals overlap for all of them. Thus, a graph for this would provide little insight.

Figure 1 shows that, independent of the station, an Exhaustive service strategy provides the shortest average waiting times for customers. The Gated and K-Limited strategies prove to be very close in waiting times for most stations, although at station 5 and 6, a Gated strategy seems to provide significantly shorter waiting times for customers.

Figure 2 shows that, independent of the station, an Exhaustive service strategy provides the shortest average queue lengths. Again, the Gated and K-Limited strategies prove to be very close in terms of average queue length. For stations 1-3 the difference is very small. While stations 5 and 6 have on average shorter queue lengths for Gated than for K-limited strategy, station 4 shows the opposite. Another interesting observation is that Figure 2 follows a very similar pattern to Figure 1. This indicates that these variables are quite correlated. This makes sense, as shorter queue lengths mean that customers have to wait less before being served. However, it should be noted that not all of the results share this pattern: station 5 has a higher waiting time than station 4 with exhaustive and gated strategies, while the queue length is shorter at station 5 than at station 4.

Finally, Figure 3 shows that once again, independent of the station, an Exhaustive service strategy provides best performance with in this case the lowest average sojourn times. A Gated strategy comes in second place at all station, with significantly longer average sojourn times than the Exhaustive strategy and slightly shorter times than the K-Limited strategy.

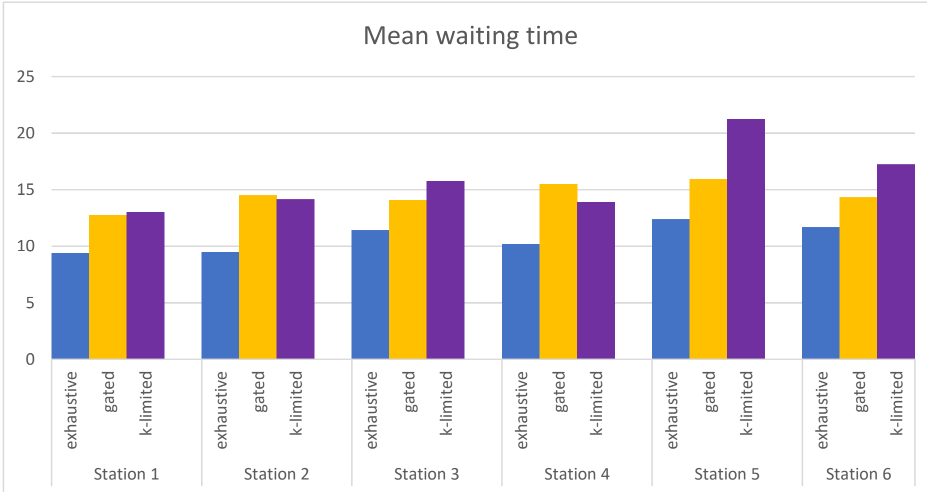

Figure 1: Mean Waiting Time, grouped for each station, per service strategy

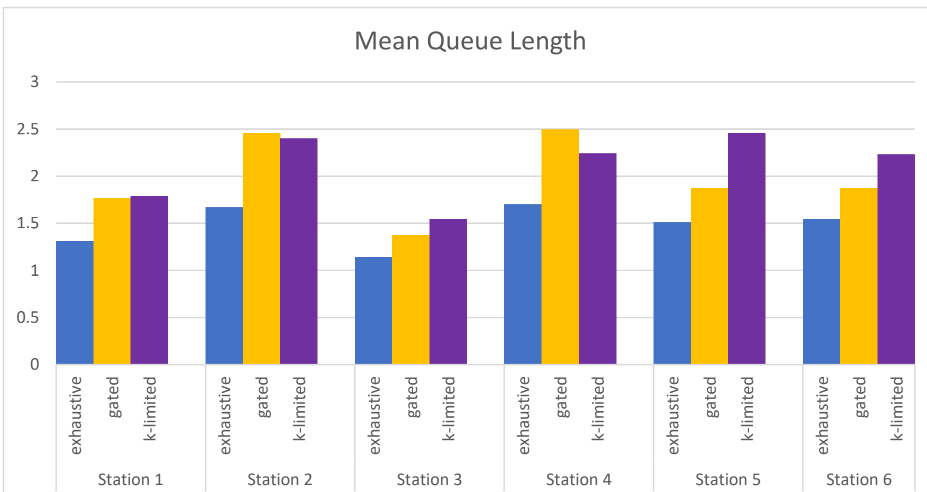

Figure 2: Mean Queue Length, grouped for each station, per service strategy

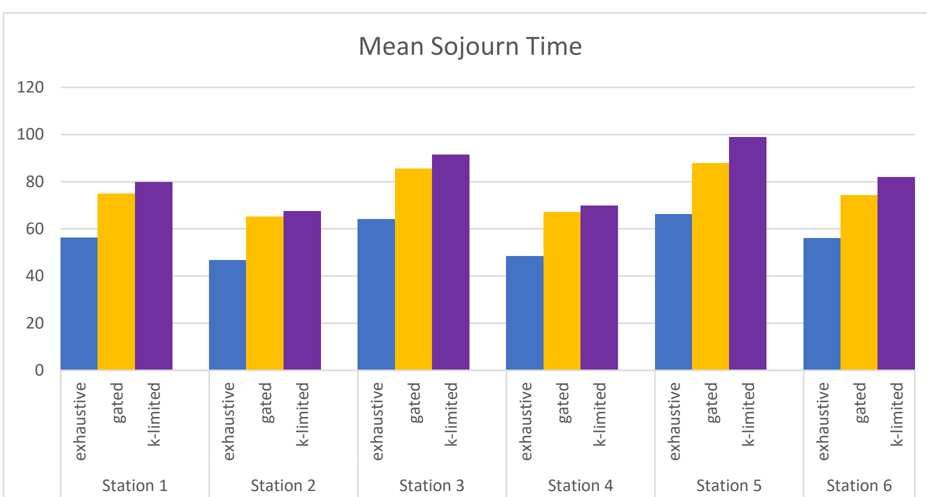

Figure 3: Mean Sojourn Time, grouped for each station, per service strategy

Conclusion

The performance measures in tables 1,3 and 5 demonstrate valuable insights. First we can see that stations' λ and $\mathbb{E}[B]$ values directly affect the steady-state waiting time and sojourn time which is expected. As λ decreases (holding $\mathbb{E}[B]$ constant) which means the arrivals are more frequent and we have the same service rate, we observe higher waiting times and higher sojourn times for that station's customers. However, the steady-state waiting time and the queue length isn't always correlated, this might be because of the unbalanced transition probabilities. Another insight is that for every station, steady-state sojourn time may be a good indicator for how many different stations that a customer visits before departing from that station to outside of the system. In addition, we have observed that for the same service discipline the mean cycle times for different stations converge approximately to the same value in the steady state.

