COL 819: Assignment -1

Shubham Gupta
csz198470@cse.iitd.ac.in

March 4, 2020

1 Introduction

In this assignment, we are comparing performances of two peer to peer systems Pastry Rowstron (2001) and Chord Stoica (2001). Both are a type of distributed hash tables which can efficiently figure out which node contain required key. The major advantage of these systems is to provide completely decentralized, scalable and self-organizing peer to peer overlay networks. Both protocols provides algorithms of addition of nodes, deletion of nodes, addition and searching for keys.

Pastry assigns numerical id to each node and key in same space.Each pastry node maintain set of numerically as well geographically close nodes which it uses to notify of arrival/departure of nodes. Each node also maintains a routing table. Routing table consists of $log_{2^b}N$ rows and $2^b - 1$ columns. N is no. of nodes in network, b is size(no. of bits) of each digit of ids space. Each row i consist of node ids which have common prefix of length i with current node. Each column represent one of $2^b - 1$ combinations of i+1 digit. Now to route key , a node first looks into its leaf set which contains of numerically nearest nodes and see if key id within the range. Otherwise it route this key to node from routing table which has 1 more prefix match than current node. This gives a $O(log_{2^b}N)$ routing time.

Chord also assigns numerical id to each node and key in same space. But these IDs (m bit) are arranged in circle modulo 2^m where m is a parameter. Chord routes key to a node which is next in position in ID circle wrt key in clock-wise direction. To do this, Node first checks if Key id is between its own node id and its successor's (next in circle) node id. If it is then its route this Key to its successor. If not, then node figures out preceding node wrt to Key ID using its finger table and route to that preceding node. This gives a routing time of $O(logN)$.

In this report, We show the experimental set up of both Pastry/Chord in a sequential setting. Furthermore, we show the distribution of no. of hops required to search the items after addition of nodes.

2 Experiment - Pastry

We have set-up 3 configurations of network which consist of 100, 500 and 1000 Nodes. Each configuration uses b= 4, $|L|$ = $|M|$ = 16. Each node contain leaf set, neighbour nodes

0	1	2
072b030ba126b2f4b2374f342	19f3cd308f1455b3fa09a282e	20f07591c6fcb220ffe637cda
c058f544c737782deacefa532	c16a5320fa475530d9583c34f	c203d8a151612acf12457e4d
cf004fdc76fa1a4f25f62e0eb	None	None
Node id - cfcd208495d565ef66e7dff9f		
Upper Leaf Set - cfecdb276f634854f3ef915e2 : cfee398643cbc3dc5eefc8933		
Lower Leaf Set - cfa0860e83a4c3a763a7e62d8 : cf004fdc76fa1a4f25f62e0eb		

Table 1: Routing table at Pastry node

and routing table. Each node is randomly assigned a 2d-geographical coordinate during creation. It is also assumed that when node is deleted, its data point gets deleted too. Now we will insert the 10000 data points and perform 1 million search queries. Further, distribution of number of hops required will be plotted.
After creation of network of 1000 nodes and adding 10000 data points and searching 1 million times, following is the pastry network summary-

Number of active nodes , 1000
Number of data add queries, 10000
Number of delete node queries, 0
Number of search queries, 1000000
Current data stored, 10000

For node with id-"cfcd208495d565ef66e7dff9f98764da", its routing table can be seen in 1.
It can be seen that first row has no matching prefix with node id, 2nd row has nodes matching prefix of length 1 and 3th row has nodes with matching prefix of length 2.
After deletion of half the nodes, pastry network summary is-

Number of active nodes , 1000
Number of data add queries, 10000
Number of delete node queries, 500
Number of search queries, 1000000
Current data stored, 5125 (This no. might vary depending upon the location assigned to each pastry node but approximately 50% data points should get deleted).

Following figures(2-7) shows hops distribution for each configuration. Figure 8 is plot of mean hops vs network size.
There are two observation, first is mean number of hops increases slightly for 100 size network to 1000 size network as

$$no.of hops \propto log_{2^b} N$$

where b = 4. For N=1000 , this comes out approx 3.
Also, it can be seen that after random dropping of nodes, mean number of hops increases slightly in later half of plot, as network keep will try to reach to node as close numerically possible to key but that numerical close node will not have that key. So, it will further try to see in neighbourhood set or leaf set thus increasing the hops.

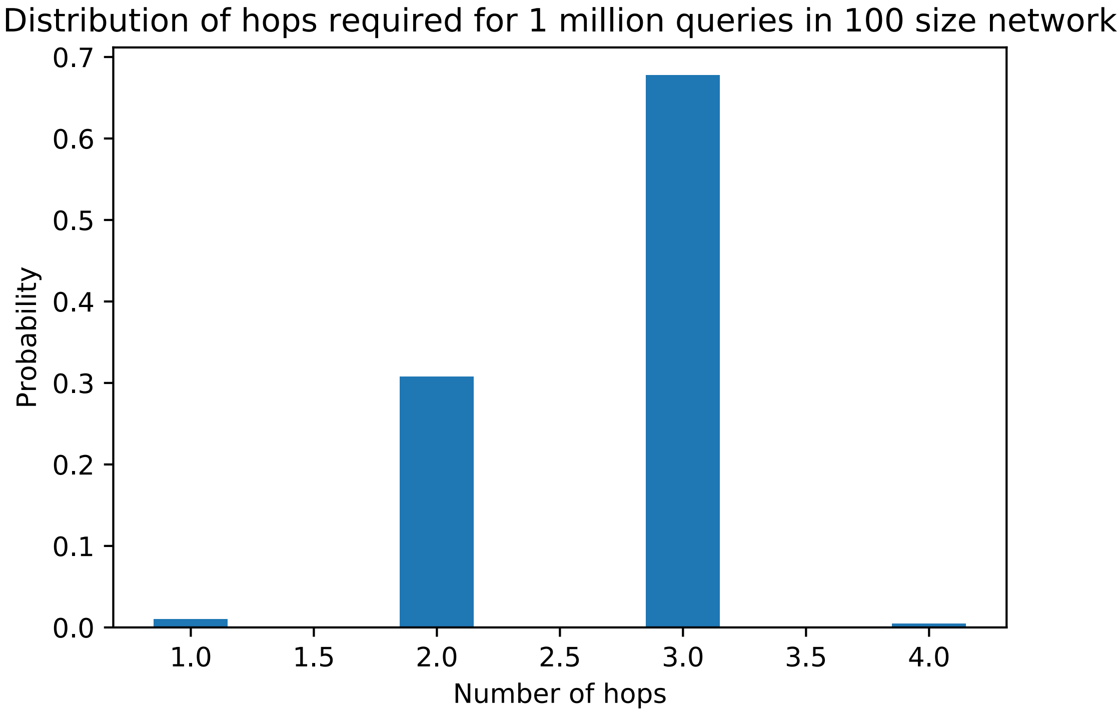

Figure 1: Hops distribution on 100 node network

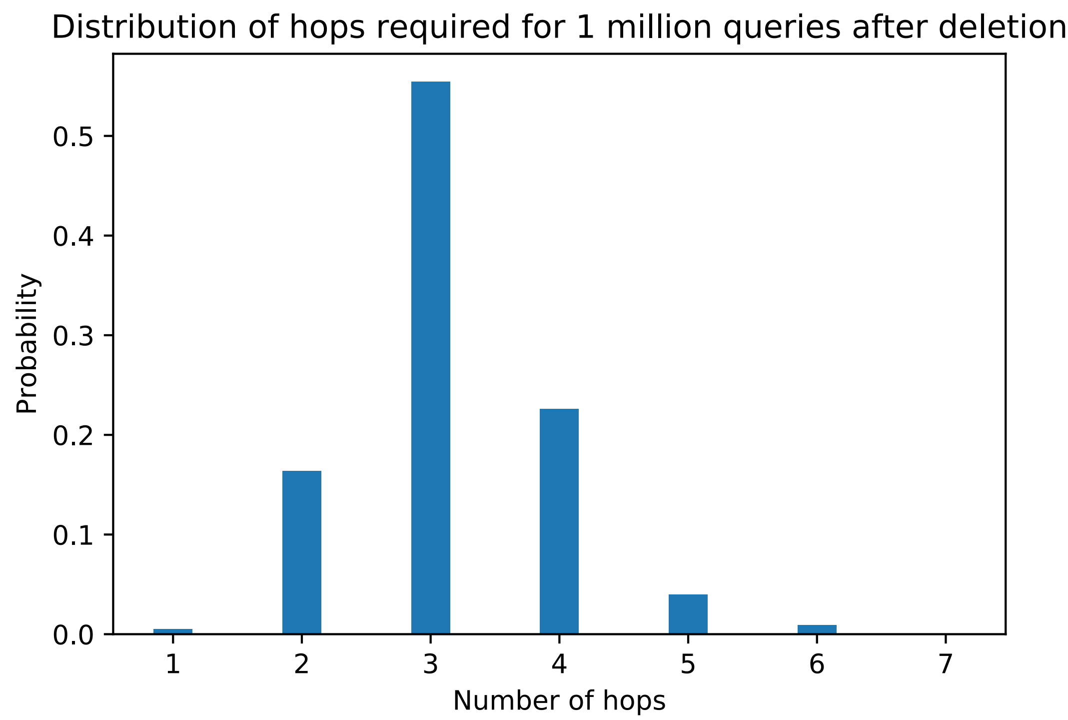

Figure 2: Hops distribution after randomly dropping 50% nodes of 100 node network

