1 - My work

In this simulation, I implement stable matching and apply this method to vlan assignment. Meanwhile, I also implement "Wait and Hop" using simpy. And I compare this two methods in terms of average completion time of all the flows, max link utilization, and the run time. This time, the link capacity, bandwidth demand of each flow and flow size are all uniformly distributed between a given range.

```
    # 每个流的带宽需求
    def bandwidth_demand(self):
        return random.uniform(10000,200000) # kbps
    # 每个flow的大小
    def flow_size(self):
        return random.uniform(1024*8,1024*20*8)  # 1~20KB (bits)
```

This time, all the links may have different capacity. So, from each flow's perspective, vlans will be *different* in terms of capcacity.

```
self.link_bandwidth = 10**7 # kbps
link_bandwidth[edge] = random.uniform(self.link_bandwidth/2,self.link_bandwidth)
```

2 - Implemention of stable matching

This are more than one way to apply stable matching to vlan assignment. It depends on what kind of perference list of each side we choose.

I choose a very simple method. For every vlan, it wants flows with smaller bandwidth demand so that the every link on this vlan may have more capacity left. For every flow, since we aim at minimizing max link utilization, it wants the vlan with less max link utilizaiton. For each routing class, I suppose half flows choose this vlan including this routing class and I can easily compute the max utilization. So each routing class will have a perference list based on this rough max utilization. In other words, the flow perfers the vlan with less flows on it.

Moreover, there are totally 8 routing classes. I use Fat-tree where $k = 4$ in this simulation and let each edge switch send a random flow to another edge swtich. Each flow will experience 2 or 4 hops before arriving at its destination. Considering the hop is either 2 or 4, I decide not to use the hop to compute the perference list.

But why just considering the flows between edges switches? This is a many-to-one stable matching problem, which means each vlan may have to match more than one flows. So the each vlan must have a limited “capacity” like each switch has a buffer size. If there is flow from aggregation to core switch, it must mactch with the vlan rooted at the core switch. In a word, some flow won't be able to get to its destination on some special vlan. That will *force* some vlan to match with some special flows. Since the edge switches are the leaves, flow between those will always find a path to the destination on each vlan.

The code is in the *Controller* class.

