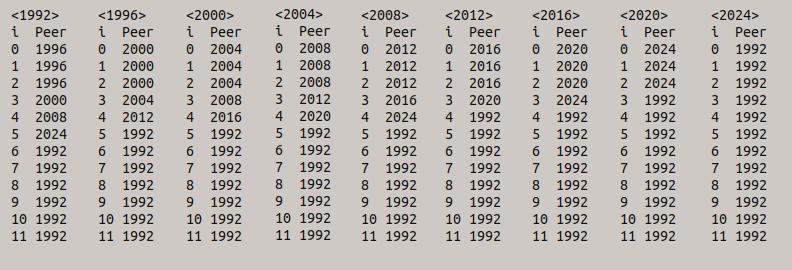
**Problem 9**

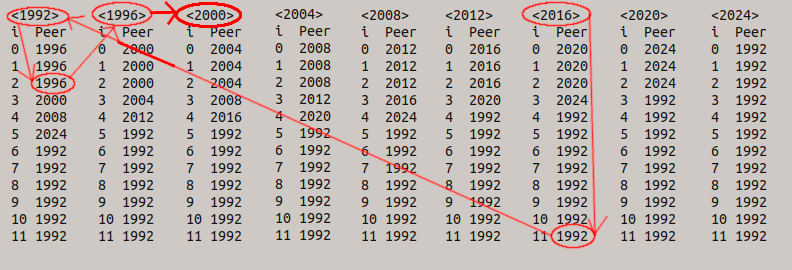
**Part A:**

Only node 2016 was asked for, however I’ve done finger tables for all nodes:



**Part B:**

Following the rules presented in lecture, the key 1999 should be mapped to 2000. The map pathway from 2016 to 2000 is illustrated below (note that 1992 and 1996 are visited only once):



This pathway works out to: 2016 → 1992 → 1996 → 2000

Starting from 2016, its finger table peer 1992 is the largest while less than the K value.

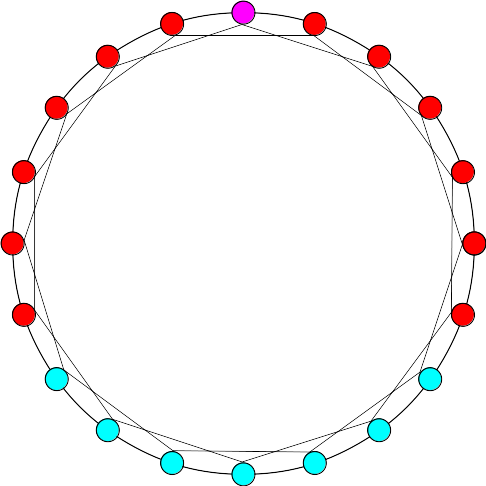
At 1992, its largest finger table peer that is less than K is 1996.

At 1996, it has no finger table peer that is less than K so it returns its successor, 2000.

**Part C:**

All nodes whose finger tables contain 2020 or have 2020 as a successor need to be updated. From the above, this comes out to: 2004, 2012, 2016.

**Problem 8**

**Part A:**

To the right is an illustration of the 20 processes. The magenta node is the Query origin while the red nodes are those it’s able to reach with a TTL of 3. Counting 12 red and 1 magenta makes 13 total nodes infected with the Query.

**Part B:**

To solve this, it is sufficient to count how many jumps through secondary neighbors it takes to get from the origin to the other side of the ring. Starting with the example in Part A, the furthest infected node from the origin is just 2 secondary neighbor hops from the opposite side of the magenta node. Thus, a minimum TTL of 3 + 2 = 5 is required to reach all processes in the network. (Note: Primary neighbors are immediately adjacent, while secondary neighbors are two nodes over on either side of a node)

**Part C:**

When a “Complete Node” is added (IE, a node that has an edge to all other nodes), the minimum TTL required for a Query to reach all nodes depends on the origin of the Query. If the “Complete Node” is the origin, the minimum TTL required is just 1. If any of the other nodes are the origin, then the minimum TTL is 2.

