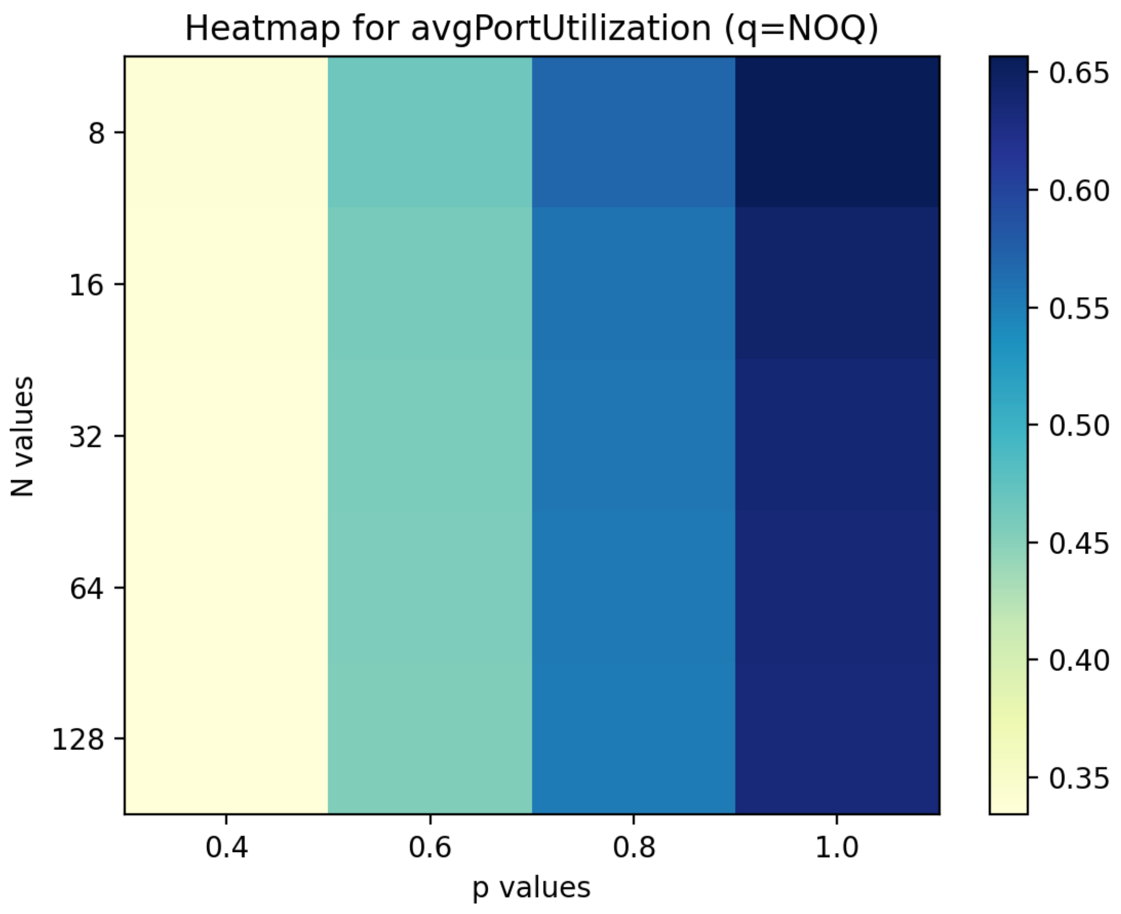
Assignment 3 - Report

The `simulate.py` file runs all the methods on all possible configurations and `outputs` a directory with all the outputs. This data is then used to create heat maps to understand the effect of different parameters on the `avgPacketDelay` and `avgPortUtilization`.

Terminology used in this report:

