Advanced Computer Networks 2022
Vrije Universiteit Amsterdam
Lin Wang, George Karlos, Florian Gerlinghoff

Lab2: Data Center Network Topology (Report)

Group number:	11
Group members:	Adithya S Vasisth, Rohaan D Almeida, Sudarsan Sivakumar
Slip days used:	0
Bonus claim:	Yes

1 Generating Fat-tree and Jellyfish Topologies

1.1 Fat-tree

Approach taken to generate the topology

- We first generate the the pods based on the number of ports. $Number\ of\ Pods = k$
- Each pods has a layer of aggregate switches(the ones connecting to the core switches) and a layer of edge switches(the ones connecting to the servers).
- We first create the edge switches. $Number\ of\ edge\ switches = k/2$
- We then create the servers. Each edge switch will be connected to $k/2$ servers.
- We will then create aggregate switches and connect them to the edge switches.
- Once we complete creating the pods, we will create core switches and connect each of these core switches to the aggregation switches. $Number\ of\ core\ switches = (k/2)^2$

Example of topology generated

We have used the networkx library to generate the topology based on the graph we have generated. We have used a simple naming convention. Each host starts with H followed by the pod number and the server number in the pod. For example, H_2_1 denotes the Host or Server in Pod 2 and the 2nd server in that particular pod. Edge Switches starts with ES and Aggregate Switches starts with AS. They are also followed by the Pod Number and the switch number in that particular pod. For example, AS_2_0 indicates the 0th Aggregate Switch in the 2nd Pod. The Core Switches start with the name CS and they are followed by the number of ports in switches of the topology and the number of the pod. For example, CS_4_1 indicates 4 ports in the topology and followed by the server number.

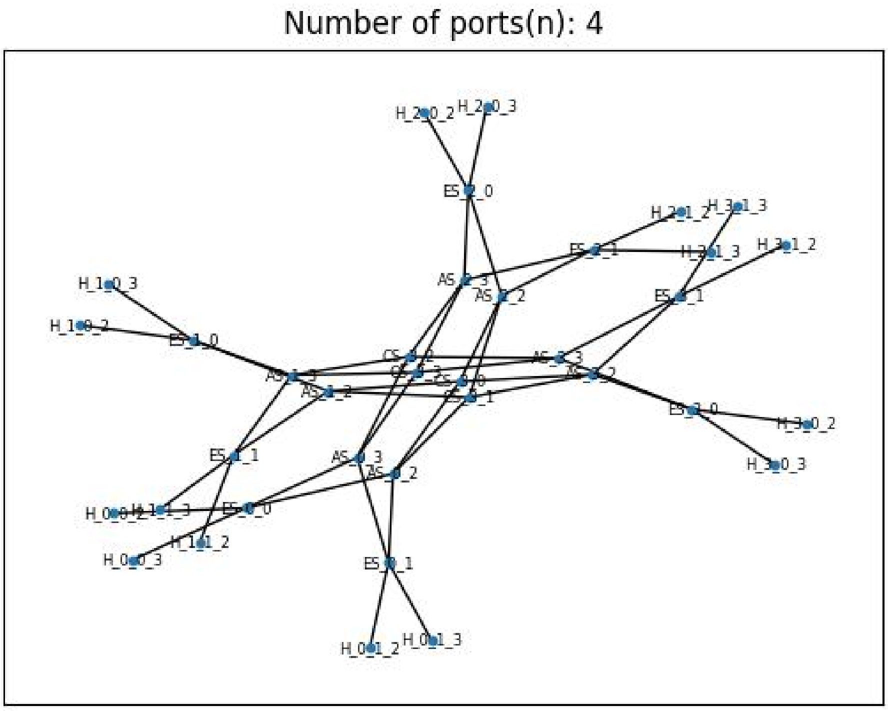

Figure 1: FatTree topology with a switch having 4 ports

Steps to run the code

- Go to visualise_topo.py. Set the number of ports as required.
- Run visualise_topo.py with *python3 visualise_ topo.py*
- The topologies of the Fat-tree should be generated in the folder named Topo.