**Problem 10**

One way to reduce the shuffle traffic and “break the barrier” would be a ***binning*** paradigm. If the keys are known and can be sorted, then the keys can be partitioned into ranges. As an example, suppose the keys were integers from 1 to 100. We could partition these keys into 10 ranges of 10. Below would be the first 2 ranges:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | | | | | | | | | | | 1 | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Note that the top row numbers are the indices of the corresponding ranges of original keys, which are in the second row. EG, all original keys from 1 to 10 are mapped to the range index 0, 11 to 20 are mapped to 1, and so on.

A hash table can be used to map original keys to the corresponding range index. Mappers can use this hash table to produce key/value pairs where the range index is the key and the value would be the original key/value pair. With a smaller number of keys, fewer Reducers are required and less strain on the shuffling phase will result. However, more complex reduce functions will be required to handle the more complex key/value pairs. While each Reducer can be assigned to one of the range indices, they will now need to maintain records of the original key/value pairs that fall into the Reducer’s range index, something like a hash table with the original key mapped to the corresponding original value. What’s nice about this though is Reducers don’t have to wait for the Mappers to completely finish, thereby completely “breaking the barrier”, before initiating reductions. Incoming shuffle data can be read and if there’s already a record of the key in the Reducer’s hash table it can update/reduce its corresponding record.

(Binning idea inspired by this video: https://www.coursera.org/learn/hadoop/lecture/WAnTz/computational-costs-of-vector-multiplication)

**Problem 1**

This problem can be interpreted, in the context of graph theory, as counting all of the 4-Cliques within a graph, which reduces to the [Clique Problem](https://en.wikipedia.org/wiki/Clique_problem). As such it is NP-complete and will therefore be challenging to come up with an efficient algorithm. Fortunately there are already decent MapReduce algorithms proposed for solving this kind of problem, notably [[I Finocchi et al]](https://dl.acm.org/citation.cfm?id=2794080) and [[M Danisch et al]](https://papers-gamma.link/static/memory/pdfs/32-main.pdf). However the former assumed that the degree of the nodes are known and the latter was too complicated for me to understand. Problem 1 does not specify knowledge of node degree, so I will assume it cannot be determined easily. While not ideal, I shall instead propose a naïve, semi-brute force method requiring 4 rounds of MapReduce. In brief, the first round maps each node to a list of their target nodes (an adjacency list), second round computes 2-Cliques (bidirectional edges), third round computes 3-Cliques, and the fourth round computes 4-Cliques.

Define Map1( (a , b)):

// Note: (a, b) represents a directional edge a to b

shuffle (a : b)

// sends a mapping from node a to node b

Define Reduce1( list( a : \_ ) ):

// Note: ‘\_’ is a wildcard, so the input is a list of key/value pairs that have same key ‘a’ mapped // to any node.

let L = list() // empty list

for each <node K : node V> in list(a: \_ )

append V to L

return (a : L)

// returns a node mapped to a list of nodes, the targets of a

Define Map2 ( a : L):

for each node V in L

if a < V then shuffle ( (a,V) : L – {V} ), continue

else shuffle ( (V, a) : L – {V} )

// sends a sorted 2-tuple of nodes, mapped to a list of nodes

Define Reduce2( list( ( a, b) : \_ ))

// Note: ‘\_’ is a wildcard, input is a list of key/value pairs with the same 2-tuple key mapped to // various lists of nodes.