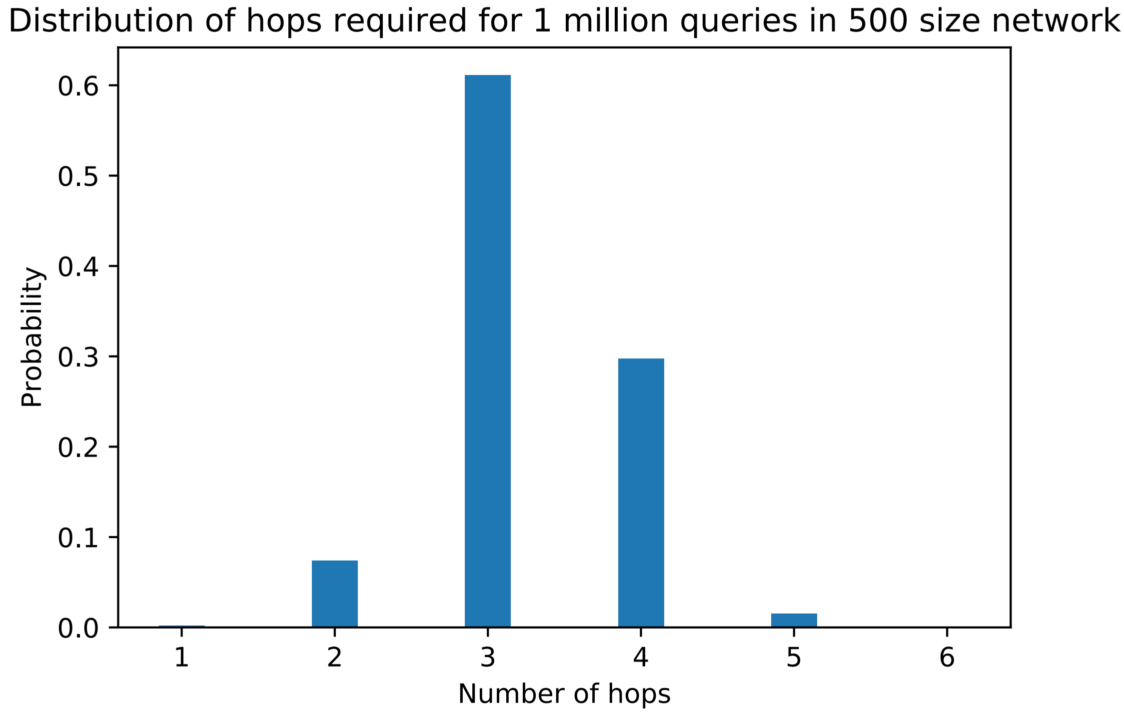

Figure 3: Hops distribution on 500 node network

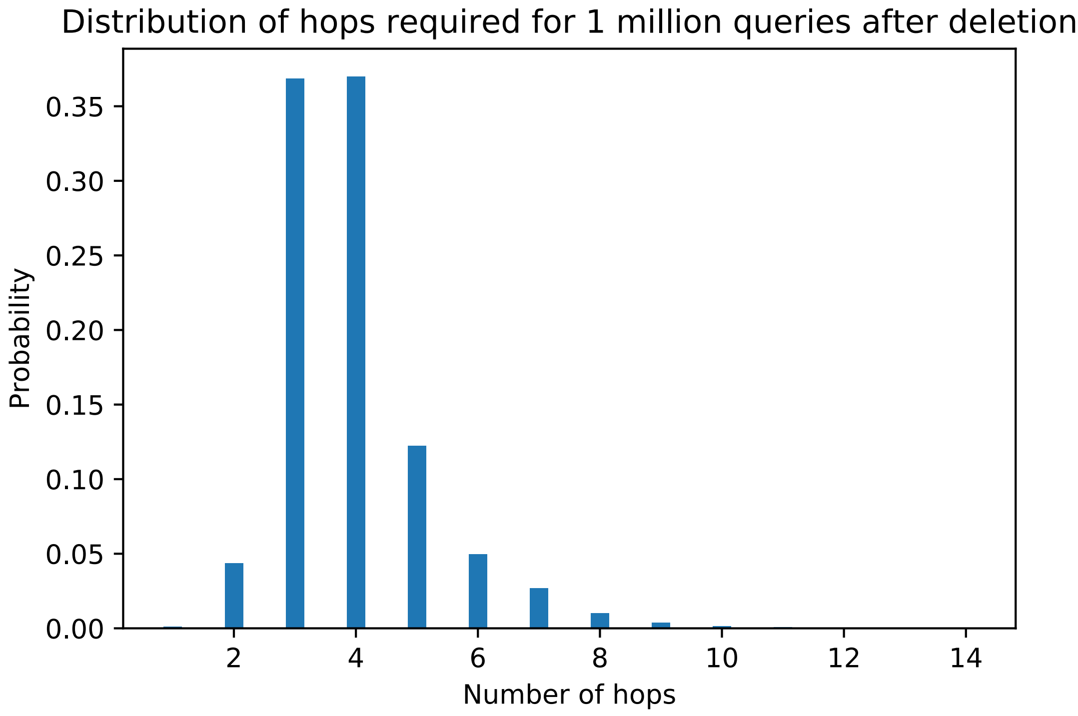

Figure 4: Hops distribution after randomly dropping 50% nodes of 500 node network

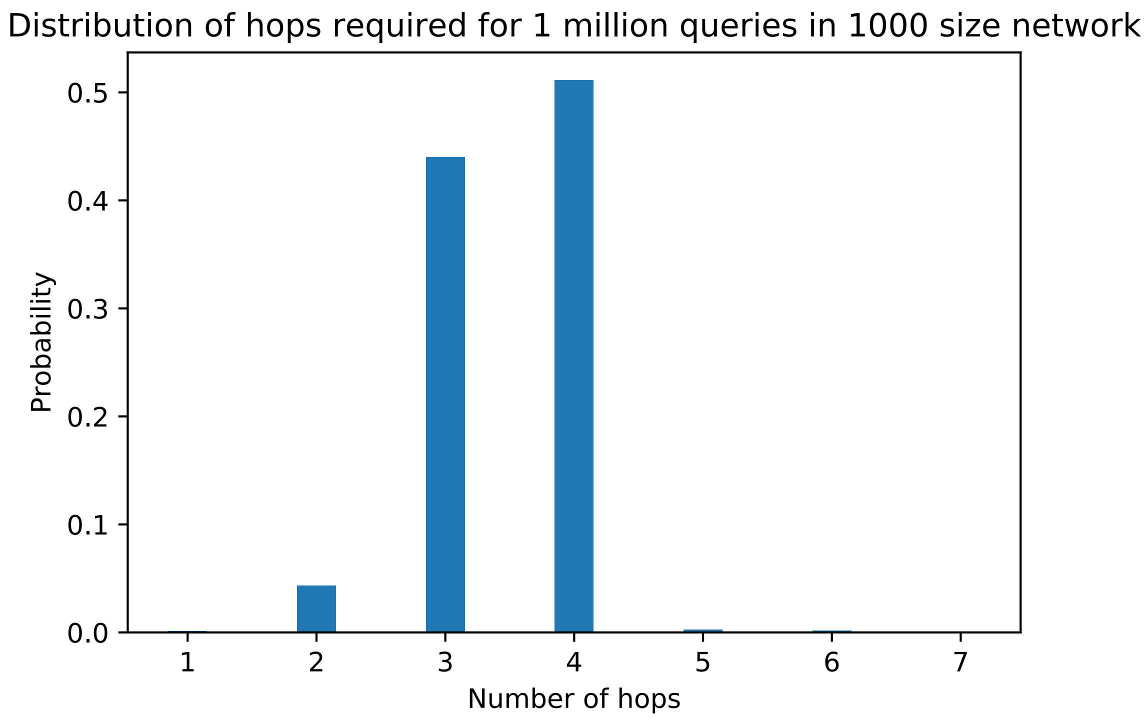

Figure 5: Hops distribution on 1000 node network

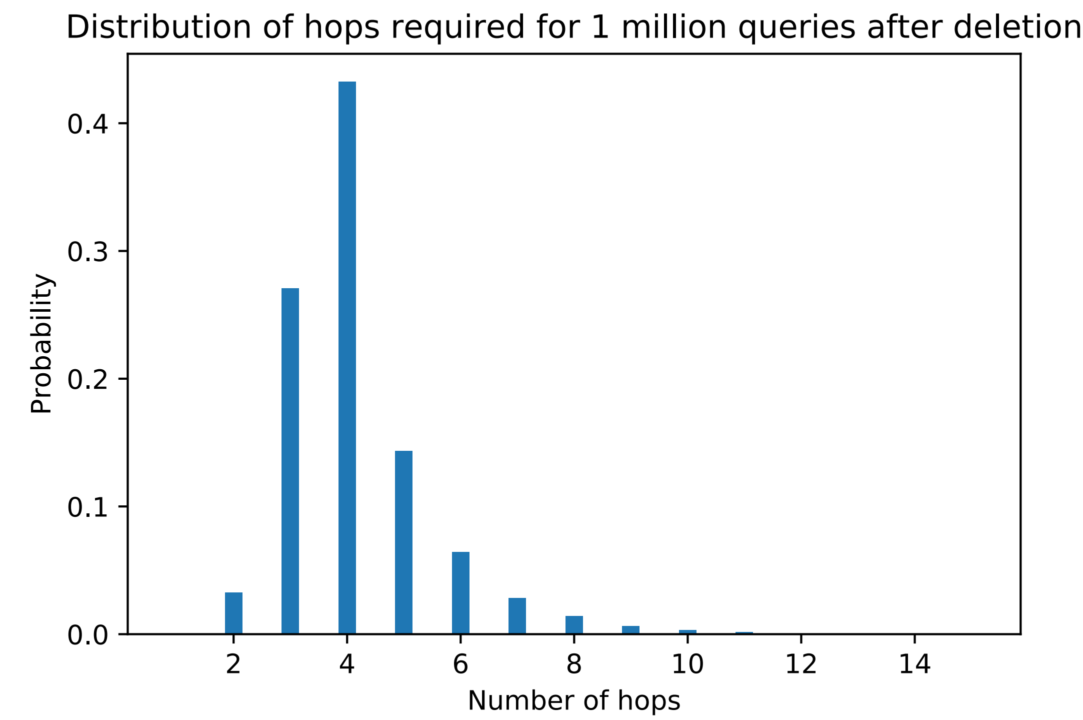

Figure 6: Hops distribution after randomly dropping 50% nodes of 1000 node network