Based on the grouped bar charts, we can conclude that for the input file considered in our assignment, an Exhaustive strategy provides the best performance in terms of average waiting time, average queue length and average sojourn time. Gated and K-Limited strategies provide similar performance, although especially station 5 and 6 have both significantly longer average waiting times and longer average queue lengths with K-limited strategies. Thus, we would argue that a Gated strategy would be better than a K-Limited strategy for the considered input file. The bar charts also indicate a correlation between the waiting time and the queue length, although this is not always the case - station 5 has a higher waiting time than station 4 with exhaustive and gated strategies, while the queue length is shorter.

References

[1] Moshe Sidi, Hanoch Levy, and Steve W. Fuhrmann. "A queueing network with a single cyclically roving server". en. In: *Queueing Systems* 11.1-2 (Mar. 1992), pp. 121–144. ISSN: 0257-0130, 1572-9443. DOI: `10.1007/BF01159291`. URL: `http://link.springer.com/10.1007/BF01159291` (visited on 03/13/2021).

A Graphs

A.1 Steady State Graphs

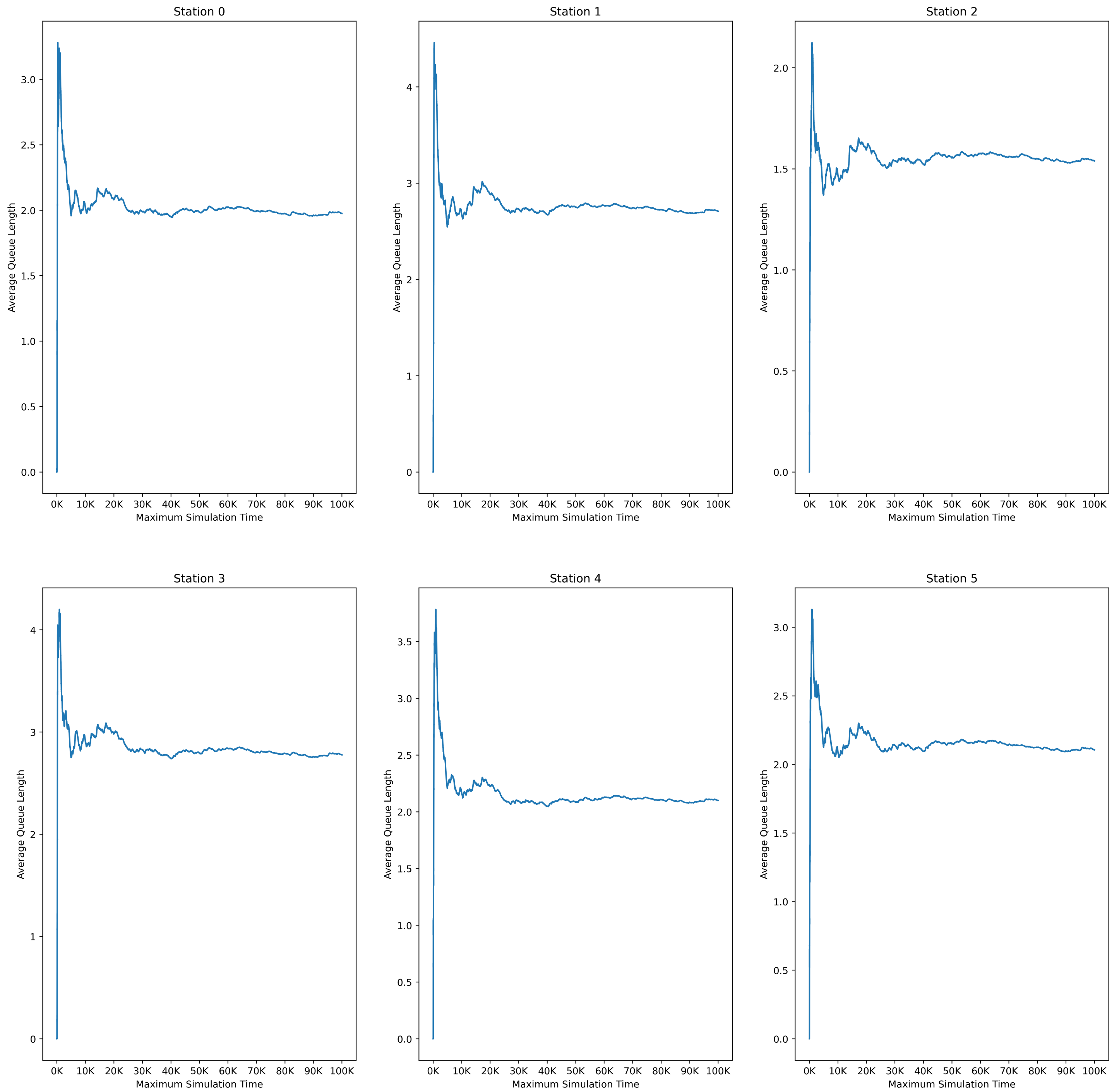

Figure 4: Average Queue Length vs Maximum Simulation Time for Gated Service Discipline