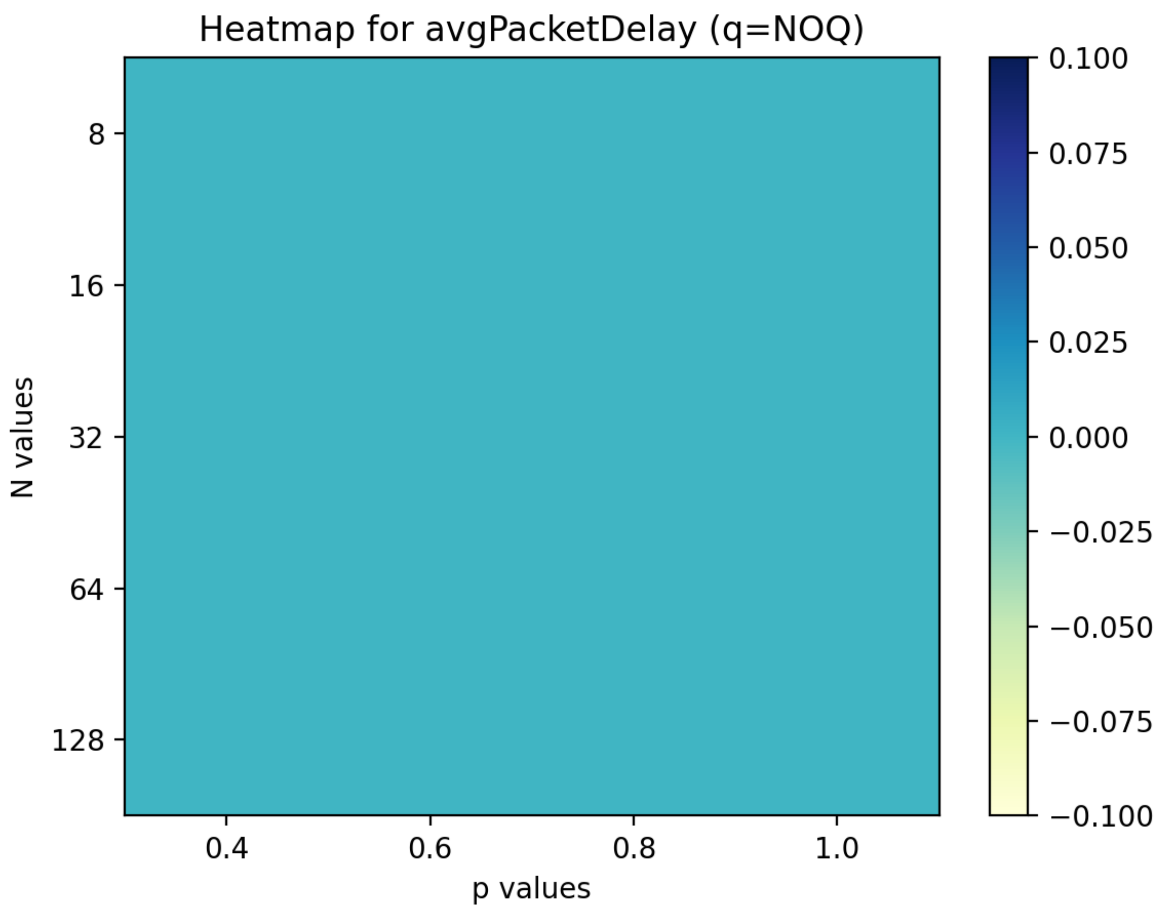
- `N`: Number of I/O ports
- `p`: Packet generation probability at each port per slot
- `B`: Buffer capacity (20 for all simulations)