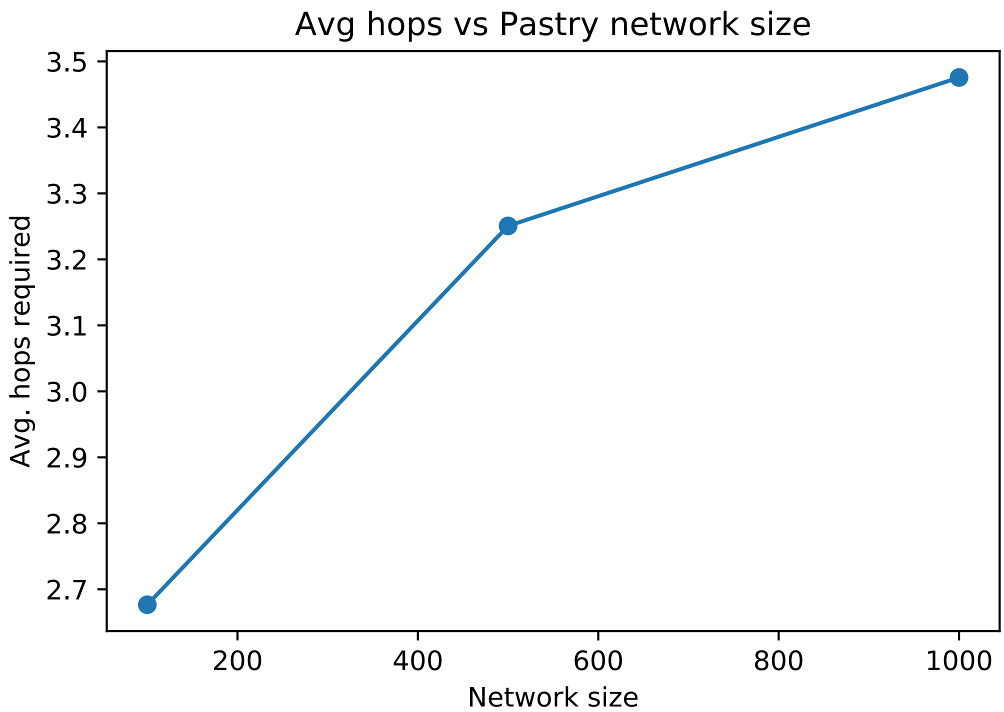

Figure 7: Means number of hops for each configuration

3 Experiment - Chord

Similar to Pastry, We have set-up 3 configurations of network which consist of 100, 500 and 1000 Nodes. It is also assumed that when node is deleted, its data point gets deleted too. For each configuration, we will insert the 10000 data points and perform 1 million search queries. Further, distribution of number of hops required will be plotted. Each node maintains a finger table to efficiently route the queries in network. Table 2 shows the format of finger table at a random node with id 243877.
Following figures (10-15) shows hops distribution for each configuration. Figure 16 is plot of mean hops vs network size.
Similar to Pastry, two observations can be made. First, mean number of hops increases slightly for 100 size network to 1000 size network as

$$no.of hops \propto log N$$

For N=100 , this comes out approx 6 and mean number of hops is approximately equal to 6.
Also, it can be seen that after random dropping of nodes, hops distribution shifts very slightly in as compared to pastry. In this case network keep will try to reach to the successor of key. But, if this successor doesn have the key, then it will know that key doesn't exist in the network and will not process further. Figure 17 shows the network route taken to find

Start	interval start	interval end	successor node
243878	243878	243879	245001
243879	243879	243881	245001
243881	243881	243885	245001
243885	243885	243893	245001
243893	243893	243909	245001
243909	243909	243941	245001
243941	243941	244005	245001
244005	244005	244133	245001
244133	244133	244389	245001
244389	244389	244901	245001
244901	244901	245925	245001
245925	245925	247973	247058
247973	247973	252069	248237
252069	252069	260261	252412
260261	260261	276645	261174
276645	276645	309413	278058
309413	309413	374949	311010
374949	374949	506021	375505
506021	506021	768165	508393
768165	768165	243877	769352
Predecessor- 243070			
Successor- 245001			