A.2 Unstable System Graphs

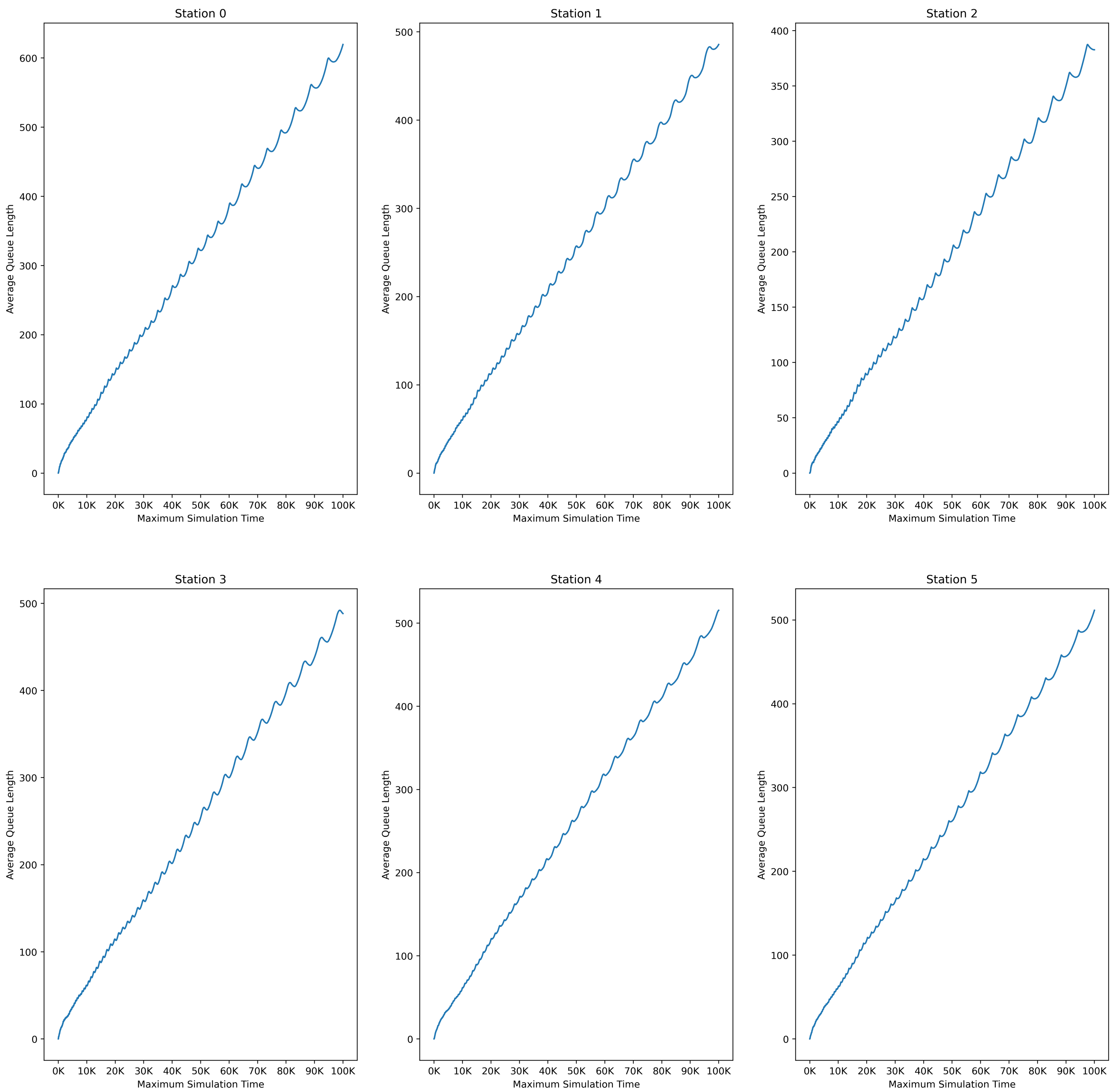

Figure 5: Average Queue Length vs Maximum Simulation Time for Exhaustive Service Discipline ($\rho > 1$)