1. NOQ (No queueing)



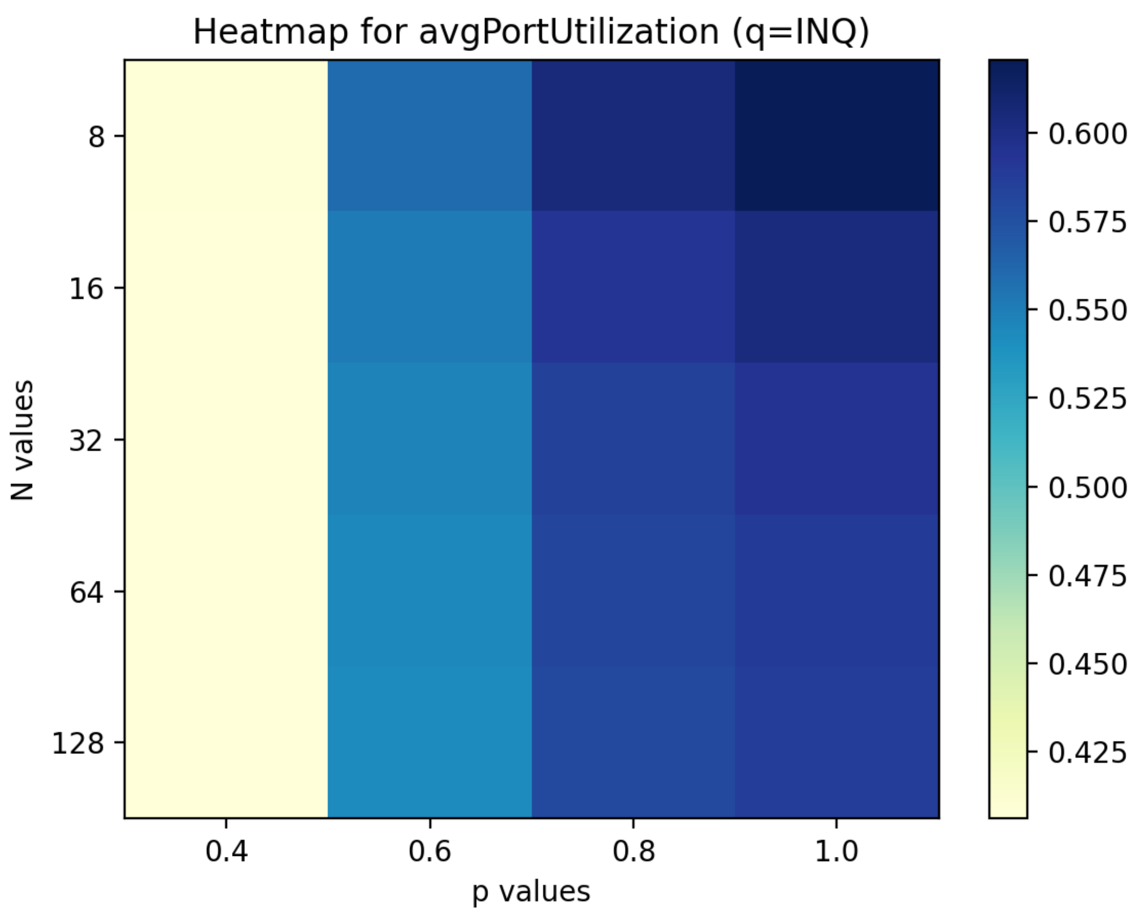
As shown here, the port utilisation does not really vary very heavily with the number of ports. However, it is noticed that the utilisation is higher with lesser number of ports. This can be attributed the lower competition among the ports over time. For example, if `N = 1`, and the `packetGenerationProbability = 1.0`, the port will always be chosen and the utilisation will be `1.0`.

The utilisation varies highly with the `packetGenerationProbability` as simulated. Closer to `1.0`, the average port utilisation comes out to be around `0.65` which is very close to the theoretical value predicted *i.e.* `0.63`. Towards the lower end of `p = 0.4`, the utilisation becomes `0.35`, which is expected since the ports will be more empty without any buffers if the expected number of packets generated is lower.



As it is shown here, the `avgPacketDelay` for `NOQ` is always `0` since it chooses one packet per slot per output port and transmits it in the same slot.
∴ the transmission time is always equal to the generation time. Since there is no buffer, there are no packets stored from old slots.