// Also note that a and b will be in chronological order, due to the boolean logic in Map2.

if | list( ( a, b) : \_ ) | < 2

return // do nothing

let Li = the value of the first key/value pair, which is a list of nodes

for each <2-tuple K : node list V> in list( ( a, b) : \_ )

Li = Li ∩ V

return ( (a, b) : Li )

// returns a sorted 2-tuple of nodes, a bidirectional edge, mapped to a list of common nodes (a and b

// both have an edge to each node in Li )

**Problem 1 (cont)**

Define Map3 ( (a, b) : L )

for each node V in L

if V < a then shuffle ( (V, a, b) : L – {V} ), continue

if V < b then shuffle ( (a, V, b) : L – {V} ), continue

else shuffle ( (a, b, V) : L – {V} )

// sends a sorted 3-tuple of nodes, mapped to a list of nodes

Define Reduce3 ( list( (a, b, c) : \_ ) )

// Note: ‘\_’ is a wildcard, input is a list of key/value pairs with the same 3-tuple key mapped to // various lists of nodes.

// Also note that a, b, c will be in chronological order, due to the boolean logic in Map2 and // Map3.

if | list( ( a, b, c) : \_ ) | < 3

return // do nothing

let Li = the value of the first key/value pair, which is a list of nodes

for each <3-tuple K : node list V> in list( ( a, b, c) : \_ )

Li = Li ∩ V

return ( (a, b, c) : Li )

// returns a sorted 3-tuple of nodes, a 3-Clique, mapped to a list of common nodes (a,b,c all have an

// edge to each node in Li )

Define Map4 ( (a, b, c) : L )

for each node V in L

if V < a then shuffle ( (V, a, b, c) : L – {V} ), continue

if V < b then shuffle ( (a, V, b, c) : L – {V} ), continue

if V < c then shuffle ( (a, b, V, c) : L – {V} ), continue

else shuffle ( (a, b, c, V) : L – {V} )

// sends a sorted 4-tuple of nodes, mapped to a list of nodes

Define Reduce4 ( list( (a, b, c, d) : \_ ) )

// Note: ‘\_’ is a wildcard, input is a list of key/value pairs with the same 4-tuple key mapped to // various lists of nodes.

// Also note that a, b, c, d will be in chronological order, due to the boolean logic in Map2, // Map3, and Map4.

if | list( ( a, b, c, d) : \_ ) | < 4

return // do nothing

else return (a, b, c, d)

// returns a sorted 4-tuple of nodes, a 4-Clique

**Problem 1 (cont)**

Using the 8 functions defined above, the following procedure can be performed:

Input is read from HDFS into the Mappers, each performs Map1, whose output is shuffled to the Reducers loaded with Reduce1 creating an adjacency list. The entries of the adjacency list is fed to Mappers loaded with Map2, whose output is shuffled to the Reducers loaded with Reduce2 creating a list of bidirectional edges. This is then fed into another set of Mappers loaded with Map3, whose output is shuffled to the Reducers loaded with Reduce3 creating a list of 3-Cliques. This is then fed into a final set of Mappers loaded with Map4, whose output is shuffled to the Reducers loaded with Reduce4 who then output the final list of 4-Cliques, the desired quadruplets of twitter users.

The proposed method of four rounds of MapReduce will not perform well with large near complete graphs, but it should do well with sparse graphs. Since social media sites’ user/follower data are typically relatively sparse graphs, this should do well enough.

**Problem 2**

This problem can have a similar start with problem 1. In total my method will use 3 rounds of MapReduce plus an initial filter step. Briefly:

1. an initial filter round to get just @packers followers,
2. a round to generate adjacency lists,
3. a round to determine pairs of disconnected nodes, ie nonexistent edges, and
4. a final round to remove duplicates.

Define prefilter ( (a, b)):

if a follows @packers and b follows @packers

then return (a, b)

Define Map1( (a , b)):**// NOTE same as problem 1**

// Note: (a, b) represents a directional edge a to b

shuffle (a : b)

// sends a mapping from node a to node b

Define Reduce1( list( a : \_ ) ):**// NOTE same as problem 1**

// Note: ‘\_’ is a wildcard, so the input is a list of key/value pairs that have same key ‘a’ mapped // to any node.

let L = list() // empty list

for each <node K : node V> in list(a: \_ )

append V to L

return (a : L)

// returns a node mapped to a list of nodes, the targets of a

Define Map2 ( a : L): ***// SUPER IMPORTANT: This is DIFFERENT from problem 1***

for each node V in L

if a < V then shuffle ( (a,V) : L), continue // note L, instead of L – {V} !!!!!

else shuffle ( (V, a) : L ) // note L, instead of L – {V} !!!!!