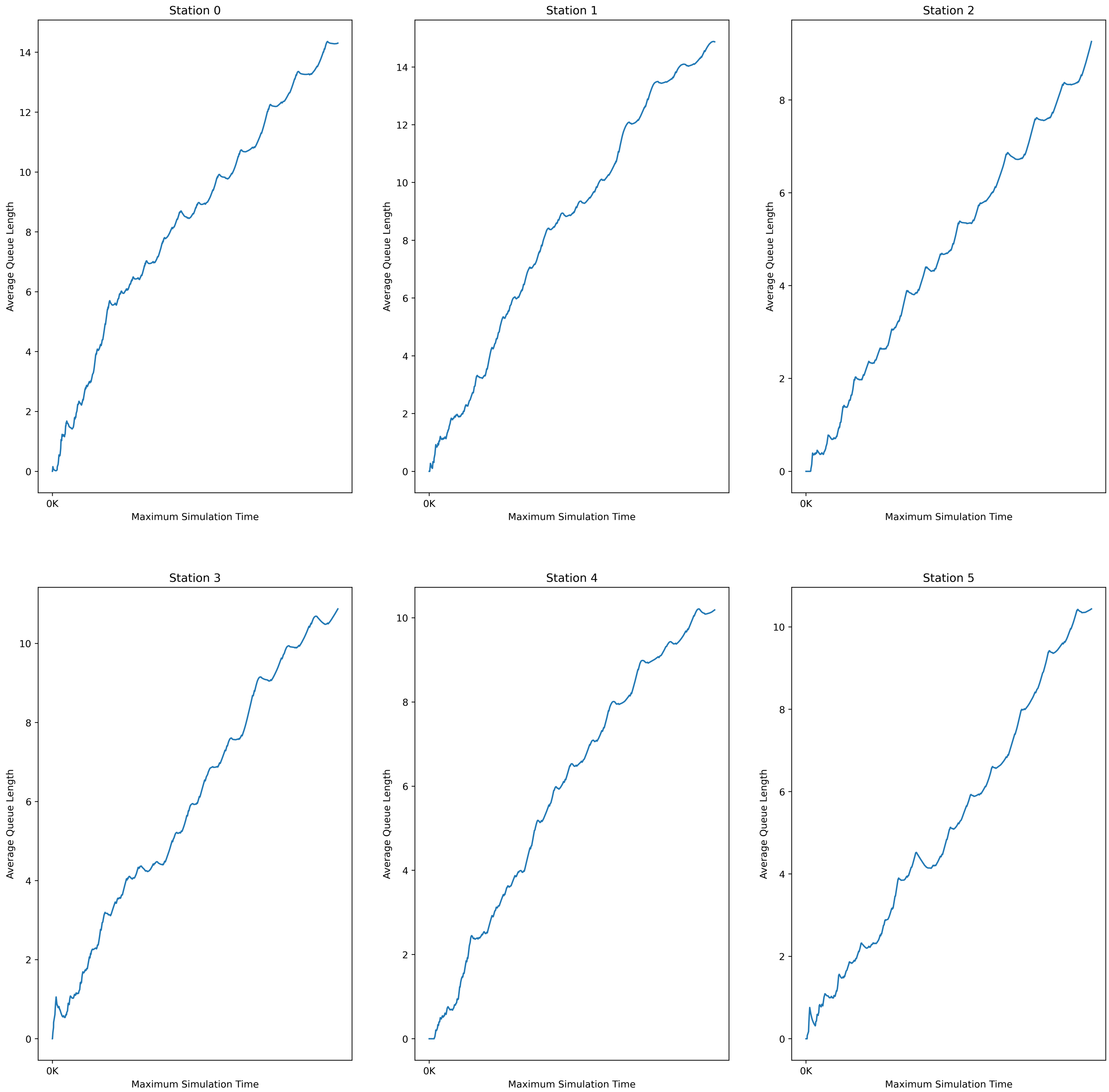

Figure 6: Average Queue Length vs Maximum Simulation Time for Gated Service Discipline ($\rho > 1$)

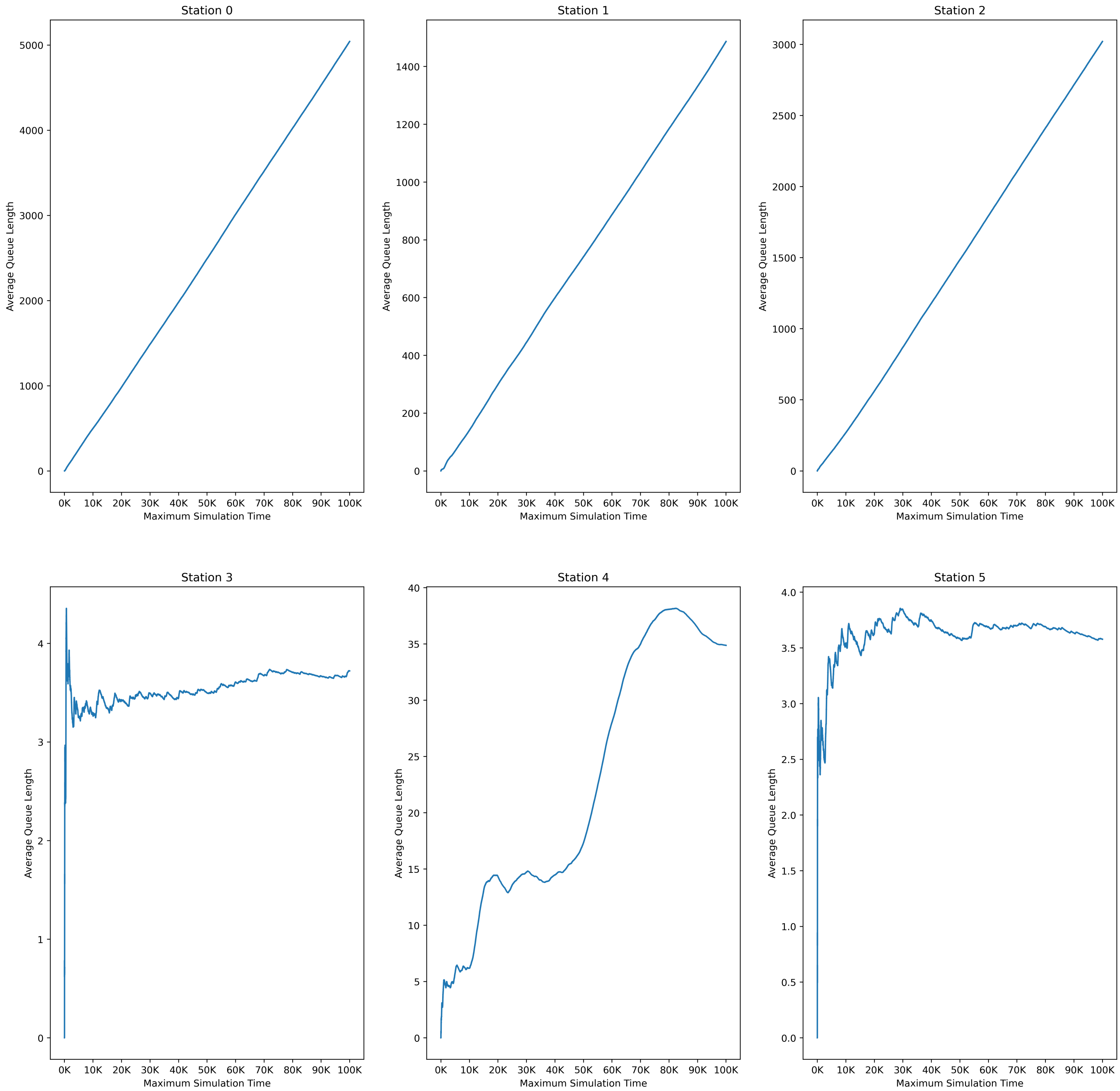

Figure 7: Average Queue Length vs Maximum Simulation Time for K-Limited Service Discipline ($\rho > 1$)

B Input file

Station	λ_i	$\mathbb{E}[B_i]$	$\mathbb{E}[R_i]$	k_i
1	0.04	0.2	1.8	8
2	0.09	0.8	2	9
3	0.01	0.2	1.4	6
4	0.08	0.7	1.2	9
5	0.01	0.7	0.8	6
6	0.04	0.1	1.2	7

Table 7: Input Variables

Station	1	2	3	4	5	6	Departure
1	0.14	0.14	0.14	0.16	0.16	0.11	0.15
2	0.14	0.14	0.11	0.14	0.11	0.16	0.20
3	0.12	0.15	0.12	0.12	0.15	0.12	0.22
4	0.14	0.11	0.11	0.17	0.11	0.11	0.25
5	0.18	0.12	0.15	0.12	0.15	0.12	0.13
6	0.15	0.15	0.10	0.10	0.12	0.12	0.26