2. INQ (Only input queueing)

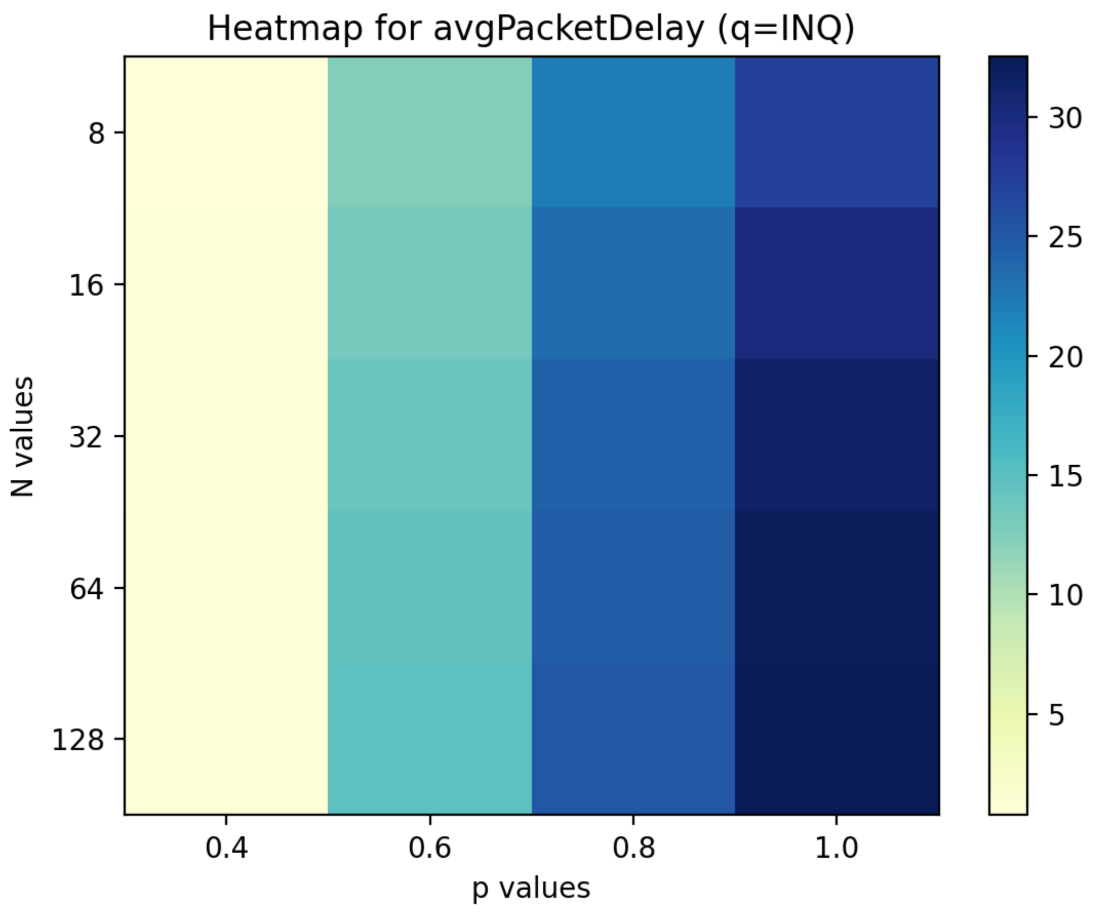


A similar but stronger trend is observed here as compared to `NOQ`. Although the `INQ` algorithm performs better than `NOQ` at high values of `N` and `p`, it starts emulating `NOQ` for very low values of `N` and high values of `p` in terms of utilisation. The utilisation ranges from `0.4 - 0.65`. This is because at high values of `p`, the buffers start getting filled quicker and packets start getting dropped. For further analysis, this requires more metrics like

- `avgPacketDropProbability`: Calculated solely based on `p`, before it gets added to the queue
- `avgBufferUtilisation`: Calculated at each time slot averaged over all the input ports
  The buffer does not make any difference at the output ports since only one packet is selected every slot.

Upon further analysis, `avgPacketDropProbability` comes out to be approximately `0.2 - 0.4` (for both `INQ` and `NOQ`) and `avgBufferUtilisation` comes out to be `16 - 19` for `B = 20` depending on different values of `N` and `p`. We can therefore infer from this that the buffer is almost always full and the extra packets generated over time are dropped and the utilisation is a result of that *i.e* the same. The only thing different in these two cases effectively is the contention resolving method, and that is not related to the output port utilisation.

This simulation cannot be compared with the theoretical predictions since we assume a Poisson process there and here it is Gaussian.