1.2 Jellyfish

Approach taken to generate the topology

- The logic to start building the topology begins with calculating the total number of server needed per switch and then forming the connections between the servers and switches.
- After connecting the servers and switches we have to now establish a connection between the switches. To do so we begin by selecting 2 switches at random from the list of switches. We check if the switches are neighbours (We want to connect switches that are not neighbours) by using the function "isNeighbours()". We then add an edge between the 2 switches(Nodes).
- After we have exhausted all the ports in the switches or if we have some ports with no connection we try to normalise the graph. We do so by selecting a switch which has a open port and then selecting a random switch with all the ports connected, we break the connection(edge) between that switch and 2 random edges then reconnect these 2 edges with switch with the open port.
- If in the rare case of there being exactly one open port we break all connections of that switch and then follow the same logic as mentioned above.

- Another case that appears often is when we find all the ports are connected but end up with multiple disjoint graphs. To handle this case we run test if server 0 can connect to all other servers by using a custom broadcasting algorithm. If there is a disjoint graph due to server 0 unable to connect to several servers we rebuild the topology.

Example of topology generated

We have used the networkx library to generate the topology based on the graph we have generated. We indicate the servers with C followed by the server number. We indicate the switches with S followed by the switch number.

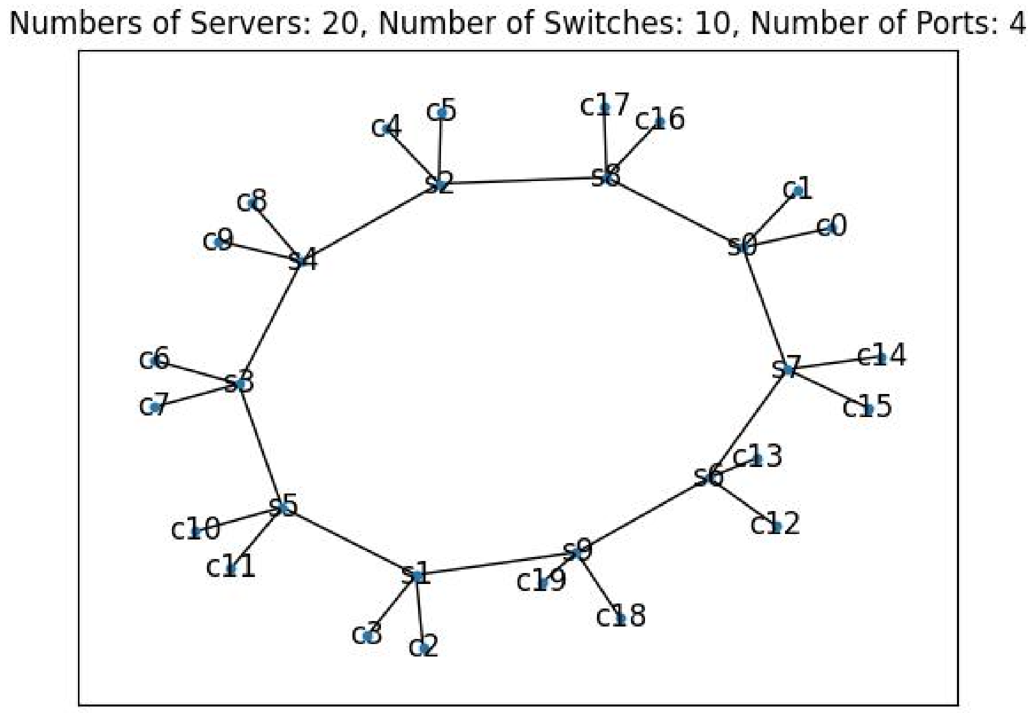

Figure 2: Jellyfish topology with 20 servers, 10 switches and 4 ports per switch

Steps to run the code

- Go to visualise_topo.py. Set the number of servers, switches and ports as required.
- Run visualise_topo.py with *python3 visualise_ topo.py*
- The topology of the Jellyfish should be generated in the same folder.

2 Comparing the Properties of Fat-tree and Jellyfish

In this section we are exploring following :

- Comparison of distribution of shortest distance between all possible server pairs in Fat-tree and Jellyfish topology(*Fig 1c in Jellyfish paper*)
- Best possible routing algorithm for jellyfish topology (*Fig 9 in Jellyfish paper*)

2.1 Distribution of shortest path lengths

Approach taken to generate the graph

We ran Dijkstra's algorithm for all possible server pairs over following setups:

1. Fat-tree with 14 ports per switch, which generates topology consist of 686 servers and 49 switches.
2. Jellyfish with 686 servers, 245 switches and 14 ports per switch.