// sends a sorted 2-tuple of nodes, mapped to a list of nodes

**Problem 2 (cont)**

Define Reduce2( list( ( a, b) : \_ ))

// Note: ‘\_’ is a wildcard, input is a list of key/value pairs with the same 2-tuple key mapped to // various lists of nodes.

// Also note that a and b will be in chronological order, due to the boolean logic in Map2.

if | list( ( a, b) : \_ ) | < 2

return // do nothing

//There should only be two key/value pairs, meaning two lists of nodes to examine

let L1= the value of the first key/value pair, which is a list of nodes

let L2 = the value of the second key/value pair, another list of nodes

// identify the origin node for first key/value pair

let O1 = {a, b} - L1

// identify the origin node for second key/value pair

let O2 = {a, b} - L2

L1 = L1 ∪ O1

L2 = L2 ∪ O2

let NEE = list() //empty list, will hold ***N***on***E***xistent ***E***dges

for each node N in L2 plus O2 that is not in L1 plus O1:

if N < O1 then NEE ∪ {(N, O1)}, continue

else NEE ∪ {( O1, N)}

for each node N in L1 plus O1 that is not in L2 plus O2:

if N < O2 then NEE ∪ (N, O2), continue

else NEE ∪ ( O2, N)

return NEE

// returns a variable length list of non existent edges, nodes sorted

Define Map3 ( NEE ):

// Note: NEE is a list of nonexistent edges

for each edge E in NEE:

shuffle (E : E)

// sends an edge mapped to itself, an identity

Define Reduce3( list(E : E) )

//Note: receives a list of key/value pairs in which both key and value are the same edge

return E

**Problem 2 (cont)**

Using the functions defined above, we start with reading the data from the HDFS and filtering out only those (a,b) lines in which both a and b follow the @packers using the prefilter function. The filtered input is read into the Mappers, each performs Map1, whose output is shuffled to the Reducers loaded with Reduce1 creating an adjacency list. The entries of the adjacency list is fed to Mappers loaded with Map2, whose output is shuffled to the Reducers loaded with Reduce2 creating a list of nonexistent edges. These lists are then fed into another set of Mappers loaded with Map3, whose output is shuffled to the Reducers loaded with Reduce3 to filter out duplicate nonexistent edges, yielding the desired list of @packers followers who do not follow each other.

**Problem 2**:

This problem can be thought of as finding pairs of nodes not connected in a graph. Here is a brief overview of the procedure to solve it:

1. MapReduce to accumulate a list of all nodes, a master list
2. MapReduce with input data plus the master list to create a “non” adjacency list
3. MapReduce to get “bidirectional” nonexistent edges, pairs of disconnected nodes

Define Map1( (a, b) ):

shuffle ( 0 : list(a, b))

Define Reduce1 ( list ( 0 : \_ ))

// input is a list of key/value pairs that all have the same key, 0. The values, ‘\_’, are lists of // nodes

let L = list() //empty list

for <int K : node list V> in list (0: \_ )

for node N in V

if L does not contain N

append N to L

return sorted(L)

// Returns a list of all of the followers

Define Map2 ( (a, b), L)

// L is the master list of nodes, (a,b) will be any data line read from the HDFS

shuffle ( a : ( b, L))

//sends a mapping from node a to a 2-tuple containing target node b and the master list of nodes

Define Reduce2( list( a : ( \_, L))

// input is a list of key/value pairs where the common key is node a, and the values will be 2- // tuples where the first entry, ‘\_’, is any target node and L is the master list of nodes in the // system.