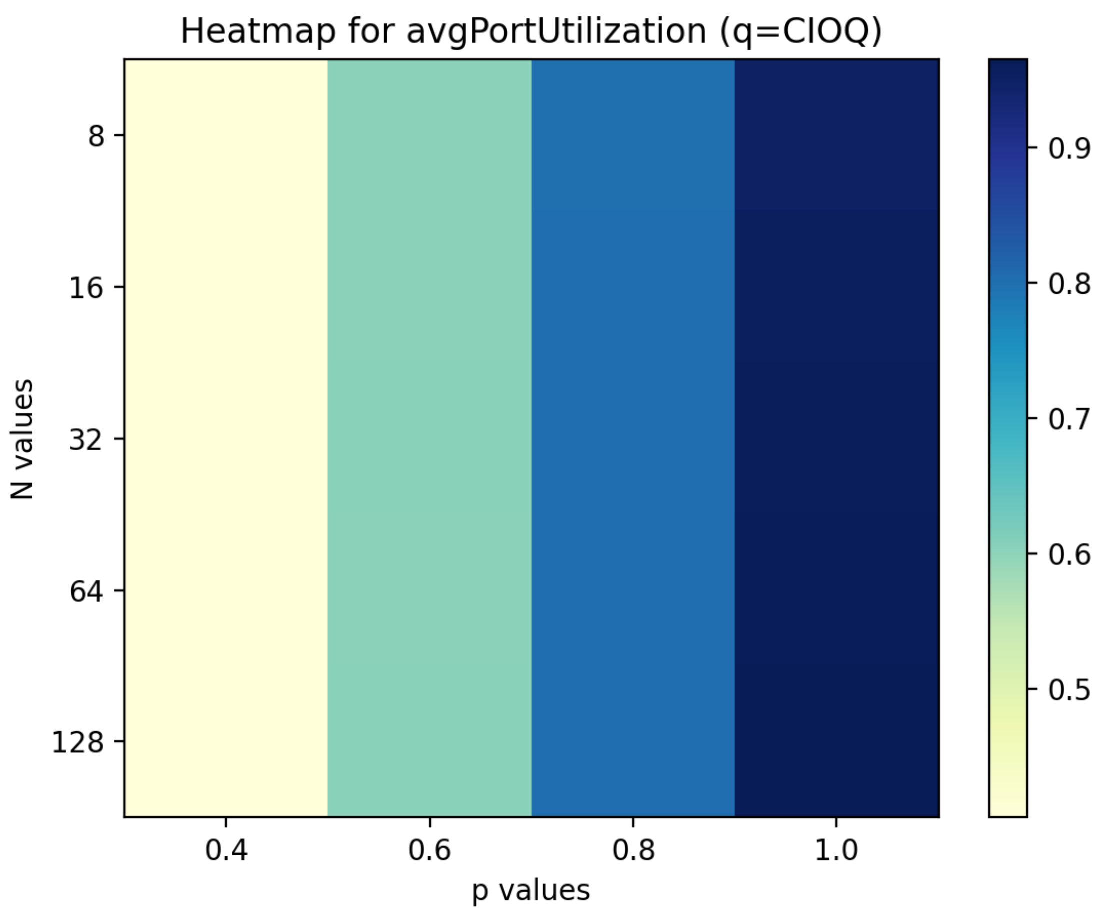
The `avgPacketDelay` is not similar to `NOQ` since the buffer allows for the packets to get queued and processed later. This grows with `N` and `p` as expected in their own gradients. The increase with `N` is not as stark as with `p`, since the contention resolution is simply choosing the port with a lower port number. Hence, the ports with higher port numbers are more likely to be queued for a longer time.

3. CIOQ (Combined input/output queueing)

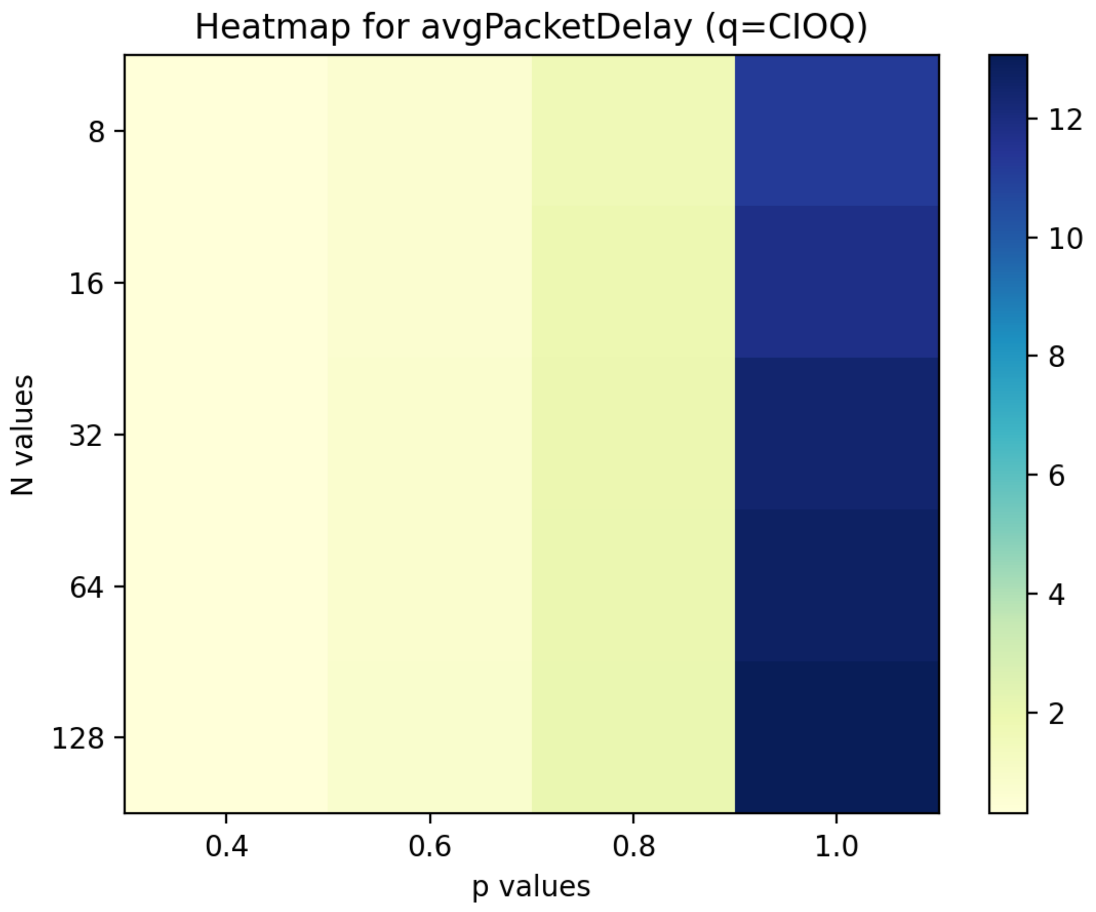
The variable parameters for `CIOQ` are (along with `N` and `p`):