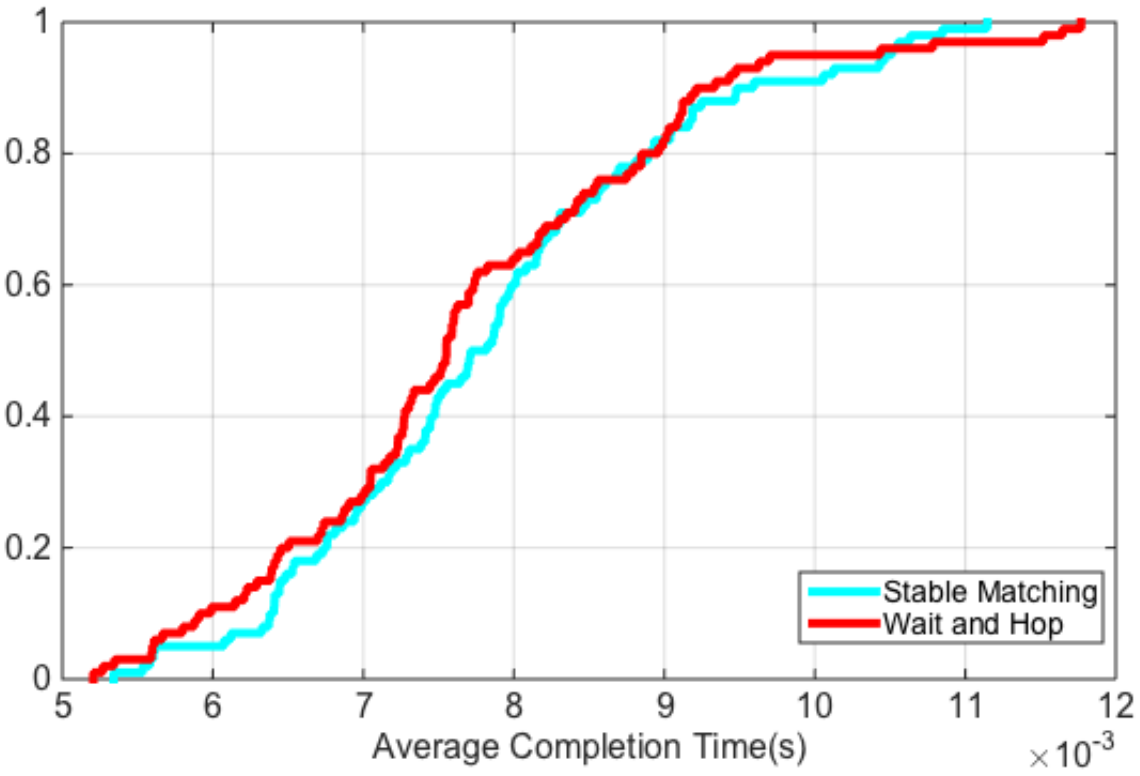
3 - Implemention of Wait and Hop

I finally implement "wait and hop' using simpy...Aftering implementing stable matching, I suddenly know how to implement it. There is slightly different from my old implement using C++. After getting all the stay-time of each state. I choose the one with the smallest max utilization from the three states with longest stay-time.

The β is equal to 1 in my code, so it will not be too slow.

4 - Simulation results

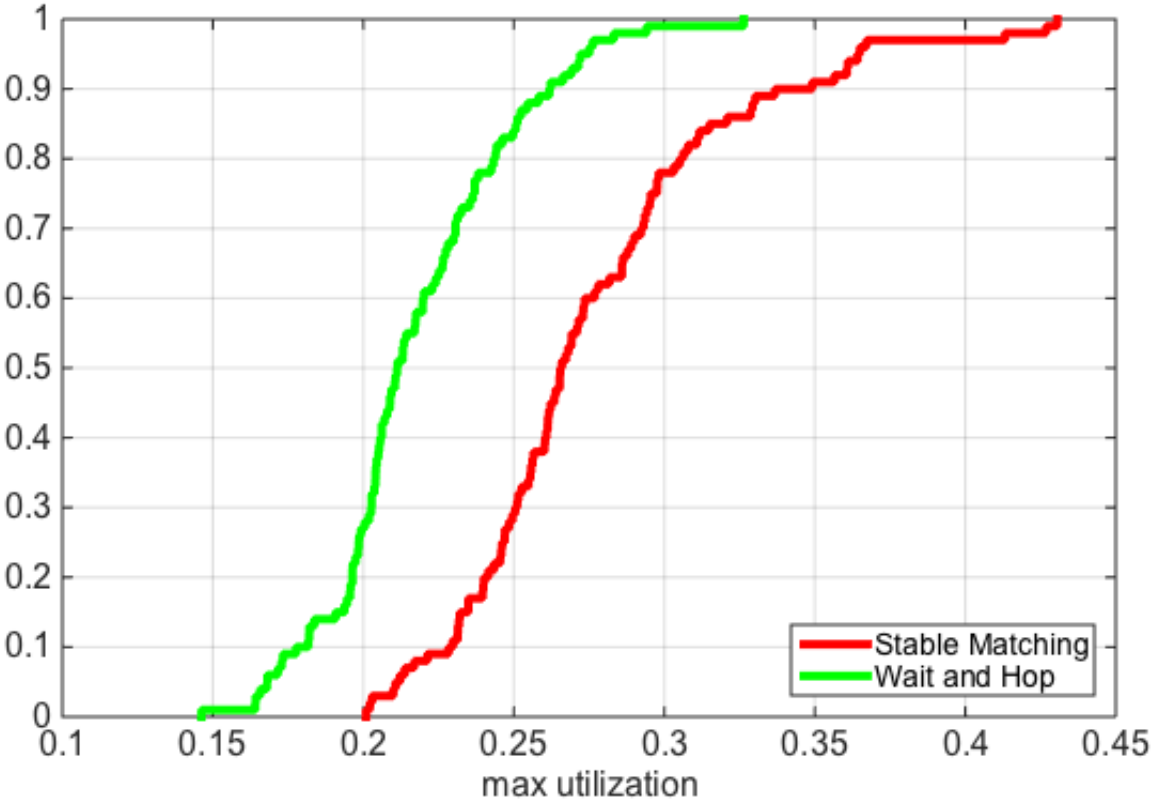
4.1 - Average completion time of each flow.