Figure Generated

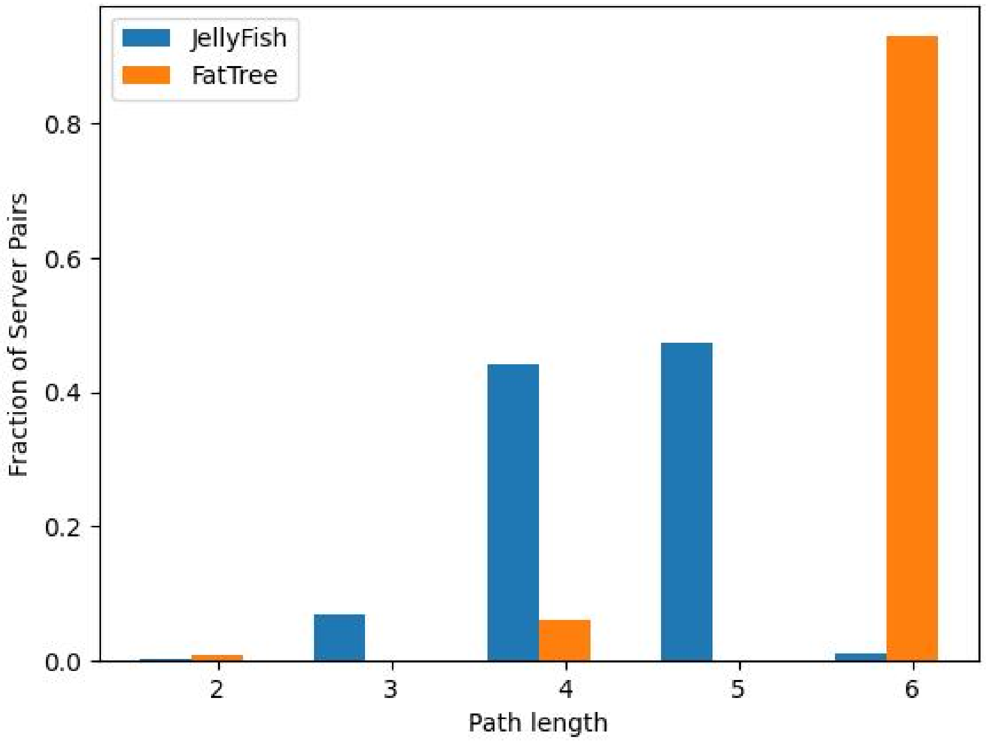

Figure 3: Shortest path length distribution in Jellyfish and Fat-tree topology

Figure 3 represents the normalized distribution of short path lengths over 10 runs. From the graph we can infer that jellyfish topology has better shorter path distribution. The reason for such distribution for fat-tree is because of its organized structure. The spike in 4 and 6 for fat-tree is for intra-pod and inter-pod paths respectively.

Steps to reproduce Figure 1c

- Change configuration such as *number of ports*, *number of servers*, *number of switches* in reproduce_1c.py
- Run *python3 reproduce_ 1c.py*

2.2 Comparison of different routing schemes for Jellyfish

Approach taken to generate the graph

To reproduce this graph, random server pairs are generated. Yen's algorithm came up with upto 64 shortest paths for each server pairs. The frequency of each links from the paths generated are maintained. We sorted links based on frequency which provides the rank. Then plot is made with rank vs frequency of the links.