- `L`: lookup
- `K`: backplane speed factor

Since the heat maps can only represent a 2D space, the results have been averaged out over all possible values of `L` and `K`. The relative analysis is still valid however, since `INQ` is simply `CIOQ` with `L = K = 1`.
∴ the `CIOQ` algorithm with higher values of `L` and `K` is only an improvement. The maximum analysis can be done for a theoretical understanding.



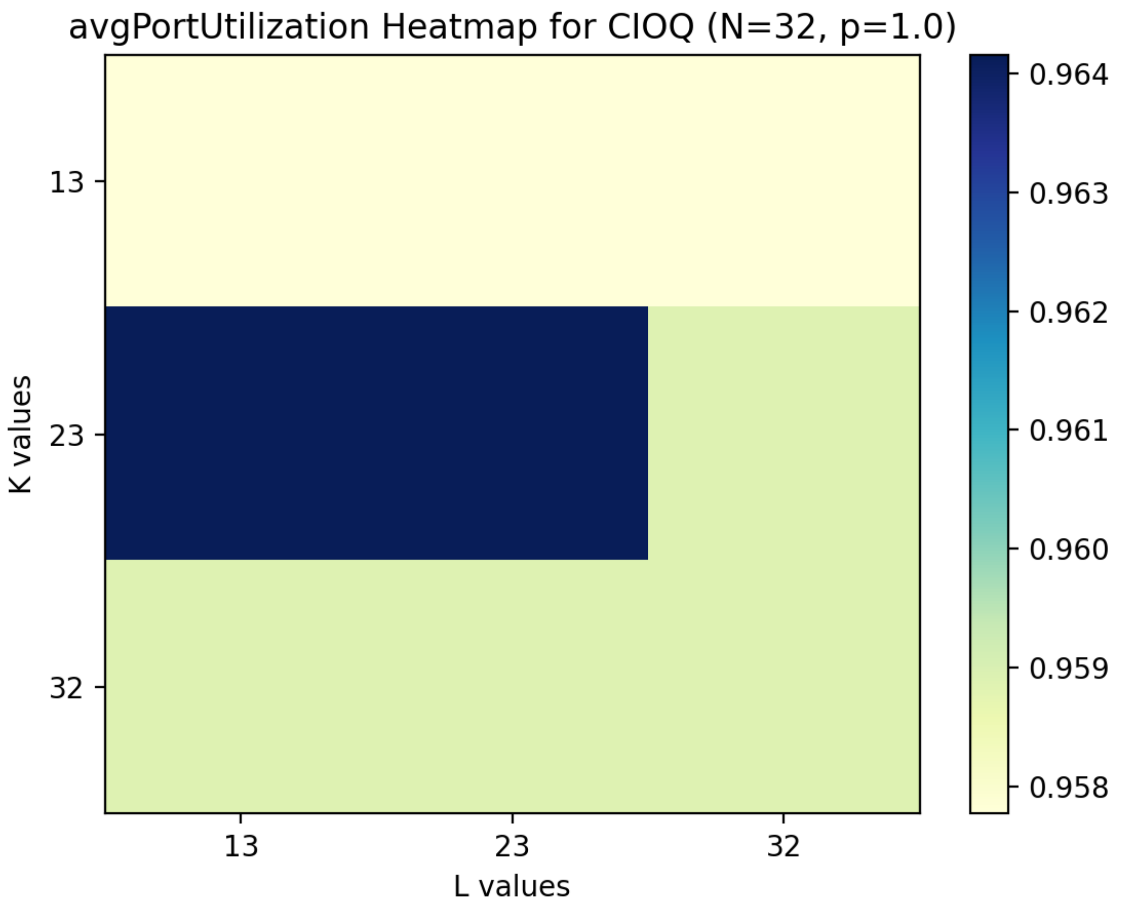
The port utilisation has very little variation with `N`, but very high direct dependence on `p`. This of course, varies highly with `L` and `K` but this heat map glosses over that detail. The algorithm as simulated, is much better performing than `INQ` and `NOQ` for all given values of `N` and `p`. The utilisation ranges from `0.5 - 0.9`, which is almost the theoretically calculated value of `1.0` at very high values of `N` and `p`.