Table 2: Finger table at Chord node

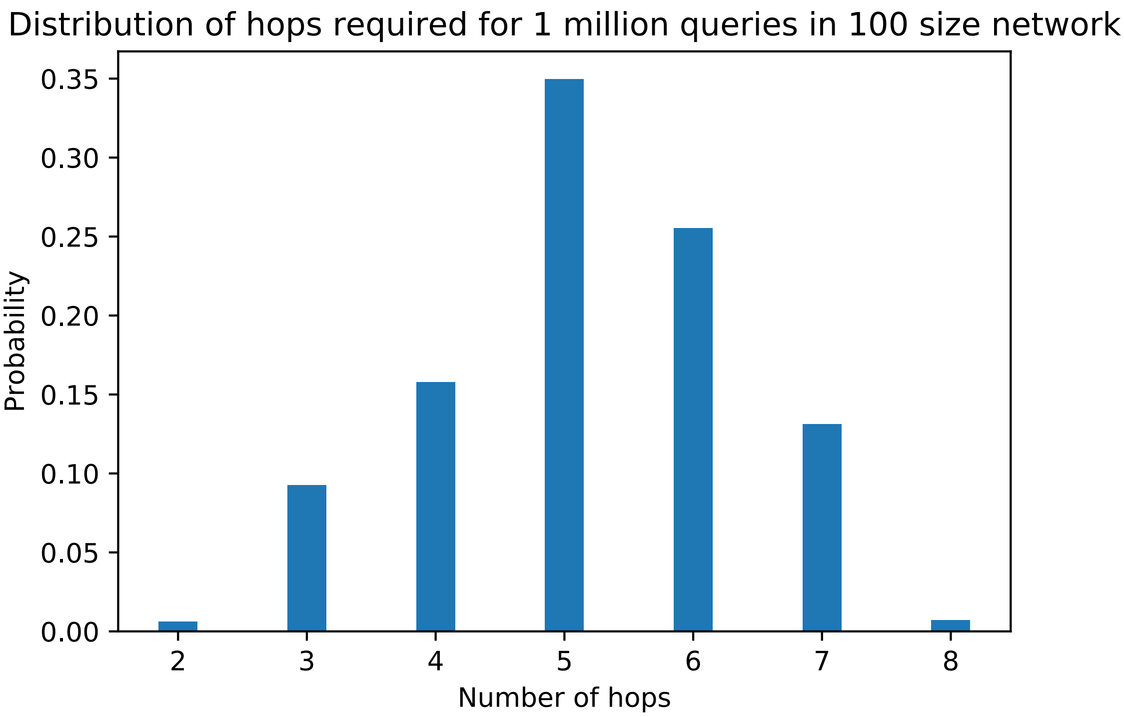

Figure 8: Hops distribution on 100 node network

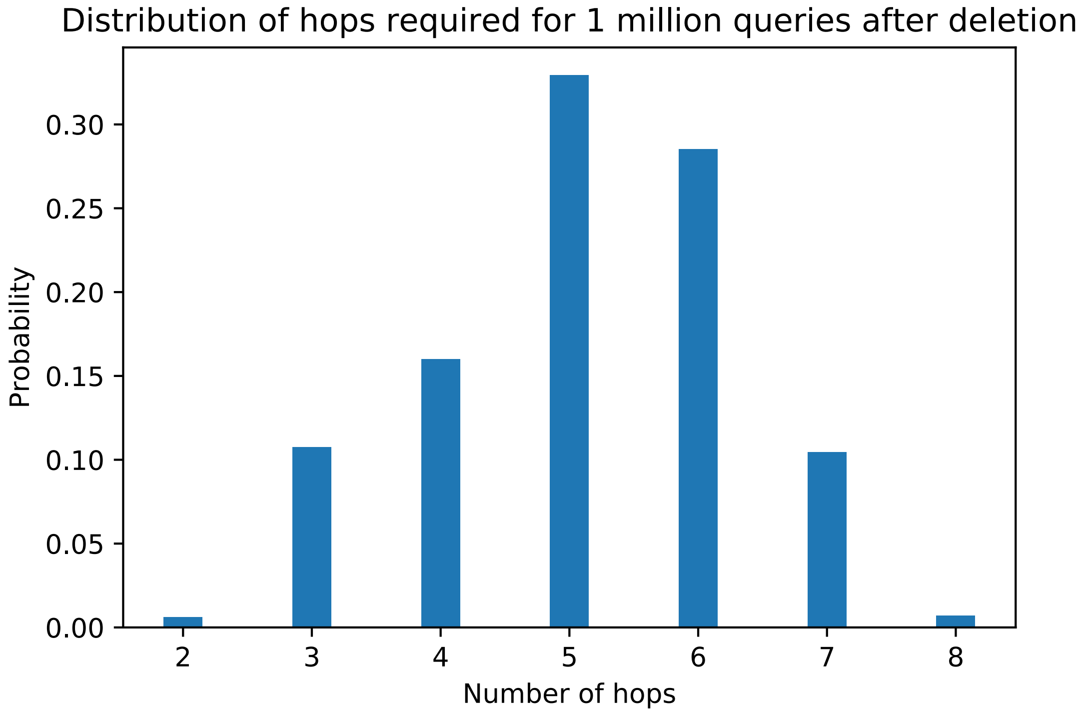

Figure 9: Hops distribution after randomly dropping 50% nodes of 100 node network

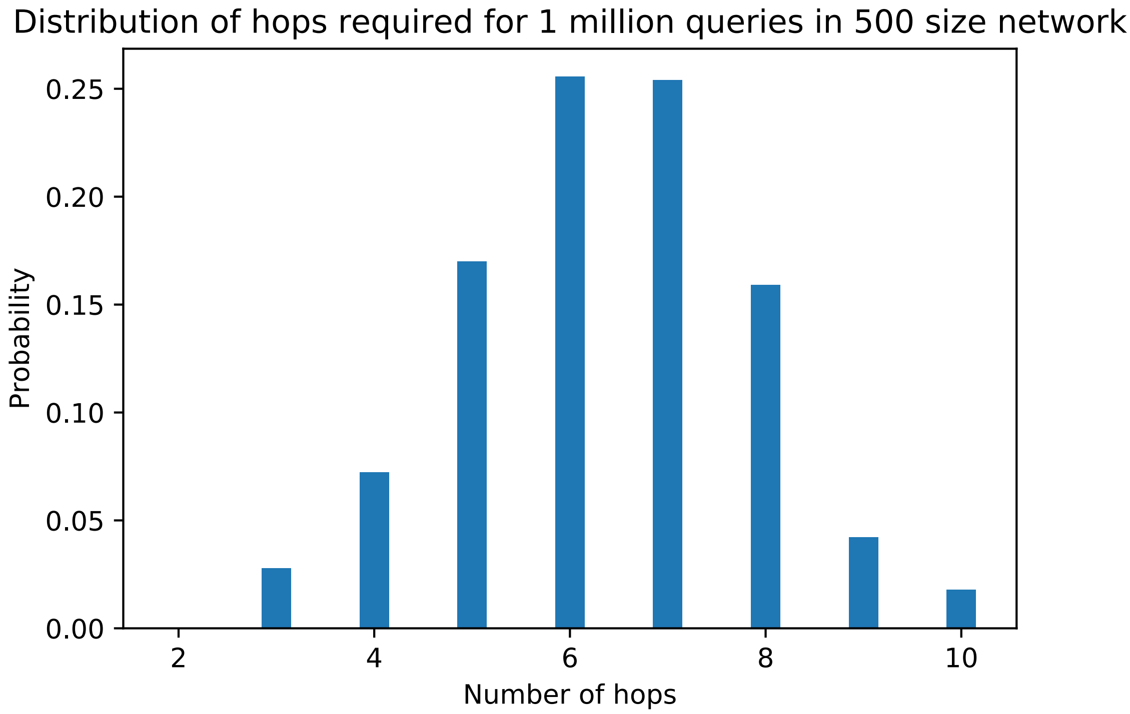

Figure 10: Hops distribution on 500 node network

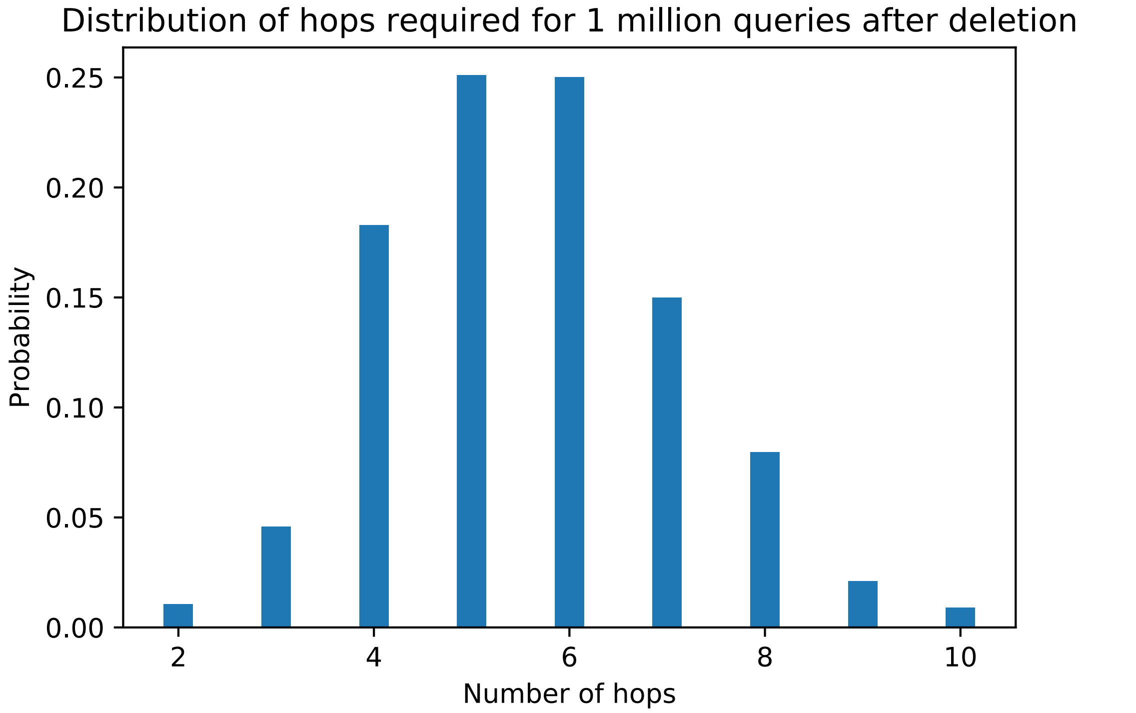

Figure 11: Hops distribution after randomly dropping 50% nodes of 500 node network