// NAD will contain the nodes ***N***on***AD***jacent to node a

// AD will contain the nodes ***AD***jacent to node a

let NAD = list()

let AD = list()

// cycle through the key/value pairs in the input list

for <node K : 2-tuple (node N, L) > in list (a : (\_, L))

append N to AD

for node N in L

if AD contains N

continue

else

append N to NAD

return ( a : NAD)

//returns a nonadjacency list of the corresponding node a (ie, a does not point to any node in NAD)

**Problem 2 (cont)**:

Define Map3 ( a : NAD):

for each node V in NAD

if a < V then shuffle ( (a,V) : (a,V) ), continue

else shuffle ( (V, a) : (V, a) )

// sends a sorted 2-tuple of nodes that are potentially not connected, mapped to itself

Define Reduce3( list( ( a, b) : ( a, b) ))

// input is a list of keys mapped to itself, a pair of nodes not adjacent to each other

// if the two nodes are not connected, the list will have two items in it

if | list( ( a, b) : ( a, b) )| != 2

return // do nothing

if a does not follow @packers and b does not follow @packers

return // do nothing

return (a, b)

// returns a sorted 2-tuple of nodes that are not connected.

The first round of MapReduce accumulates all twitter users from the HDFS into a master list of followers. Hence why the Map1 function will shuffle everything to the same Reducer with the common key “0”. This master list will be stored on the HDFS and then read into the next round of MapReduce. Mappers loaded with Map2 will take each original data line from the HDFS plus the master list of users and will create a key/value pair where the source user is the key and the value will be a tuple containing the target user and the master list of users. These will be shuffled to the Reducers who will generate a “non”adjacency list for each corresponding user, stored onto the HDFS. The final round of MapReduce starts with the Mappers loaded with Map3 who will read in the “non”adjacency lists and will generate sorted 2 tuples that represent potential disconnected users. If they are not connected, then there will be exactly two of these sorted 2-tuples for each pair of disconnected users. These are mapped to themselves and shuffled to the Reducers loaded with Reduce3 who will determine if both users follow @packers and if there’s exactly two 2-tuples representing the pair. If so, then they will output the sorted 2-tuple representing the pair of disconnected users who both follow @packers.

**Problem 3:**

Here is a brief overview of the procedure to solve it:

1. MapReduce to accumulate those that have >107 followers who aren’t KJ, creating the Big list
2. Feed the Big list with the original data to another round of MapReduce to filter out those that follow KJ

**Problem 4:**

**Part A:**

Assuming that each process’ random k targets that are not among its k predecessors or k successors, completeness will be violated when M = 3k. Consider the case when all of a process’ targets fail. Then the rest of the system will be unable to detect if that process has failed. Since that process was connected to 3k targets, then completeness fails at M = 3k.

**Part B:**

The algorithm is not 100% accurate because no heartbeat protocol can ever be 100% accurate. Heartbeats from a healthy node may be dropped (Week 3 Quiz Problem 3)

**Part C:**

Since each process is connected to a constant number of processes, the load per process will be linear on k. Each process is connected to 3k targets meaning it needs to send out 3k heartbeats per period, hence the load per process = 3k / T = 3k / 1 = 3k.

Since load is constant, the worst case, best case, and average loads are all = 3k.

**Problem 5:**

**Part A:**

This protocol is something like a hybrid of a SWIM and a Gossip protocol, except there’s no forwarding of messages. The process will be complete, weakly much sooner than strongly. It may take a while however, could take more than one ping period. But since each process randomly picks m different processes to ping every ping period, the entropy implied by uniform randomness suggests that eventually at least one process will detect a failed process. The same can be said for ***all*** processes, but this will likely take a much longer amount of time to accomplish, depending on m and T, but ***not*** N. (see part b)

**Part B:**

The expected detection time can be computed in a similar manner to the example in lecture 2.5. Consider some process pi and there’s one failed process pf. If pi sends out m pings, the probability that one of those pings does not uncover pf is (N-1)/N. The probability that all m pings do not detect the failed process will then be [(N-1)/N]m, and subsequently the probability that at least one of those pings uncovers the failed process will be 1 – [(N-1)/N]m. This is the main difference from the example calculation from lecture, which had probability of at least one ping detecting the failed process at