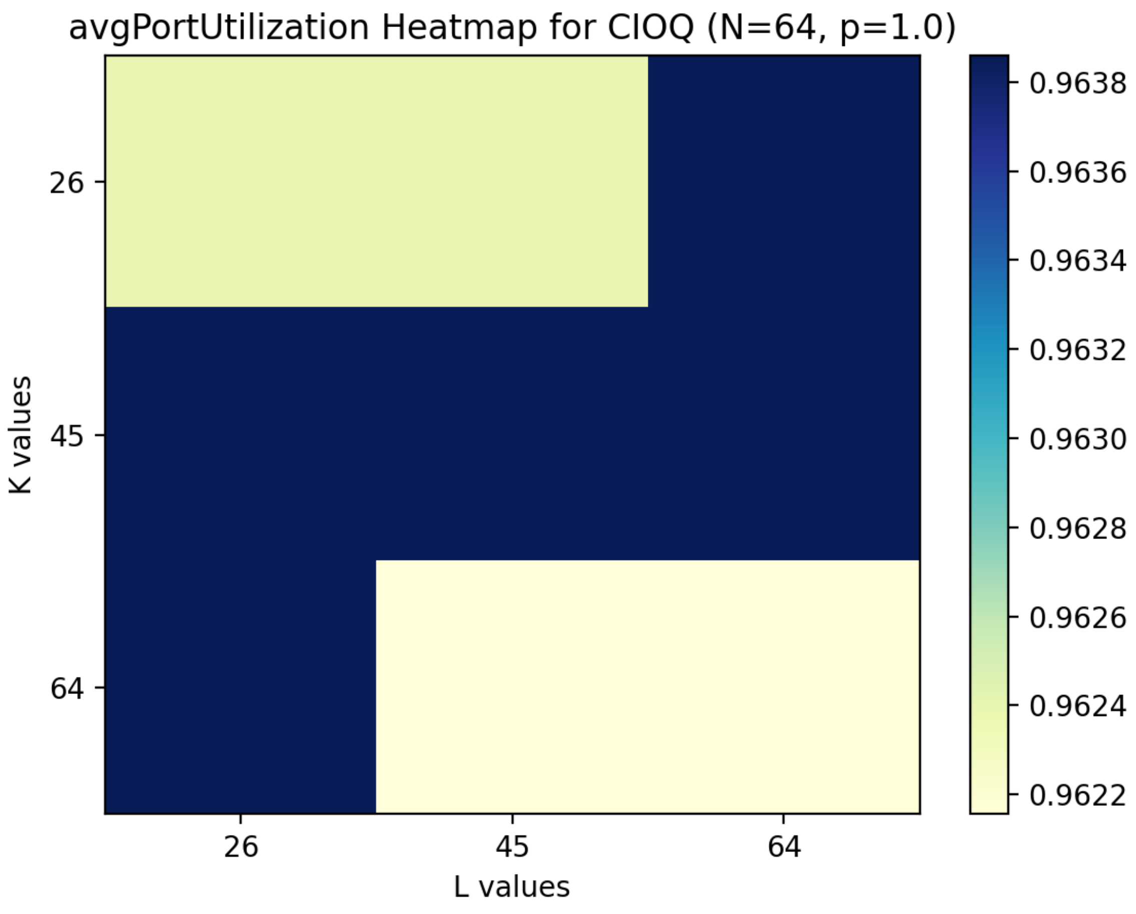


The packet delay is only significant at very high values of `N` and `p`. This is due to the algorithm being overloaded with a high number of packets. In all other cases, the $L \cdot N$ lookup resolves packets and the waiting time is very low. It is also observed that the delay does not hugely depend on `N` and this can be attributed to the iterated selection algorithm discussed below.