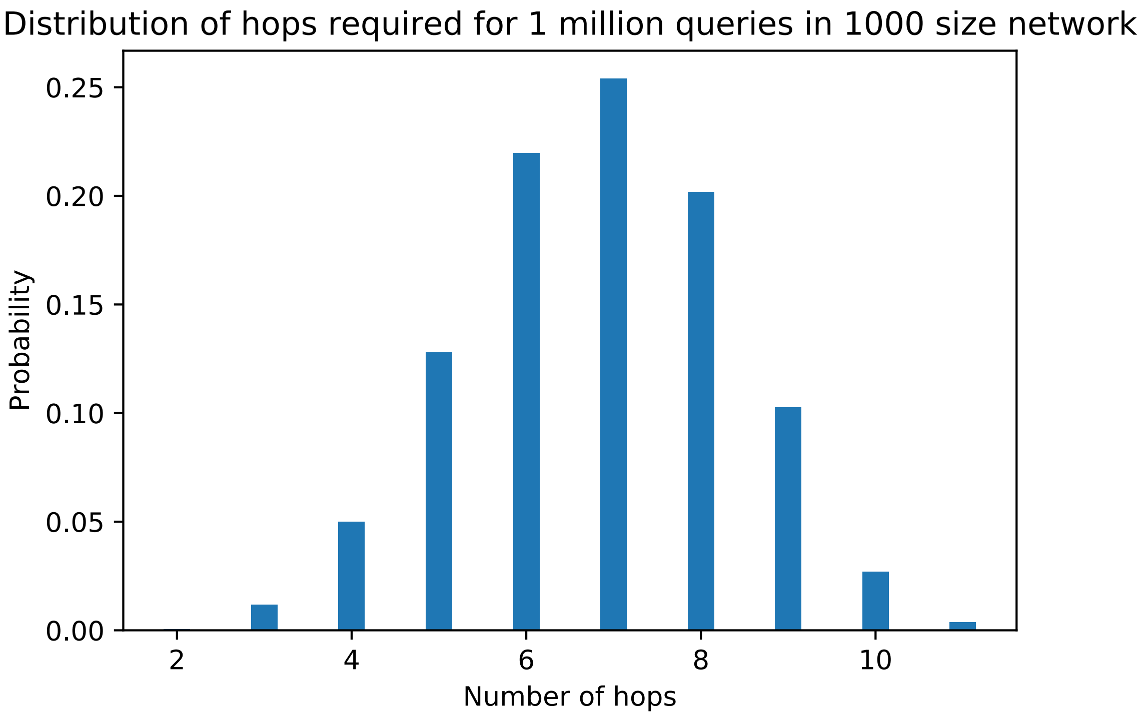

Figure 12: Hops distribution on 1000 node network

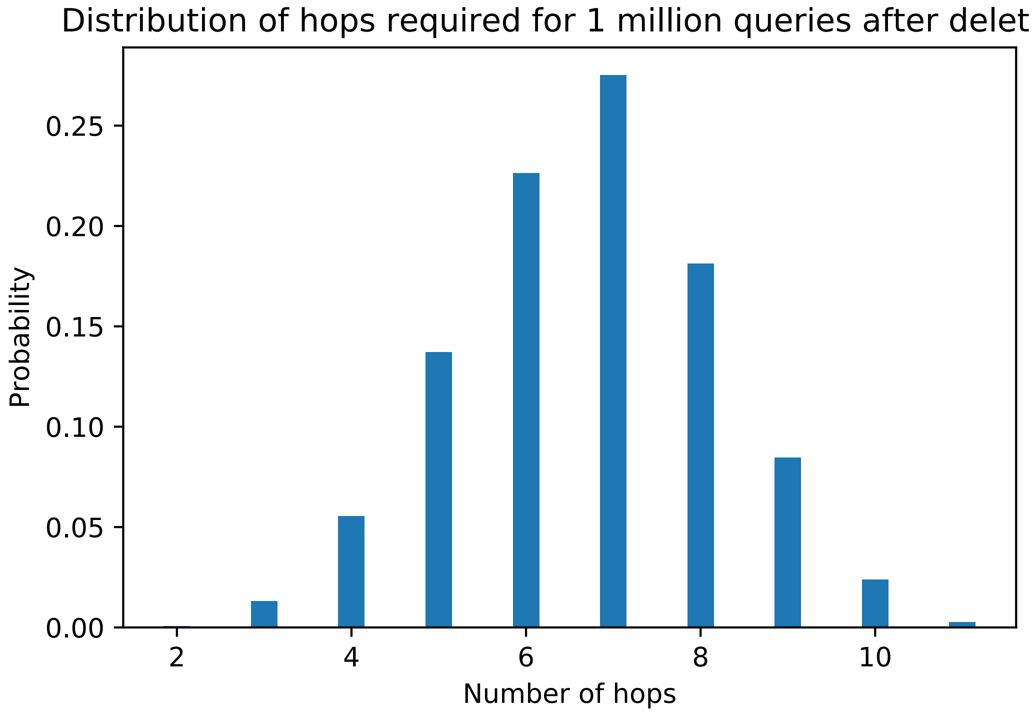

Figure 13: Hops distribution after randomly dropping 50% nodes of 1000 node network

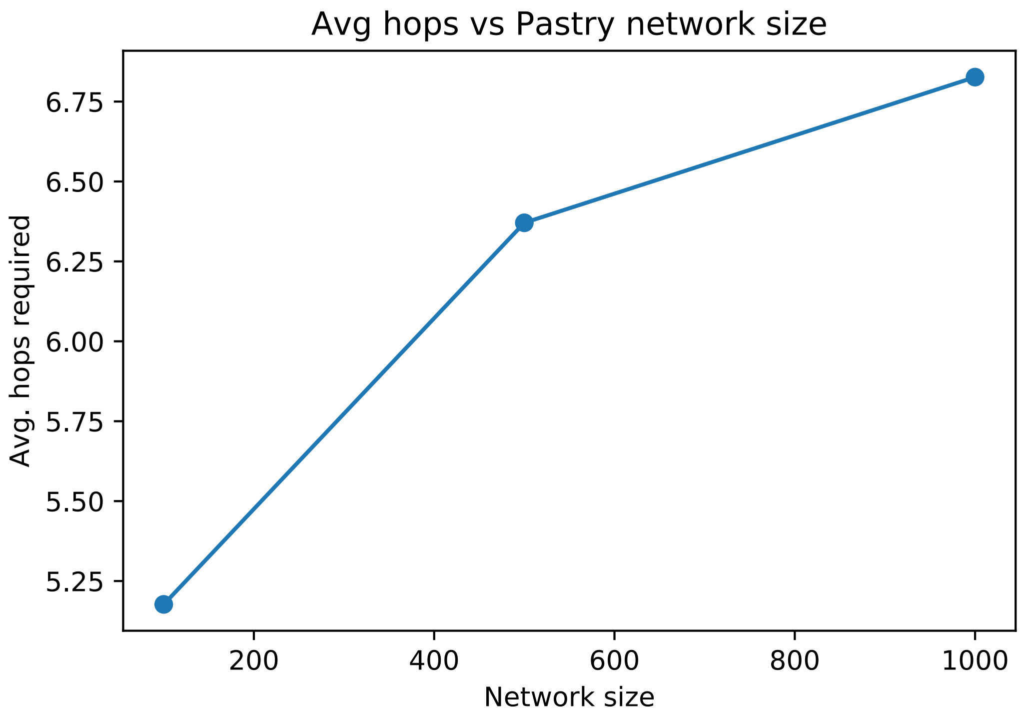

Figure 14: Means number of hops for each configuration

the key. It can be seen that at each next step, node is numerically closer to key.

```
Lookup route of key, 443636  : 339802 -> 406612 -> 439539 -> 443008 -> 443620 -> 445408

Lookup route of key, 793980  : 339802 -> 604542 -> 737729 -> 773200 -> 789882 -> 793125 -> 794210

Lookup route of key, 581388  : 339802 -> 473113 -> 541369 -> 574480 -> 579174 -> 581268 -> 581785

Lookup route of key, 677453  : 339802 -> 604542 -> 670131 -> 675207 -> 676548 -> 677094 -> 678426

Lookup route of key, 312023  : 339802 -> 864870 -> 80052 -> 211957 -> 279246 -> 297321 -> 306900 -> 311378
-> 312498

Lookup route of key, 492485  : 339802 -> 473113 -> 490930 -> 492425 -> 492964

Lookup route of key, 1001091  : 339802 -> 864870 -> 995976 -> 1000543 -> 1008914

Lookup route of key, 915454  : 339802 -> 864870 -> 897943 -> 906407 -> 910892 -> 913026 -> 913952 -> 918690

Lookup route of key, 189274  : 339802 -> 864870 -> 80052 -> 148233 -> 181162 -> 185383 -> 187565 -> 188668
-> 189191 -> 190928

Lookup route of key, 632042  : 339802 -> 604542 -> 623331 -> 632006 -> 632873
```

Figure 15: Path taken by network to route the 10 random queries

4 Performance Comparison

Chord is bit slower compared to Pastry as run time complexity of query search in Chord is $O(logN)$ while Pastry has time complexity of $O(log_{2^b}N)$. But Chord is easy to maintain and implement compared to Pastry.

5 How to run code

For Pastry,

```
python Pastry.py
```

Similarly for Chord-

```
python Chord.py
```

Please use python3, and code will run for all 3 configurations. It will generate required plots too.

```
Shubhams-MacBook-Air:Assignment-1 shubham$ /Users/shubham/anaconda3/bin/python Pastry.py
Creating network for  100
adding data points
Searching,
Number of active nodes ,  100
Number of data add queries,  10000
Number of delete node queries 0
Number of search queries,  1000000
Current data stored  10000
deleting 50% nodes
searching for queries
Creating network for  500
adding data points
Searching,
Number of active nodes ,  500
Number of data add queries,  10000
Number of delete node queries 0
Number of search queries,  1000000
Current data stored  10000
```

Figure 16: Screenshot of Pastry code execution

```
Shubhams-MacBook-Air:Assignment-1 shubham$ /Users/shubham/anaconda3/bin/python Chord.py
Creating network for  100
Adding nodes
Adding keys
Searching,
deleting 50% nodes
searching for queries
Creating network for  500
Adding nodes
Adding keys
Searching,
deleting 50% nodes
searching for queries
Creating network for  1000
Adding nodes
Adding keys
Searching,
deleting 50% nodes
searching for queries
Lookup route of key, 797292  : 909833 -> 388062 -> 650305 -> 782369 -> 791688 -> 797104 -> 7992
17
```

Figure 17: Screenshot of Chord code execution

References

Rowstron. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001 lecture notes in computer science* (p. 329–350).