1 – 1/N instead. Note that when m = 1, the above derived probability will reduce to 1 – 1/N. Then to compute the expected detection time we just follow the same steps from lecture with 1 – [(N-1)/N]min place of the 1 – 1/N. First we compute the probability that pf is pinged in T:

Note that once again, if m = 1 it will simplify to the corresponding value determined in lecture. Once we get here then the same calculation from lecture can be used to compute the E[T], which comes out to E[T] = T · em / (em – 1 ). Note that the expected detection time does not depend on N, so long as N is sufficiently large, but will depend on T and m. Also note that it simplifies to the value in lecture when m = 1.

**Problem 6:**

Since the proposed protocol is supposed to have the same behavior as the gossip model discussed in lecture , then we know that dx/dt = -βxy and dy/dt = βxy where β = B / N, B is the number of neighbors gossiped to, and N is the total number of processes. In the proposed protocol we can imagine each process’s membership list to be like a mini version of the Gossip model discussed in lecture. As such, the probability of infecting uninfected processes at each gossip period will be m / k, as m are being selected and there’s a total of k to choose from in the membership list. Then, if we sum these all up for each process, the total is mN / k. This is what B equals. Alternatively, each sub group of processes who are in another process’s membership list should by themselves behave like the gossip protocol. Meaning: β = B / N = m / k ⇒ B = mN / k

**Problem 3:**

Here is a brief overview of the procedure to solve this problem:

1. MapReduce to create DHT mapping user to 2-tuple containing lists of follows and followers
2. Boolean MapReduce to determine the users that meet the conditions

Define Map1( (a, b)):

shuffle ( a : ( list(), list(b)) )

shuffle ( b : ( list(a), list()) )

Define Reduce1 ( list (a : T) )

// Input is a list of key/value pairs where they all have a common key, a node, and the value is a // variable 2-tuple, T, containing lists of follows and followers in that order

let F1 = list() //empty list, will contain follows

let F2 = list() //empty list, will contain followers

for each <node a : 2-tuple T> in list(a: T)

for each node N in first list of T

append N to F1

for each node N in second list of T

append N to F2

return (a : ( Follows, Followers))

// returns a node mapped to a 2-tuple containing two lists that hold the nodes that a follows and the

// nodes that follow a, respectively

Define Map2 ( a : (Follows, Followers))):

if Follows contains @KingJames:

for each node V in Follows:

if V is not @KingJames:

shuffle ( (a, V) : 1 )

else if | Followers | > 107

// If we get here, Follows does NOT have @KingJames, IE ‘a’ does not follow @KJ

for each node U in Followers:

shuffle ( ( U, a) : 1 )

Define Reduce2 ( list ((u, v) : 1) ):

// input is a list of key/values pairs where the common key is a pair of nodes and the value is 1.

let C = 0

for each < (a,b) : 1 > in list ((u, v) : 1)

C += 1

if C == 2

return u

// returns a user that follows @KingJames and also follows one other user with >107 followers

**Problem 3 (cont)**

The first round of MapReduce will have the Mappers, loaded with Map1, reading data from the HDFS. The data read will be of (a,b) form and the Mappers will generate two key/value pairs, mapping each node in the original pair to a 2-Tuple of lists, one empty and the other containing the other node in the original pair. These will be fed to the Reducers loaded with Reduce1 who will generate a mapping from each node to the lists of their follows and followers. This data will be fed into the second round of MapReduce, where the Mappers will be loaded with Map2. This is where some boolean logic will be used to generate logical potential (u,v) pairs where u satisfies the desired conditions. These will be fed to the Reducers loaded with Reduce2 who will determine if the potential (u,v) pairs are bona fide and will return u if it is indeed bona fide. Basically, if the potential (u,v) pair occurs twice (once from each boolean series in Map2), then it will be bona fide. The returned users will then be stored on the HDFS for user access.