Table 8: Input Variables - Transition Probabilities

B.1 input47.txt

```
0.04    0.09    0.01    0.08    0.01    0.04
0.2     0.8     0.2     0.7     0.7     0.1
1.8     2.      1.4     1.2     0.8     1.2
8.      9.      6.      9.      6.      7.
0.14    0.14    0.14    0.16    0.16    0.11
0.14    0.14    0.11    0.14    0.11    0.16
0.12    0.15    0.12    0.12    0.15    0.12
0.14    0.11    0.11    0.17    0.11    0.11
0.18    0.12    0.15    0.12    0.15    0.15
0.15    0.15    0.1     0.1     0.12    0.12
```

C Source code files

C.1 Simulator.py

```python
# Main class to run simulations of the network.
from datetime import datetime
import math
import numpy as np
import random
from Network import Network
import pandas as pd


def main(input_file):

    # Simulation parameters
    max_time = 40000
    warm_up_time = 5000
    nr_runs = 9
    serv_discipline = "EXHAUSTIVE"  # Set to "EXHAUSTIVE", "GATED" or "K-LIMITED"

    print(f"Running the simulation {nr_runs} times with {serv_discipline} strategy, with a max time of {max_time} and a warm up time of {warm_up_time}.")

    # Read the initial board layout from the board_file with given filename into a 2d list
    nr_stations, list_lambdas, list_expected_serv_times, list_expected_switch_times, \
        list_k_limited, list_2d_probabilities = read_input(input_file)

    # Initializing the average output variables
    average_queue_lengths = []
    mean_cycle_times = []
    mean_sojourn_times = []
    mean_waiting_times = []

    for i in range(nr_runs):

        # Randomly selecting a starting station
        starting_station = np.random.choice(np.arange(nr_stations))

        # Create network
        network = Network(serv_discipline, nr_stations, list_lambdas, list_expected_serv_times,
                          list_expected_switch_times, list_k_limited,
                          list_2d_probabilities, starting_station, warm_up_time)

        # Simulate network and get output variables
        waiting_times, cumulative_queue_lengths, sojourn_times, cycle_times , cumulative_queue_lengths_time, \
            waiting_times_time= simulate_network(network, max_time)

        # Calculating average queue lengths(Q_i) using cumulative queue lengths for the run
        average_queue_lengths.append([queue / max_time for queue in cumulative_queue_lengths])

        # Calculating mean cycle times(C_i) for the run
        mean_cycle_times.append([sum(i) / len(i) for i in cycle_times])

        # Calculating mean sojourn times(T_i) for the run
        mean_sojourn_times.append([sum(i) / len(i) for i in sojourn_times])

        # Calculating mean waiting times(W_i) for the run
        mean_waiting_times.append([sum(i) / len(i) for i in waiting_times])

        del network

    # Time to calculate the confidence intervals for all output variables for different stations
    # Initialize list of standard deviations
    standard_deviations = []

    # Initializing output table that is going to be used for exporting the results to excel file
    output_table_dictionary = {}

    for station in range(nr_stations):

        # Calculating the averages for output variables
        average_queue_lengths_for_station = [item[station] for item in average_queue_lengths]
        average_cycle_times_for_station = [item[station] for item in mean_cycle_times]
        average_sojourn_times_for_station = [item[station] for item in mean_sojourn_times]
        average_waiting_times_for_station = [item[station] for item in mean_waiting_times]

        # Calculating the standard deviation for output variables
        std_queue_length_for_station = round(np.std(average_queue_lengths_for_station), 4)
        std_cycle_time_for_station = round(np.std(average_cycle_times_for_station), 4)
        std_sojourn_time_for_station = round(np.std(average_sojourn_times_for_station), 4)
        std_waiting_time_for_station = round(np.std(average_waiting_times_for_station), 4)

        # Calculating the variance for output variables
        variance_queue_length_for_station = round(std_queue_length_for_station ** 2, 4)
        variance_cycle_time_for_station = round(std_cycle_time_for_station ** 2, 4)
        variance_sojourn_time_for_station = round(std_sojourn_time_for_station ** 2, 4)
        variance_waiting_time_for_station = round(std_waiting_time_for_station ** 2, 4)

        # Adding standard deviations to use it for estimating standard deviation on the required number of runs equation

        standard_deviations.extend([std_queue_length_for_station, std_cycle_time_for_station,
                                    std_sojourn_time_for_station, std_waiting_time_for_station])

        queue_length_lower, queue_length_upper, queue_length_mean = calc_conf_interval_list(average_queue_lengths_for_station, nr_runs)
        cycle_time_lower, cycle_time_upper, cycle_time_mean = calc_conf_interval_list(average_cycle_times_for_station, nr_runs)
        sojourn_time_lower, sojourn_time_upper, sojourn_time_mean = calc_conf_interval_list(average_sojourn_times_for_station, nr_runs)
        waiting_time_lower, waiting_time_upper, waiting_time_mean = calc_conf_interval_list(average_waiting_times_for_station, nr_runs)

        output_table_dictionary[station] = [waiting_time_mean, variance_waiting_time_for_station,
                                            queue_length_mean, variance_queue_length_for_station,
                                            cycle_time_mean, variance_cycle_time_for_station,
                                            sojourn_time_mean, variance_sojourn_time_for_station,
                                            waiting_time_lower, waiting_time_upper,
                                            queue_length_lower, queue_length_upper,
                                            cycle_time_lower, cycle_time_upper,
                                            sojourn_time_lower, sojourn_time_upper]

        print(f"Average queue length for station {station} is {queue_length_mean} on average, standard_deviation = {std_queue_length_for_station}, lower = {queue_length_lower}, upper = {queue_length_upper}")
        print(f"Average cycle time for station {station} is {cycle_time_mean} on average, standard_deviation = {
```

```python
            std_cycle_time_for_station}, lower = {cycle_time_lower}, upper = {cycle_time_upper}")
        print(f"Average sojourn time for station {station} is {sojourn_time_mean} on average, standard_deviation = {std_sojourn_time_for_station}, lower = {sojourn_time_lower}, upper = {sojourn_time_upper}")
        print(f"Average waiting time for station {station} is {waiting_time_mean} on average, standard_deviation = {std_waiting_time_for_station}, lower = {waiting_time_lower}, upper = {waiting_time_upper}")
        print()

    # Convert the results dictionary to dataframe and export it to excel file
    df = pd.DataFrame.from_dict(output_table_dictionary, orient='index',columns=['w-mean', 'w-variance', 'q-mean', 'q-variance',
                                                                                  'c-mean', 'c-variance', 't-mean', 't-variance',