Stoica. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* (p. 149–160). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/383059.383071` doi: 10.1145/383059.383071

6 Pastry Python code

```
import hashlib
import random
import math
import numpy as np
from matplotlib import pyplot as plt
import math
import pandas as pd
from collections import Counter
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)
inf = math.inf
def hex_id(id):
   return hashlib.md5(str(id).encode()).hexdigest()
def random_num(a,b):
   return a + random.random()*(b-a)
def euclidean_distance(a,b):
   return np.linalg.norm(np.array(a)-np.array(b))
def node_abs_id_distance(a,b):
   return abs(int(a,16) - int(b,16))
def node_id_distance(a,b):
   return int(a,16) - int(b,16)
def comman_prefix_length(a,b):
   ct = 0
   for i in range(0,len(a)):
      if a[i] == b[i]:
         ct += 1
      else:
         return ct
   return ct

def compare(a,b,mode="eq"):
   if mode == "eq":
      return int(a,16) == int(b,16)
   if mode == "g":
      return int(a,16) > int(b,16)
   if mode == "l":
      return int(a,16) < int(b,16)
   if mode == "ge":
      return int(a,16) >= int(b,16)
```

```
    if mode == "le":
        return int(a,16) <= int(b,16)
def min_node_id(leafSet):
        return min(leafSet,key = lambda val: int(val,16))

def max_node_id(leafSet):
        return max(leafSet,key = lambda val: int(val,16))

def get_prob_distribution(lst):

    a = Counter(lst)
    ct = sum(a.values())

    vals = list(a.keys())
    probs = [item*1.0/ct for item in a.values()]
    idx = np.argsort(vals)
    vals = np.array(vals)[idx]
    probs = np.array(probs)[idx]
    return vals,probs

class Key():
    def __init__(self,key):
        self.name = key
        self.id = hex_id(self.name)



class Node():
    def __init__(self,node_name,b=4):
        self.name = node_name
        self.id = hex_id(self.name)
        self.data = {}
        self.b = b
        self.L = self.M = int(math.pow(2,b))
        self.location = (random_num(0,90),random_num(0,180))##
            ↪ latitude and longitude
        self.routingTable = [[ None for item in range(0,int(math.pow
            ↪ (2,self.b)))] for i in range(0,int(128/self.b))] ## b
            ↪ = 4 configuration
        self.leafUSet = [] ### Upper leaf set
        self.leafLSet = [] ### Lower leaf set
        self.nbrSet = []
```

```
        self.isOnRoute = False

    def findNearestLeafNode(self,key): ### Accepts id
        nearest_node = None
        nearest_node_dist = inf
        for nbr in self.leafUSet + self.leafLSet:
            distance = node_abs_id_distance(nbr,key)
            if nearest_node_dist > distance:
                nearest_node_dist = distance
                nearest_node = nbr
        return nearest_node
    def check_if_key_lies_in_leaf_range(self,key):
        if len(self.leafUSet) == 0 and len(self.leafLSet) == 0:
            return False

        if len(self.leafUSet) == 0:
            return compare(key,min_node_id(self.leafLSet),"ge") and
                ↪ compare(key,self.id,'le')
        if len(self.leafLSet) == 0:
            return compare(key,max_node_id(self.leafUSet),'le') and
                ↪ compare(key,self.id,'ge')
        return compare(key,min_node_id(self.leafLSet),'ge') and
            ↪ compare(key,max_node_id(self.leafUSet),'le')

    def find_closest_node_in_routing_table(self,key,
        ↪ node_id_to_object): ## This is tough
        shl = comman_prefix_length(self.id,key.id)
        node = self.routingTable[shl][int(key.id[shl],16)]
        if node is not None and node in node_id_to_object:
            return node
        else:
            for node in self.leafUSet + self.leafLSet:
                if node in node_id_to_object:
                    shl_node = comman_prefix_length(node,key.id)
                    if node_abs_id_distance(node,key.id) <
                        ↪ node_abs_id_distance(node,self.id) and
                        ↪ shl_node >= shl:
                        return node
            for row in self.routingTable:
                for node in row:
                    if node is not None and node in node_id_to_object:
```

```
            shl_node = comman_prefix_length(node,key.id)
            if node_abs_id_distance(node,key.id) <
              ↪ node_abs_id_distance(node,self.id) and
              ↪ shl_node >= shl:
              return node

      return None

  def add_key(self,key,value,node_id_to_object,mode,ct=0): ##
    ↪ mode can be of deletion
    #if self.isOnRoute:
    #print("Come to, ", self.name," ", self.id)
    #if
    ct += 1
    if ct > 100:
      return (None,None,None)
    if mode == "find":
      if key.name in self.data:
        return (self.data[key.name],0,[self.id])

    if_lies_in_node_range = self.check_if_key_lies_in_leaf_range
      ↪ (key.id)
    leaf_node_id = self.findNearestLeafNode(key.id)

    if if_lies_in_node_range and leaf_node_id is not None and
      ↪ leaf_node_id in node_id_to_object:
      leaf_node = node_id_to_object[leaf_node_id]
      if node_abs_id_distance(leaf_node.id,key.id) <=
        ↪ node_abs_id_distance(self.id,key.id):
        val,hops,route = leaf_node.add_key(key,value,
          ↪ node_id_to_object,mode)
        if val is None and hops is None and route is None:
          return (None,None,None)
        return (val,hops+1,[self.id]+route)
      else:
        if mode == "find":
          if key.name in self.data:
            return (self.data[key.name],0,[self.id])
          return (None,0,[self.id])
        else:
          if mode == "find_closest_node":
```

```
                    return (self.id,0,[self.id])

                self.data[key.name] = value
                return (value,0,[self.id])
        routing_node_id = self.find_closest_node_in_routing_table(
           ↪ key,node_id_to_object)
        if routing_node_id is not None and routing_node_id in
           ↪ node_id_to_object:
            routing_table_node = node_id_to_object[routing_node_id]
            #print(routing_table_node.name,routing_table_node.id)
            val,hops,route = routing_table_node.add_key(key,value,
               ↪ node_id_to_object,mode,ct)
            if val is None and hops is None and route is None:
                return (None,None,None)
            return (val,hops+1,[self.id]+route)
        if mode== "addition":
            self.data[key.name] = value
            return (value,0,[self.id])
        if mode == "find_closest_node":
            return (self.id,0,[self.id])
        return (None,0,[self.id])


    def updateLeafNodes(self,node):
        leaf_nodes = node.leafUSet + [node.id] + node.leafLSet
        distances_high = []
        distances_low = []
        for nbr in leaf_nodes:
            if nbr != self.id:
                distance = node_id_distance(nbr,self.id)
                if distance > 0:
                    distances_high.append((nbr,distance))
                else:
                    distances_low.append((nbr,abs(distance)))


        distances_high.sort(key=lambda val: val[1],reverse=False)
        distances_low.sort(key=lambda val: val[1], reverse= False)
        self.leafUSet = [item[0] for item in distances_high[:int(
           ↪ self.L/2)]]
```

```
        self.leafLSet = [item[0] for item in distances_low[:int(self
            ↪ .L/2)]]
        return

    def updateMembershipNodes(self,node,id_to_node):
        local_nodes = node.nbrSet + [node.id]
        distances = []
        for nbr in local_nodes:
            if nbr != self.id:
                distance = euclidean_distance(id_to_node[nbr].location
                    ↪ ,self.location)
                distances.append((nbr,distance))
        distances.sort(key=lambda val: val[1],reverse=False)
        self.nbrSet =[item[0] for item in distances[0:self.M]]
        return

    def updateRoutingTable(self,route):
        ct= 0
        for node in route:
            shl = comman_prefix_length(self.id,node.id)
            if ct == 0:
                for i in range(0,shl+1):
                    self.routingTable[i] = node.routingTable[i]
            else:
                if self.routingTable[shl][0] is None:
                    self.routingTable[shl] = node.routingTable[i]
        return

    def stabilize(self,node_id_to_object):
        #print("self id ,",self.id)
        newleafSet = []
        lost = False
        lost_index_upper = -1
        lost_index_lower = -1
        ct = 0
        removed_node_id = None
        for nodeid in self.leafUSet:
            if nodeid not in node_id_to_object:
                lost = True
                lost_index_upper = ct
                removed_node_id = nodeid
```

```
                break
            ct += 1
    ct = 0
    if lost_index_upper == -1:
        for nodeid in self.leafLSet:
            if nodeid not in node_id_to_object:
                lost = True
                lost_index_lower = ct
                removed_node_id = nodeid
                break
            ct += 1
    len_lset = len(self.leafLSet)
    len_uset = len(self.leafUSet)
    if lost and (len_lset > 0 or len_uset >0):
        ask_leaf_node_from = None

        if lost_index_upper != -1:
            if len_uset > 1:
                if lost_index_upper == len_uset - 1:
                    ask_leaf_node_from = self.leafUSet[len_uset-2]
                else:
                    ask_leaf_node_from = self.leafUSet[len_uset-1]
            else:
                self.leafUSet = []

        else:
            if lost_index_lower != -1:
                if len_lset > 1:
                    #print("index, ", lost_index_lower,len_lset, )
                    if lost_index_lower == len_lset - 1:
                        ask_leaf_node_from = self.leafLSet[len_lset
                            ↪ -2]
                    else:
                        ask_leaf_node_from = self.leafLSet[len_lset
                            ↪ -1]
                else:
                    self.leafLSet = []
        if ask_leaf_node_from is not None:
            ask_leaf_node_from = node_id_to_object[
                ↪ ask_leaf_node_from]
```

```
        newLeafSet = list(set(self.leafLSet + self.leafUSet +
            ↪ ask_leaf_node_from.leafLSet + ask_leaf_node_from
            ↪ .leafUSet))
        newLeafSet.remove(removed_node_id)
        distances_high = []
        distances_low = []