3.1 L-K analysis

Below are two heat maps showing the values for `avgPortUtilisation` for `N = 32, 64` and `p = 1.0`. This is done to show maximum performance analysis. As it can be observed, the values are around `0.96`. The variation with `L` and `K` is not very high (the scale of heat maps is different), since the effect is dominated by the high value of `p = 1.0`. This causes the buffers to be mostly full and the utilisation to be almost `1.0` since all the outputs almost always have a packet in them.


avgPortUtilization Heatmap for CIOQ (N=32, p=1.0)
K values
13
23
32
L values
13
23
32
0.964
0.963
0.962
0.961
0.960
0.959
0.958

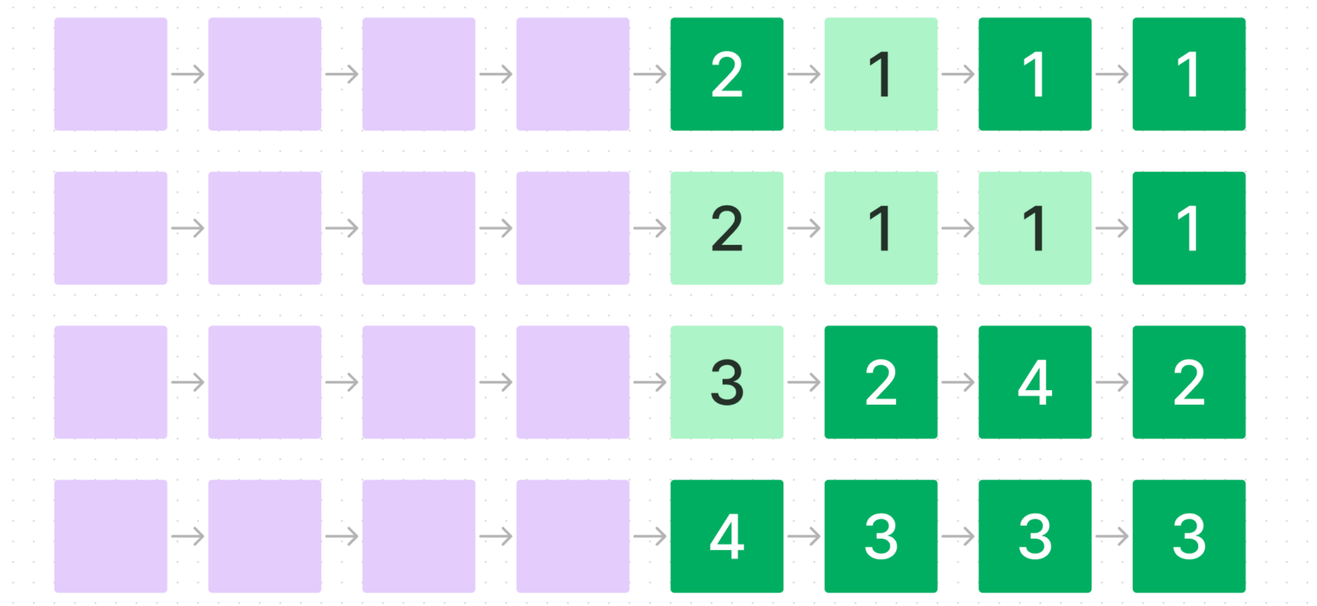

avgPortUtilization Heatmap for CIOQ (N=64, p=1.0)
K values
26
45
64
L values
26
45
64
0.9638
0.9636
0.9634
0.9632
0.9630
0.9628
0.9626
0.9624
0.9622

3.2 CIOQ Iterated Selection

The algorithm is:

1. Initialise `n, l = 0` and `knockoutTable[N] = { 0 } x N`
2. Iterate over `n < N` and `l < L`
3. Check the output port of a packet and select it if `knockoutTable[outPort] < K`

The example shown below is for `L = 4`, `K = 3`, `N = 4`. The arrows represent the queue/LL.



All the green (dark and light) shaded packets are looked up while selection and the dark green ones end up getting selected in the order of evaluation. The time complexity of this algorithm is `O(L * N)` since each packet within this range has to be checked. The space complexity is `O(N)` since the only variable that is dependent on the parameters is `knockoutTable[N].

The pseudo code for the algorithm is given below:

```
for n = 0 ... N:
    for l = 0 ... L:
        packet <- packets[n][l]
        outPort <- packet.outPort
        if knockoutTable[outPort] < K:
            sendToOutput(packet)
```

This is assuming that the packets are stored in a 2D array. The ideal implementation is a DLL where the packets are traversed with an iterator. For the exact implementation, refer to the `scheduleCIOQ` function in `main.c`.