Considering that all the flows have 2 or 4 hops before getting to destination and the flow size is too small while each link has a wide bandwidth and each flow's bandwidth demand is also very big, it won't be strange that the two curves are roughtly the same.

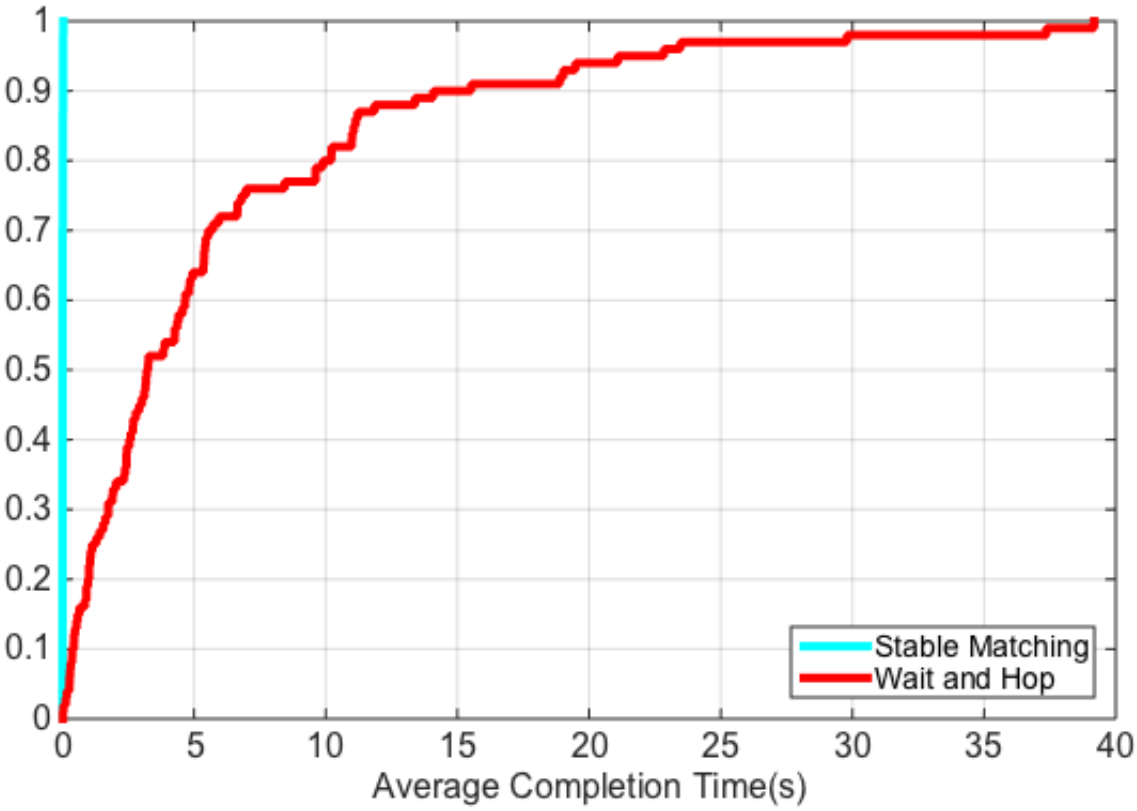
4.2 - Max utilization

Obviously, "Wait and Hop" ends up with smaller max utilization. But the gap is not more than 10%.



4.3 - Run time

“Wait and Hop” takse much much more time than "Stable Matching"...



5 - Short summary

Apparently, "Wait and Hop" ($\beta = 1$) takes much more time that "Stable Matching" while achieving better performance. However, "Stable Matching" is not so bad. The gap of max utilization of the two methods is

not greater than 10%. In my opinion, although "Wait and Hop" has better performance, it takes too much time compared with "Stable Matching". On such a large time scale, "Stable Matching" hardly takes time while achieving not bad performance.