        for nbr in newLeafSet:
            if nbr != self.id:
                distance = node_id_distance(nbr,self.id)
                #print(distance)
                if distance > 0:
                    distances_high.append((nbr,distance))
                else:
                    distances_low.append((nbr,abs(distance)))


        distances_high.sort(key=lambda val: val[1],reverse=
            ↪ False)
        distances_low.sort(key=lambda val: val[1], reverse=
            ↪ False)
        self.leafUSet = [item[0] for item in distances_high[:
            ↪ int(self.L/2)]]
        self.leafLSet = [item[0] for item in distances_low[:
            ↪ int(self.L/2)]]
        #print(len(self.leafUSet),len(self.leafLSet))

else:
    if lost:
        self.leafUSet = []
        self.leafLSet = []



#if not lost:
    #### repair routing table
row_index = 0
import copy
dup = copy.deepcopy(self.routingTable.copy())
dup1 = copy.deepcopy(self.routingTable.copy())
for row in dup:
    for node in row:
```

```
            col_index = 0
            if node is not None and node not in node_id_to_object:
                found = False
                #print("Found, ", self.id, row_index,col_index,
                    ↪ node)
                # node is dropped###
                self.routingTable[row_index][col_index] = None

                for new_row in dup1:
                    for new_node in new_row:
                        if not found:
                            if new_node is not None and new_node in
                                ↪ node_id_to_object and new_node !=
                                ↪ node and node_id_to_object[new_node
                                ↪ ].routingTable[row_index][col_index]
                                ↪  is not None and node_id_to_object[
                                ↪ new_node].routingTable[row_index][
                                ↪ col_index] in node_id_to_object:
                                #print(row_index,col_index)
                                self.routingTable[row_index][col_index]
                                    ↪  = node_id_to_object[new_node].
                                    ↪ routingTable[row_index][col_index
                                    ↪ ]
                                found = True
                                #print(self.routingTable[row_index][
                                    ↪ col_index])
            col_index +=1
        row_index += 1

    return

def updateState(self,new_node,node_id_to_object):
    self.updateLeafNodes(new_node)
    self.updateMembershipNodes(new_node,node_id_to_object)
    shl = comman_prefix_length(self.id,new_node.id)
    location = int(new_node.id[shl],16)

    existing_node_id = self.routingTable[shl][location]
    if existing_node_id is None:
        self.routingTable[shl][location] = new_node.id
    else:
```

```
            existing_node = node_id_to_object[existing_node_id]
            if euclidean_distance(new_node.location,self.location) <
                ↪ euclidean_distance(existing_node.location,self.
                ↪ location):
                self.routingTable[shl][location] = new_node.id

        return
    def print_node_properties(self):
        print("Name," , self.name)
        print("Id,", self.id)
        print("location, ", self.location)
        print("leaf Upper set",self.leafUSet)
        print("leaf Lower set", self.leafLSet)
        print("nbr set", self.nbrSet)
        print("Routing Table")
        df = pd.DataFrame.from_records(self.routingTable)
        display(df)



class Pastry():
    def __init__(self,num_of_nodes):
        self.N = num_of_nodes
        self.nodes = []
        self.node_id_to_object = {}
        for i in range(0, num_of_nodes): ### Creating nodes
            node = Node(i)
            self.nodes.append(node)
            self.node_id_to_object[node.id] = node
        for i in range(0,num_of_nodes):
            self.updateLeafNodes(self.nodes[i])
            self.updateLocalNode(self.nodes[i])
            self.updateRoutingTable(self.nodes[i])


        self.add_queries = 0
        self.delete_queries = 0
        self.search_queries = 0
        self.data_add_queries = 0
    def findNearestNode(self,node):
        nearest_node = None
```

```
    nearest_node_dist = inf
    for nbr in self.nodes:

        distance = node_abs_id_distance(nbr.id,node.id)
        if nearest_node_dist > distance:
            nearest_node_dist = distance
            nearest_node = nbr
    return nearest_node

def updateRoutingTable(self,node):
    routingTable = [[ [] for item in range(0,int(math.pow(2,node.b)))] for i in range(0,int(128/node.b))] ## b = 4 configuration
    for nbr in self.nodes:
        if nbr.id != node.id:
            shl = comman_prefix_length(node.id,nbr.id)
            routingTable[shl][int(nbr.id[shl],16)].append(nbr.id)
    for i in range(0,len(routingTable)):
        for j in range(0,len(routingTable[i])):
            if len(routingTable[i][j]) == 0:
                node.routingTable[i][j] = None
            else:
                distances = []
                for nbr in routingTable[i][j]:
                    distances.append((nbr,euclidean_distance(self.node_id_to_object[nbr].location,node.location)))
                distances.sort(key=lambda val:val[1],reverse= False)
                node.routingTable[i][j] = distances[0][0] ## stored the closest nbr node
    return

def updateLocalNode(self,node):
    distances = []
    for nbr in self.nodes:
        if nbr.id != node.id:
            distance = euclidean_distance(nbr.location,node.location)
            distances.append((nbr.id,distance))
    distances.sort(key=lambda val: val[1],reverse=False)
```

```
        node.nbrSet =[item[0] for item in distances[0:node.M]]
        return



    def updateLeafNodes(self,node):
        distances_high = []
        distances_low = []
        for nbr in self.nodes:
            if nbr.id != node.id:
                distance = node_id_distance(nbr.id,node.id)
                if distance > 0:
                    distances_high.append((nbr.id,distance))
                else:
                    distances_low.append((nbr.id,abs(distance)))


        distances_high.sort(key=lambda val: val[1],reverse=False)
        distances_low.sort(key=lambda val: val[1], reverse= False)
        node.leafUSet = [item[0] for item in distances_high[:int(
            ↪ node.L/2)]]
        node.leafLSet = [item[0] for item in distances_low[:int(node
            ↪ .L/2)]]
        return

    def add_key(self,key,value,mode="addition",ct=0):
        if mode == "addition":
            self.add_queries += 1
        if mode == "find":
            self.search_queries += 1

        node_index= random.choice(range(0,self.N))
        return self.nodes[node_index].add_key(key,value,self.
            ↪ node_id_to_object,mode,ct)

    def add_node(self):
        node = Node(self.N+1)
        self.node_id_to_object[node.id] = node
        self.nodes.append(node)
        random_node= self.nodes[random.choice(range(0,self.N))]
        self.N += 1
```

```
        nearest_node,hops,route = random_node.add_key(node,'',self.
            ↪ node_id_to_object,"find_closest_node")
        node.updateLeafNodes(self.node_id_to_object[nearest_node])
        node.updateMembershipNodes(self.node_id_to_object[
            ↪ nearest_node],self.node_id_to_object)
        node.updateRoutingTable([self.node_id_to_object[id] for id
            ↪ in route])


        for nbr in node.leafLSet + node.leafUSet + node.nbrSet + [
            ↪ item for row in node.routingTable for item in row ]:
            if nbr is not None:
                nbr_node = self.node_id_to_object[nbr]
                nbr_node.updateState(node,self.node_id_to_object)


        return "success"
    def delete_node(self):
        self.delete_queries += 1
        index= random.choice(range(0,self.N))
        del self.node_id_to_object[self.nodes[index].id]
        del self.nodes[index]
        self.N = self.N-1
        for node in self.nodes:
            node.stabilize(self.node_id_to_object)
        return
    def print_network_information(self):
        print("Number of active nodes , " , len(self.nodes))
        print("Number of data add queries, ", self.add_queries)
        print("Number of delete node queries" , self.delete_queries)
        print("Number of search queries, ", self.search_queries)
        data_len= 0
        for node in self.nodes:
            data_len += len(node.data)
        print("Current data stored ", data_len)
num_of_nodes = [100,500,1000]
hops_per_nodes = []
for nodes in num_of_nodes:
    print("Creating network for ", nodes)
    pastry = Pastry(nodes)
    print("adding data points")
```

```
keys = []
for i in range(0,10000):
   key = Key(i)
   keys.append(key)
   pastry.add_key(key,"val_"+str(key.name),"addition")
hops_needed = []
print("Searching,")
for i in range(0,1000000):
   key = random.choice(keys)
   hops = pastry.add_key(key,"","find",0)[-1]
   if hops is not None:
      hops_needed.append(len(hops))
hops_per_nodes.append(np.mean(hops_needed))
vals,probs = get_prob_distribution(hops_needed)
pastry.print_network_information()
plt.clf()
plt.bar(vals, probs,width=0.3)
#plt.hist(hops_needed) # density
plt.ylabel('Probability')
plt.xlabel('Number of hops')
plt.title("Distribution of hops required for 1 million queries
   ↪ in "+str(nodes) +" size network")
plt.savefig(str(nodes)+"_search_queries.svg")
print("deleting 50% nodes")
for i in range(0,int(nodes*.5)):
#print("deleting , i ", i)
   pastry.delete_node()
print("searching for queries")
for i in range(0,1000000):
   key = random.choice(keys)
   hops = pastry.add_key(key,"","find",0)[-1]
   if hops is not None:
      hops_needed.append(len(hops)+1)
vals,probs = get_prob_distribution(hops_needed)
plt.clf()
plt.bar(vals, probs,width=0.3)
#plt.hist(hops_needed) # density
plt.ylabel('Probability')
plt.xlabel('Number of hops')
plt.title("Distribution of hops required for 1 million queries
   ↪ after deletion")
```

```
    plt.savefig(str(nodes)+"_delete_search_queries.svg")

plt.clf()
plt.plot(num_of_nodes, hops_per_nodes,'o-')
#plt.hist(hops_needed) # density
plt.ylabel('Avg. hops required')
plt.xlabel('Network size')
plt.title("Avg hops vs Pastry network size")
plt.savefig("pastry_avg_hops_needed.svg")
```