                                                                                  'w-lower', 'w-upper', 'q-lower', 'q-upper',
                                                                                  'c-lower', 'c-upper', 't-lower', 't-upper'])
    df.to_excel("output %s.xlsx" %(serv_discipline))

    # Time to validate theoretical results from papers with output variables
    # Initializing a coefficient matrix to be used in solving linear equations for finding gamma values
    coefficient_matrix = np.zeros((nr_stations, nr_stations))

    for i in range(nr_stations):

        # For each linear equation the coefficients are extracted
        coefficient_row = [-item[i] for item in list_2d_probabilities]
        coefficient_row[i] += 1

        # For each linear equation extracted coefficients are added to the coefficient matrix
        coefficient_matrix[i] = coefficient_row

    # Extracting the constant terms for the linear equation system
    B = np.array(list_lambdas)

    # Solving the system of linear equations and finding the gamma values
    gamma_solution = np.linalg.solve(coefficient_matrix, B)

    # Calculating the network utilisation with given formula
    network_utilisation = round(sum(np.multiply(gamma_solution, list_expected_serv_times)), 4)

    # Calculating theoretical mean cycle time with given formula
    theoretical_mean_cycle_time = round(sum(list_expected_switch_times) / (1 - network_utilisation), 4)

    print(f"Theoretical network utilisation: {network_utilisation}.")
    print(f"Theoretical mean cycle time: {theoretical_mean_cycle_time}.")


# Method to read the input file with a given name
# Returns several input variables: nr_stations, list_lambdas, list_expected_serv_times, list_expected_switch_times
# list_k_limited,, list_2d_probabilities (INCLUDING probability to leave system)
def read_input(input_file):
    # Initialise line/wordcount
    linecount = 0
    wordcount = 0

    # Initialise all input variables to return, except the 2d list
    nr_stations = 0
    list_lambdas = []
    list_expected_serv_times = []
    list_expected_switch_times = []
    list_k_limited = []

    # Initialise variable to track probability that current station would leave the system
    prob_leave_system = 1.0

    # Open the file with given filename
    with open(input_file, 'r') as file:

        # Iterate over each line
        for line in file:

            # Get the number of stations from the line length
            if linecount == 0:
                nr_stations = len(line.split())

                # Initialise the 2d list based on the nr_stations, with 1 extra column used for prob to leave system
                list_2d_probabilities = [[0] * (nr_stations + 1) for i in range(nr_stations)]

            # Iterate over each word in the line
            for word in line.split():

                # Convert the word to a float
                number = float(word)

                # Line 4 and beyond contains the probabilities to switch from station i to station j
                # Store in linecount-4 as that stores station 1 in list[0]
                # list[0][0] until [0][5] are used for probabilities to switch to station 1 until 6
                # list[0][6] is used for probability to leave system from station 1
                if linecount > 3:
                    prob_leave_system -= number
                    list_2d_probabilities[linecount-4][wordcount] = number

                # Line 0 contains the poisson process rates
                elif linecount == 0:
                    list_lambdas.append(number)

                # Line 1 contains the expected service times
                elif linecount == 1:
                    list_expected_serv_times.append(number)

                # Line 2 contains the expected switch times
                elif linecount == 2:
                    list_expected_switch_times.append(number)

                # Line 3 contains the max number of customers served per station during one visit period for k-limited
                elif linecount == 3:
                    list_k_limited.append(number)

                # Iterate wordcount
                wordcount += 1

            # After finishing a line with linenumber > 3, set probability to leave system for current station
            # and reset the prob_leave_system
            if linecount > 3:
                list_2d_probabilities[linecount - 4][nr_stations] = round(prob_leave_system, 2)
                prob_leave_system = 1.0
```

```
            # Reset wordcount and iterate linecount
            wordcount = 0
            linecount += 1

    # Return all collected input variables
    return(nr_stations, list_lambdas, list_expected_serv_times, list_expected_switch_times,
           list_k_limited, list_2d_probabilities)


# Method to simulate the system with given parameters
def simulate_network(network, max_time):

    nr_stations = len(network.stations)

    # Initializing the time of previous visit of the server to each station
    previous_time_at_station = [0] * nr_stations

    # Initializing result variables for the run
    waiting_times = [[] for i in range(nr_stations)]
    waiting_times_time = [[(0,0)] for i in range(nr_stations)] #Time Stamped Waiting Times
    cumulative_queue_lengths = [0] * nr_stations
    cumulative_queue_lengths_time = [[(0,0)] for i in range(nr_stations)] #Time Stamped Cumulative Queue Lengths
    sojourn_times = [[] for i in range(nr_stations)]
    cycle_times = [[] for i in range(nr_stations)]

    while network.current_time < max_time:

        # get earliest event from the front of the FES
        current_event = network.fes.next()

        # Update time based on current_event
        old_time = network.current_time
        network.current_time = current_event.time

        # Wait until warm-up finishes for steady state variables
        if network.current_time > network.warm_up_time:

            # Update Cumulative Queue Lengths for each station
            for station in range(nr_stations):

                t = network.current_time - old_time

                cumulative_queue_lengths[station] += t * network.stations[station].queue.size()

                # Time Stamped Cumulative Queue Lengths(for plotting to decide on max_simulation time)
                cumulative_queue_lengths_time[station].append((network.current_time,
                                                               cumulative_queue_lengths[station]))

        if current_event.typ == "ARRIVAL":
            # Handling the current arrival event
            network.handle_arrival(current_event)

            # Schedule the first departure if it is not scheduled before
            if current_event.queue == network.server.current_station and not network.scheduled_first_departure:

                network.schedule_departure(network.server.current_station)

                network.scheduled_first_departure = True

        elif current_event.typ == "DEPARTURE":
            # Retrieving the departing customer and the queue
            current_queue = current_event.queue
            current_customer = network.stations[current_queue].queue.queue[0]