Figure Generated

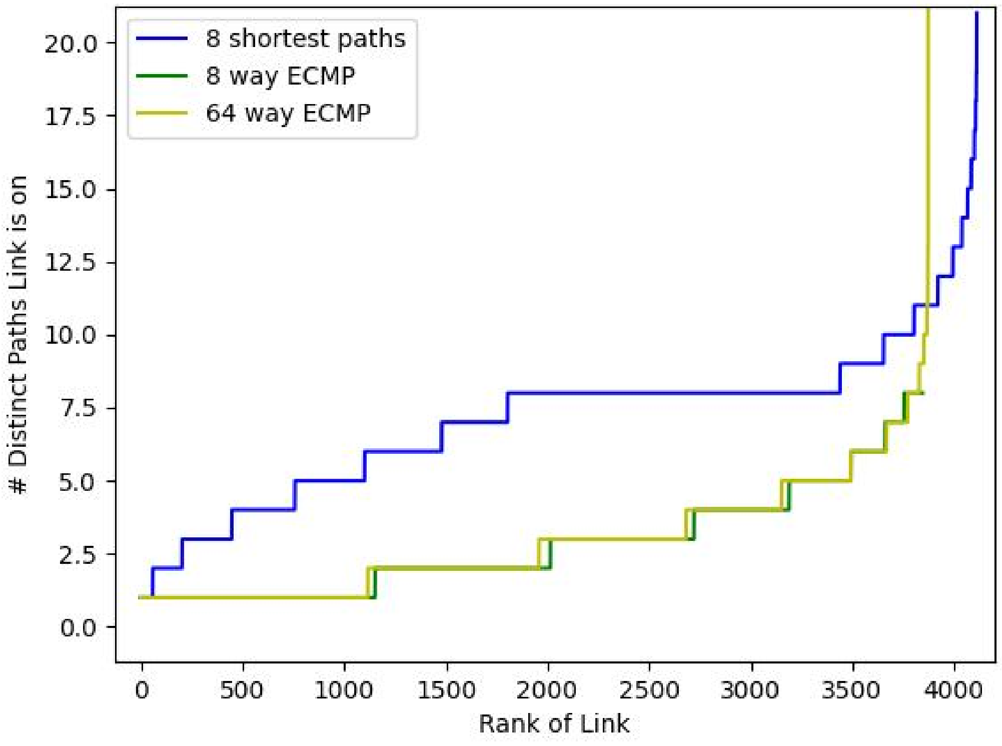

Figure 4: Path diversity of different routing schemes in Jellyfish topology

From Figure 4 we can infer that in jellyfish topology, 8 shortest path routing scheme has more path diversity. The difference (the ECMP routing schemes have explored only limited links) in reproduction of graphs is seen because of randomization in topology and generated server pairs.

Steps to reproduce Figure 9

- Change configuration in fig9_JellyFish.py
- Run python3 fig9_JellyFish.py, it will create a pickle file with following name JellyFish.pickle. It contains the dictionary of server pairs with k different paths. Since it takes very long time to compute. So to facilitate the testing we have pre computed pickles in K_shortest_paths_pickle folder.
- Once pickles are generated, run *python3 generate_fig9_graph_from_pickle.py*. Provide the pickle path in stdin.

3 Bonus

3.1 Approach taken to generate the topology

3.1.1 DCell

- The logic to start building the topology begins by creating n+1 dcells and n.(n+1) servers. We store the servers and switches in their own respective arrays. We number the id of the servers as 'dcellno.server index'.
- We then connect the the respective dcell switches to each server by running a for loop. We get the dcell switch index by server index in the server array divided by n. The dcell server index can be calculated by server index in the server array modulo n. Using the above 2 variables we establish connection between the servers.
- The logic to implement to the connections between the servers is we get the index of a dcell and index of the server in that dcell by running a for loop through the server array. We computer the next dell id by adding one to the current dcell index. We then run a for loop for the edges of the dcell switch and check if a server has a max of 2 edges. We select the first server with edges less than 2 and establish a connection between the 2 servers.
- The above process is repeated until all the servers have 2 edges.
- One interesting fact about Dcell is that servers can also act as routers.

Example of a Topology generated

In the figure 5, we have a topology of DCell created using the networkx library. We provide the graph we created to the library to generate the figure. The switches are denoted with the letter S followed by the switch number in the topology. The servers are indicated by the letter C followed by the cell number it belongs to and then by the server number in the particular cell.

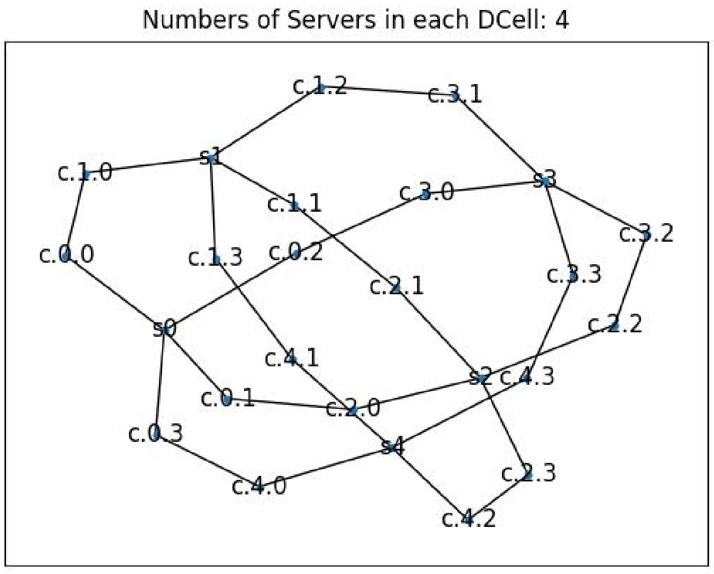

Figure 5: DCell topolgy with 4 servers in each cell.