7 Chord Python code

```
import hashlib
import random
import math
import numpy as np
import math
import pandas as pd
from collections import Counter
inf = math.inf
def hex_id(id):
    return hashlib.md5(str(id).encode()).hexdigest()
def random_num(a,b):
    return a + random.random()*(b-a)



def power_m(M):
    return int(math.pow(2,M))

def in_range(item,id1,id2,M):
    #print("in range, ", item, id1, id2 )
    if id1 is None or id2 is None or item is None:
        return False
    item = item%int(power_m(M))
    id1 = id1%int(power_m(M))
    id2 = id2%int(power_m(M))
    if id1 == id2:
        return id1 == item
    if id1 < id2:
        return item >= id1 and item<= id2
    else:
```

```
        return item >= id1 or item <= id2

def get_prob_distribution(lst):

    a = Counter(lst)
    ct = sum(a.values())

    vals = list(a.keys())
    probs = [item*1.0/ct for item in a.values()]
    idx = np.argsort(vals)
    vals = np.array(vals)[idx]
    probs = np.array(probs)[idx]
    return vals,probs




global id_to_node
id_to_node = {}

class Finger():
    def __init__(self,start=None,interval=None,node_id=None):
        self.start = start
        self.interval = interval
        self.node = node_id
        return

class Node():
    def __init__(self,id,M):
        self.id = id
        self.M = M
        self.finger = []
        self.data= {}
        for i in range(0,self.M):
            start = (id + int(math.pow(2,i)))%int(math.pow(2,self.M))
            end = (id + int(math.pow(2,i+1)))%int(math.pow(2,self.M))
            self.finger.append(Finger(start,(start,end)))
        self.successor = None
        self.predecessor = None
```

```
def print_finger_table(self):
    print("Node id,", self.id)
    print("start interval successor")
    for i in range(0,self.M):
        print(self.finger[i].start, " " ,self.finger[i].interval, " ", self.finger[i].node)
    print("Predecesssor, ", self.predecessor)
    print("Successor, ", self.successor)
    print("###########################")
    return

def find_successor(self,id,route=False):
    #if id == self.id:
        #return self.id
    if route:
        n_,route_ = self.find_predecessor(id,route)
        return id_to_node[n_].successor,route_ + [id_to_node[n_].successor]
    else:
        n_ = self.find_predecessor(id,route)
        return id_to_node[n_].successor
    #print("predecessor,", n_)



def find_predecessor(self,id,route=False):
    n_ = self.id
    route_ = [n_]
    #print("find predecessor key, current node , successor ", id ,n_,id_to_node[n_].successor)
    while not in_range(id,n_+1,id_to_node[n_].successor,self.M):
        #print("loop")
        n_ = id_to_node[n_].closest_preceding_finger(id)
        #print("closest preceding finger, ",n_)
        route_.append(n_)
    if route:
        return n_,route_
    else:
        return n_
def closest_preceding_finger(self,id):
    for i in range(self.M-1,-1,-1):
```

```
        #print("in closest precedding, ", self.finger[i].node,
            ↪ self.id,id)
        if in_range(self.finger[i].node,self.id+1,id-1,self.M):
        #if in_range(id,self.finger[i].interval[0],self.finger[i
            ↪ ].interval[1]-1,self.M):
            return self.finger[i].node

def join(self,n_):
    if n_ not in id_to_node: ### that means it is first node to
        ↪ join###
        for i in range(0,self.M):
            self.finger[i].node = self.id
        self.predecessor = self.id
        self.successor = self.id
    else: ### n_ exists
        #print("initializing")
        self.initialize_finger(n_)
        #print(id_to_node[n_].print_finger_table())
        #print("updating other nodes")
        self.update_other_nodes()
def initialize_finger(self,n_):
    n_ = id_to_node[n_]
    #print(self.finger[0].interval)
    self.finger[0].node = n_.find_successor(self.finger[0].start
        ↪ )
    self.successor = self.finger[0].node
    self.predecessor = id_to_node[self.successor].predecessor
    id_to_node[self.successor].predecessor = self.id

    for i in range(1,self.M):
        self.finger[i].node = n_.find_successor(self.finger[i].
            ↪ start)

def update_other_nodes(self):
    for i in range(0,self.M):
        if self.id-int(math.pow(2,i)) == self.predecessor:
            id_to_node[self.predecessor].update_finger_table(self.
                ↪ id,i)
        else:
            #print("key to find predecessor", self.id-int(math.pow
                ↪ (2,i)))
```

```
            p= self.find_predecessor(self.id-int(math.pow(2,i)))
            #print("in update other, ",p)
            id_to_node[p].update_finger_table(self.id,i)
    def update_finger_table(self,node,i):
        #print("here i , self, self.finger[i] node, self predec",i ,
            ↪ self.id,self.finger[i].node,self.predecessor)
        if self.id == node:
            return
        if in_range(node,self.id,self.finger[i].node-1,self.M):
            self.finger[i].node = node
            if i == 0:
                self.successor = node
            id_to_node[self.predecessor].update_finger_table(node,i)


    def add_key(self,key,value):
        n_ = self.find_successor(key)
        id_to_node[n_].add_key_to_itself(key,value)
        return
    def add_key_to_itself(self,key,value):
        self.data[key] = value
        return
    def return_val_for_key(self,key):
        if key in self.data:
            return self.data[key]
        return None
    def find_key(self,key):
        n_,route = self.find_successor(key,route = True)
        return (id_to_node[n_].return_val_for_key(key), route)
        #print(len(route))

    def update(self,delete_node,successor,predecessor):
        if self.successor == delete_node:
            self.successor = successor
        if self.predecessor == delete_node:
            self.predecessor = predecessor

        for finger in self.finger:
            if finger.node == delete_node:
                finger.node = successor
```

```
    def delete(self):
        successor = self.successor
        predecessor = self.predecessor
        for node in id_to_node:
            if node != self.id:
                id_to_node[node].update(self.id,successor,predecessor)

        return




class Chord():
    def __init__(self,N):
        print("Adding nodes")
        self.seed = False
        M = 20
        for i in range(0,N):
            id = random.choice(range(0,int(pow(2,M))))
            if i ==0:
                self.seed = id
            node = Node(id,M)
            id_to_node[id] = node
            if i == 0:
                node.join(0)
            else:
                node.join(self.seed)
        self.keys = []

        print("Adding keys")
        for i in range(0,10000):
            key = random.choice(range(0,int(pow(2,20))))
            self.keys.append(key)
            id_to_node[self.seed].add_key(key,str(key)+"_val")
    def find_key(self,key=None):
        if key is None:
            key = random.choice(self.keys)
        if self.seed in id_to_node:
            val,route = id_to_node[self.seed].find_key(key)
        else:
```

```
        val,route = id_to_node[list(id_to_node.keys())[0]].
            ↪ find_key(key)
        if route is not None:
            return len(route)
        else:
            return None
    def find_key_and_route(self):
        key = random.choice(self.keys)
        if self.seed in id_to_node:
            val,route = id_to_node[self.seed].find_key(key)
        else:
            val,route = id_to_node[list(id_to_node.keys())[0]].
                ↪ find_key(key)
        route = " -> ".join([str(item) for item in route])
        print("Lookup route of key,", str(key)," :",route)
    def delete_nodes(self,N):
        import random
        for i in range(0,N):
            node_id = random.choice(list(id_to_node.keys()))
            id_to_node[node_id].delete()
            del id_to_node[node_id]




if __name__ == "__main__":


    num_of_nodes = [100,500,1000]
    hops_per_nodes = []
    for nodes in num_of_nodes:
        print("Creating network for ", nodes)
        chord = Chord(nodes)
        hops_needed = []
        print("Searching,")
        for i in range(0,1000000):
            key = random.choice(chord.keys)
            hops = chord.find_key(key)
            if hops is not None:
                hops_needed.append(hops)
        hops_per_nodes.append(np.mean(hops_needed))
```

```
    vals,probs = get_prob_distribution(hops_needed)
    plt.clf()
    plt.bar(vals, probs,width=0.3)
    #plt.hist(hops_needed) # density
    plt.ylabel('Probability')
    plt.xlabel('Number of hops')
    plt.title("Distribution of hops required for 1 million
        ↪ queries in "+str(nodes) +" size network")
    plt.savefig(str(nodes)+"_chord_search_queries.svg")
    print("deleting 50% nodes")
    chord.delete_nodes(int(nodes*.5))
    print("searching for queries")
    for i in range(0,1000000):
        key = random.choice(chord.keys)
        hops = chord.find_key(key)
        if hops is not None:
            hops_needed.append(hops)
    vals,probs = get_prob_distribution(hops_needed)
    plt.clf()
    plt.bar(vals, probs,width=0.3)
    #plt.hist(hops_needed) # density
    plt.ylabel('Probability')
    plt.xlabel('Number of hops')
    plt.title("Distribution of hops required for 1 million
        ↪ queries after deletion")
    plt.savefig(str(nodes)+"_chord_delete_search_queries.svg")

plt.clf()
plt.plot(num_of_nodes, hops_per_nodes,'o-')
#plt.hist(hops_needed) # density
plt.ylabel('Avg. hops required')
plt.xlabel('Network size')
plt.title("Avg hops vs Pastry network size")
plt.savefig("chord_avg_hops_needed.svg")




for i in range(0,10):
    chord.find_key_and_route()
    print()
```

```
id_to_node[list(id_to_node.keys())[0]].print_finger_table()
```