            # Remove the customer from the corresponding queue
            network.stations[current_queue].queue.queue.pop(0)

            # Increase served_at_current_station tracking variable
            network.server.served_at_current_station += 1

            # Decide where the customer will go next(including departure) with given probabilities
            next_queue = random.choices(range(nr_stations+1),
                                        network.stations[current_queue].transition_prob)[0]

            # Wait until warm-up finishes for steady state variables
            if network.current_time > network.warm_up_time:
                # Calculate the waiting time for that customer in that station and update the results
                waiting_time = network.current_time-(current_event.service_time + current_customer.arrival_time_station)
                waiting_times[current_queue].append(waiting_time)

                # Time stamped average waiting time (for plotting to decide on max_simulation time)
                waiting_times_time[current_queue].append((network.current_time, sum(waiting_times[current_queue]) / len(waiting_times[current_queue])))

            # If the customer departs from the system, update result variable
            if next_queue == len(network.stations):
                # Wait until warm-up finishes for steady state variables
                if network.current_time > network.warm_up_time:
                    # Calculate the sojourn time for the departing customer
                    departure_time = current_event.time
                    sojourn_times[current_queue].append(departure_time - current_customer.arrival_time_system)

            else:
                # If the customer does not depart the system, they move internally
                # Update the customers arrival_time_station variable to the current_time
                current_customer.arrival_time_station = network.current_time
                # Immediately add the customer to the new (or same) queue
                network.stations[next_queue].queue.add_customer(current_customer)

            # If EXHAUSTIVE strategy is used and after handling internal arrivals the queue is not empty
            # schedule another departure for the current queue
            if network.server.strategy == "EXHAUSTIVE" and network.stations[current_queue].queue.size() >= 1:
                network.schedule_departure(current_queue)

            # If K-LIMITED strategy is used and after handling internal arrivals the queue is not empty and
            # k[i] customers have not been served in current visit to station i schedule another departure
            # for the current queue
            elif (network.server.strategy == "K-LIMITED" and
                  not network.server.k_i_customers_served() and
                  network.stations[current_queue].queue.size() >= 1
                  ):
                network.schedule_departure(current_queue)

            # If GATED strategy is used and GATED customers have not been served in current visit to station i
            # schedule another departure for the current queue
```

```
            elif network.server.strategy == "GATED" and not network.server.gated_target_customers_served():
                network.schedule_departure(current_queue)

        # While loop to move the station if it should switch.
        # While instead of if, to make sure server switches again if queue of new station was empty
        while network.should_switch_station():

            # Get switchover time from the switch time arrays and compute time after switching
            switchover_time = network.server.switch_time[network.server.current_station]
            time_done_switching = network.current_time + switchover_time

            # Peek at the event in front of FES, and handle it if it is earlier than time_done_switching
            # While loop to ensure we handle any events happening before time_done_switching
            while network.fes.peek().time < time_done_switching:
                # Get the event in front of the FES
                current_event = network.fes.next()

                # Throw exception if non-arrival event was found. Should never happen.
                if current_event.typ != "ARRIVAL":
                    raise Exception('There should not be a non-arrival event while the server is switching. '
                                    'The event type while switching was: {}'.format(current_event.typ))
                else:
                    # Handle the arrival event in front
                    network.handle_arrival(current_event)

            # Move server to new station
            network.server.move_station()

            # If using Gated strategy, update gated_target based on queue size of new station
            new_station = network.server.current_station
            if network.server.strategy == "GATED":
                network.server.gated_target = network.stations[new_station].queue.size()

            # Update current_time to (start_of_switch_time + switchover time)
            network.current_time = time_done_switching

            # Store the time of server's arrival to the new station to use in calculating cycle time statistics
            if previous_time_at_station[new_station] != 0 or new_station == network.server.starting_station:

                time_between_station_visits = network.current_time - previous_time_at_station[new_station]
                previous_time_at_station[new_station] = network.current_time

                # Wait until warm-up finishes for steady state variables
                if network.current_time > network.warm_up_time:
                    cycle_times[new_station].append(time_between_station_visits)

            else:
                previous_time_at_station[new_station] = network.current_time

            # If queue is not empty, schedule the first departure event for new station
            if network.stations[new_station].queue.size() >= 1:
                network.schedule_departure(new_station)
                network.scheduled_first_departure = True

    return waiting_times, cumulative_queue_lengths, sojourn_times, cycle_times, cumulative_queue_lengths_time, waiting_times_time


# Method to calculate confidence intervals of result variables
# Note that we use 1.96, thus 95% confidence interval
def calc_conf_interval_list(lst, nr_runs):

    sd = np.std(lst)
    m = np.mean(lst)

    halfwidth = 1.96 * sd/math.sqrt(nr_runs)
    lower = m - halfwidth
    upper = m + halfwidth

    return round(lower, 4), round(upper, 4), round(m, 4)


if __name__ == "__main__":

    # Print the starting time of the simulation
    dateTimeObj = datetime.now()
    print("Current time is: ")
    print(dateTimeObj.hour, ':', dateTimeObj.minute, ':', dateTimeObj.second)

    # Run simulation with given input file
    main("input47.txt")

    # Print the ending time of the simulation
    dateTimeObj = datetime.now()
    print("Finishing time is: ")
    print(dateTimeObj.hour, ':', dateTimeObj.minute, ':', dateTimeObj.second)
```