3.1.2 BCube

- We start by accepting the values for k and n, where $k =$ *Number of levels* and $n =$ *Number of ports in each switch.*
- We start by adding servers to the topology. The *Total number of servers* $= n^{(k+1)}$
- There will be $k + 1$ levels of switches in the topology. Each level will have n^k number of switches.
- We then have to then begin to connect the servers and switches to each other.
- We iterate through each level of the switch. We find out the hop distance that the switch has to hop to connect to the list of servers.
- We then have a nested iteration to go through each switch in that level. Each switch will have n ports, and it is connected to the server based on the *hop* and the *current switch position.* We came up with an equation after analysing multiple configurations of the BCube. We preferred this approach over recursion due to lower complexity and time constraint.

Example of a Topology generated

In the figure 6, we have an example topology of a BCube generated using networkx library. We supply the graph we have created in the topology to plot this figure. We have hosts beginning with H followed by the BCube number of level k and the server number in the particular BCube. We indicate the switches with the letter S followed by the switch level number that it belongs to and then by the switch number in that particular level.

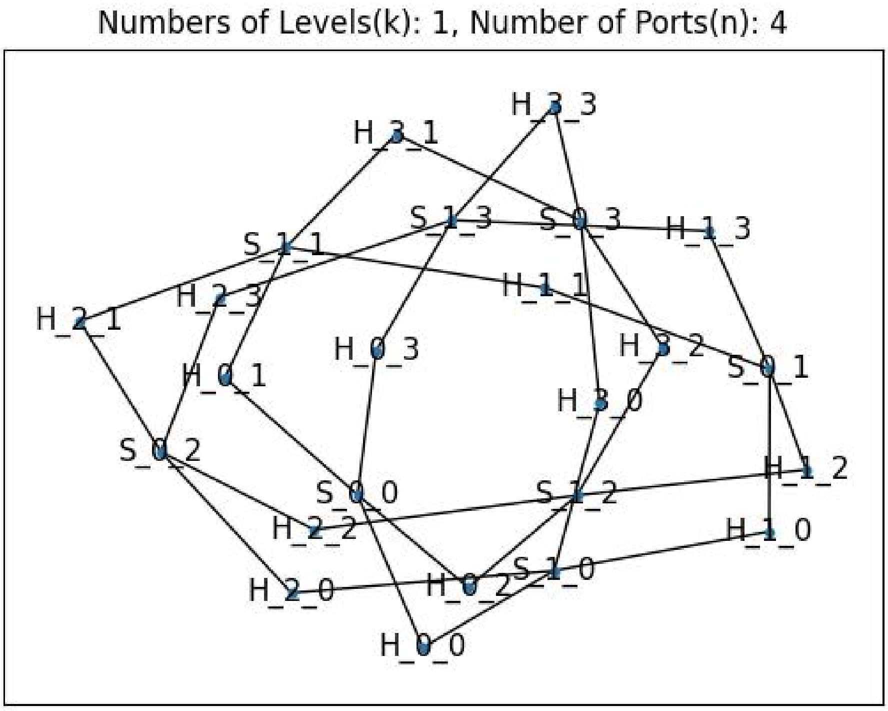

Figure 6: BCube topolgy with 1 level and 4 ports for each switch.

Steps to reproduce the topology

DCell

- Go to visualise_topo.py. Set the number of servers as required in each DCell.
- Run visualise_topo.py with *python3 visualise_ topo.py*
- The topologies of the DCell should be generated in the folder named Topo.

BCube

- Go to visualise_topo.py. Set the number of levels, and number of ports we need in each switch of the topology.
- Run visualise_topo.py with *python3 visualise_ topo.py*
- The topologies of the BCube should be generated in the folder named Topo.

3.2 Comparison of routing schemes for each topologies

Approach taken to generate the graph

To reproduce these graphs, random server pairs are generated. Yen's algorithm came up with upto 64 shortest paths for each server pairs. The frequency of each links from the paths generated are maintained. We sorted links based on frequency which provides the rank. Then plot is made with rank vs frequency of the links.

3.2.1 Fat tree

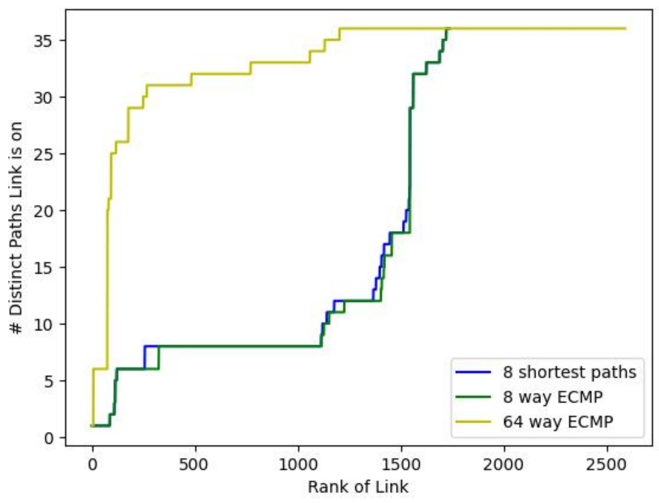

Figure 7: Path diversity of different routing schemes in FatTree topology

From Figure 7 we can infer that 64 way ECMP routing scheme has better path diversity in fat tree compared to other routing schemes. The layered approach provides this topology more equidistant shortest paths.

3.2.2 DCell

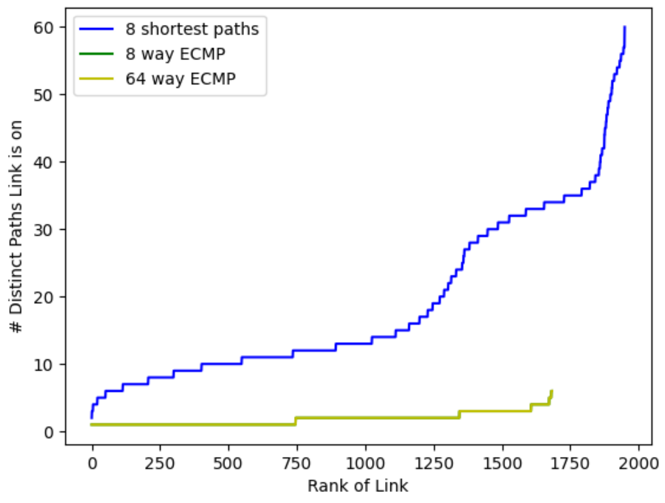

Figure 8: Path diversity of different routing schemes in DCell topology

From Figure 8 we can infer that 8 shortest path routing scheme has better path diversity. ECMP scheme has very poor performance, as equidistant path are less probable is such topology.

3.2.3 BCube

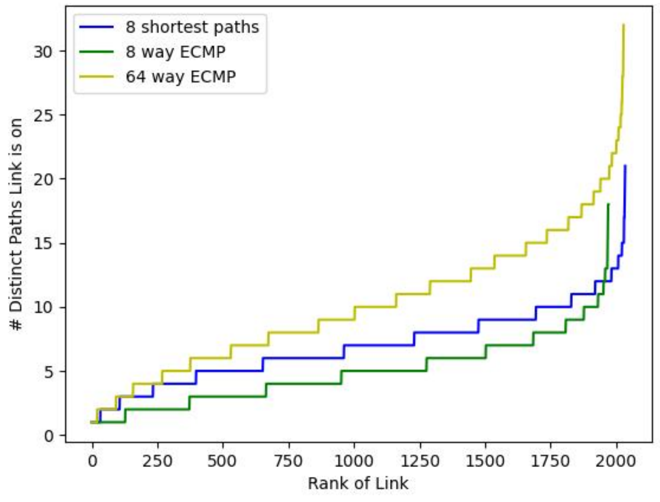

Figure 9: Path diversity of different routing schemes in BCube topology

From Figure 9 we can infer that 64 ECMP scheme has better path diversity in BCube topology.

Steps to reproduce Path diversity graph for all topologies

- Change configuration in fig9_*{topology}*.py
- Run python3 fig9_*{topology}*.py, it will create a pickle file with following *{topology}*name .pickle. It contains the dictionary of server pairs with k different paths. Since it takes very

long time to compute. We have also facilitated the testing with pre computed pickles in K_shortest_paths_pickle folder.

- Once pickles are generated, run *python3 generate_fig9_graph_from_pickle.py*. Provide the pickle path in stdin.