C.2 Network.py

```
# Object class for a network.
# Keeps track of core variables of the simulation, such as the current_time and if scheduled_first_departure.
# At initialization, creates a Server, a list of nr_stations Stations, and a FES.
# Also schedules the first arrival events for each station.
# Has methods to schedule departures/arrivals, to handle an arrival and to check if the server should switch stations.

from Server import Server
from Station import Station
from FES import FES
from Event import Event
from Customer import Customer

import numpy as np


class Network:

    def __init__(self, strategy, nr_stations, list_lambdas, list_exp_serv_times, list_exp_switch_times,
                 list_k_limited, list_2d_prob, starting_station, warm_up_time):

        # Boolean to detect if the first departure of the network is scheduled or not
        self.scheduled_first_departure = False
        self.stations = []
        self.current_time = 0
        self.warm_up_time = warm_up_time  # Until the warm-up time is reached the output variables are not updated

```

```python
        # Initializing the server
        self.server = Server(strategy, nr_stations,
                             list_exp_serv_times, list_exp_switch_times,
                             list_k_limited, starting_station)

        self.lambdas = list_lambdas

        # Initializing FES to store all events
        self.fes = FES()
        self.service_times = list_exp_serv_times

        for station in range(nr_stations):

            # Adding the stations with their respective attributes to the list
            self.stations.append(Station(list_lambdas[station], list_exp_serv_times[station], list_2d_prob[station]))

            # Scheduling the first arrival to each station
            self.schedule_arrival(station)

    # Function to schedule a departure event for given station
    def schedule_departure(self, station_nr):

        # Compute new service time
        service_time = np.random.exponential(self.service_times[station_nr])

        # Schedule departure event at current_time + service_time
        event = Event("DEPARTURE", self.current_time + service_time, station_nr, service_time)

        # Add scheduled event to the FES
        self.fes.add(event)

    # Function to schedule an arrival event for given station
    def schedule_arrival(self, station_nr):

        # Compute new interarrival time
        interarrival_time = np.random.exponential(1 / self.lambdas[station_nr])

        # Schedule arrival event at current_time + interarrival_time
        event = Event("ARRIVAL", self.current_time + interarrival_time, station_nr, interarrival_time)

        # Add scheduled event to the FES
        self.fes.add(event)

    # Function that returns a boolean specifying if the server should switch its station
    def should_switch_station(self):

        # First, check if the queue of the current station is empty
        current_station = self.server.current_station

        if self.stations[current_station].queue.size() < 1:

            # If the queue is empty, should switch to next station
            return True

        # Extra checks for K-Limited or Gated strategy
        # If k_i customers are served at station i, should switch to next station

        if self.server.strategy == "K-LIMITED":

            if self.server.k_i_customers_served():

                return True

        # If nr of customers served is equal to target set for gated strategy, should switch to next station
        elif self.server.strategy == "GATED":

            if self.server.gated_target_customers_served():

                return True

        # Should not switch in any other case
        return False

    # Function to handle an arrival event, by creating and adding a customer to the queue and scheduling a new arrival
    def handle_arrival(self, event):

        # Create a new customer
        customer = Customer(self.current_time, event.queue)

        # Add customer to the corresponding queue
        self.stations[event.queue].queue.add_customer(customer)

        # Lastly, schedule a new arrival in the queue
        self.schedule_arrival(event.queue)
```

C.3 Server.py

```python
# Object class for a server.
# Keeps track of the current station of the Server, and the number of customers served at the current station.
# Also keeps track of the target number of customers to serve for the gated server discipline.
# Contains methods to check if k-limited or gated server discipline switch conditions have been met,
# and a method to move the Server to the next station

class Server:

    def __init__(self, strategy, nr_stations, array_exp_serv_time,
                 array_exp_switch_time, array_k_limited, starting_station):

        # Variable used to track nr customers served at current station, resets after moving
        self.served_at_current_station = 0

        # The number of customers present when the server arrives at that station when GATED is the strategy
        # Initialized as 1 because we want to handle the first arriving customer to the starting station
        # before switching to the next station otherwise we can observe departure event during the first switch
        # of the server
        self.gated_target = 1

        self.strategy = strategy
        self.k_parameters = array_k_limited
        self.nr_stations = nr_stations
        self.service_time = array_exp_serv_time
        self.switch_time = array_exp_switch_time
        self.current_station = starting_station
        self.starting_station = starting_station
```

```python

    # Function that returns if k_i customers have been served in current visit to station i
    def k_i_customers_served(self):
        k_i = self.k_parameters[self.current_station]
        return self.served_at_current_station >= k_i

    # Function that returns if gated_target customers have been served in current visit to station
    def gated_target_customers_served(self):
        return self.served_at_current_station >= self.gated_target

    # Function to move the server to the next station
    # Also reset served_at_current_station
    def move_station(self):
        self.current_station = (self.current_station + 1) % self.nr_stations
        self.served_at_current_station = 0
```

C.4 Station.py

```python
# Object class for a station

from Queue import Queue


class Station:

    def __init__(self, arrival_rate, service_rate, list_prob):

        self.arrival_rate = arrival_rate
        self.service_rate = service_rate
        self.transition_prob = list_prob
        self.queue = Queue()
```

C.5 Queue.py

```python
# Queue class based on the SwitchingServerQueue example provided by the course.
# Keeps track of the customers at a certain Station.

class Queue:

    def __init__(self):
        self.queue = []

    def size(self):
        return len(self.queue)

    def add_customer(self, customer):
        self.queue.append(customer)

    # Function to remove the selected customer from the queue
    def pop(self, i):
        self.queue.pop(i)
```

C.6 Customer.py

```python
# Object class for a customer.
# Keeps track of the arrival time in the system, arrival time in the current station,
# and the original queue in which the customer entered the system.

class Customer:

    def __init__(self, arr, queue):
        self.arrival_time_system = arr
        self.arrival_time_station = arr
        self.queue = queue

    def __str__(self):
        return "Customer arrived at " + str(self.arrival_time_system) + " at queue " + str(self.queue)
```

C.7 Event.py

```python
# Event class based on the SwitchingServerQueue example provided by the course.
# Contains the type of event, the time of the event and the queue for which the event is.

class Event:

    def __init__(self, typ, time, queue, service_or_interarrival_time):

        self.typ = typ  # type 1: ARRIVAL, type 2: "DEPARTURE"
        self.time = time  # real positive number
        self.queue = queue  # 0, 1, 2, 3, 4, 5 based on station number

        if self.typ == "ARRIVAL":

            self.interarrival_time = service_or_interarrival_time

        elif self.typ == "DEPARTURE":

            self.service_time = service_or_interarrival_time

    def __lt__(self, other):
        return self.time < other.time

    def __str__(self):
        return self.typ + " of new customer " + ' at t = ' + str(self.time)
```

C.8 FES.py

```python
# FES class based on the SwitchingServerQueue example provided by the course.
# Keeps track of the arrival and departure events, in order of time the events happen.
import heapq


class FES:
    def __init__(self):
        self.events = []

    def add(self, event):
        heapq.heappush(self.events, event)
        # rewrite this to insert an event at a certain time

    def next(self):
        return heapq.heappop(self.events)
```

```python

    def is_empty(self):
        return len(self.events) == 0

    def get_length(self):
        return len(self.events)

    def peek(self):
        return self.events[0]

    def __str__(self):

        s = ''
        sorted_events = sorted(self.events)
        for e in sorted_events:
            s += str(e) + '\n'
        return s
```

D Workload Distribution

	Code	**Report**	**Total**
Robin	50%	50%	50%
Muhammed	50%	50%	50%