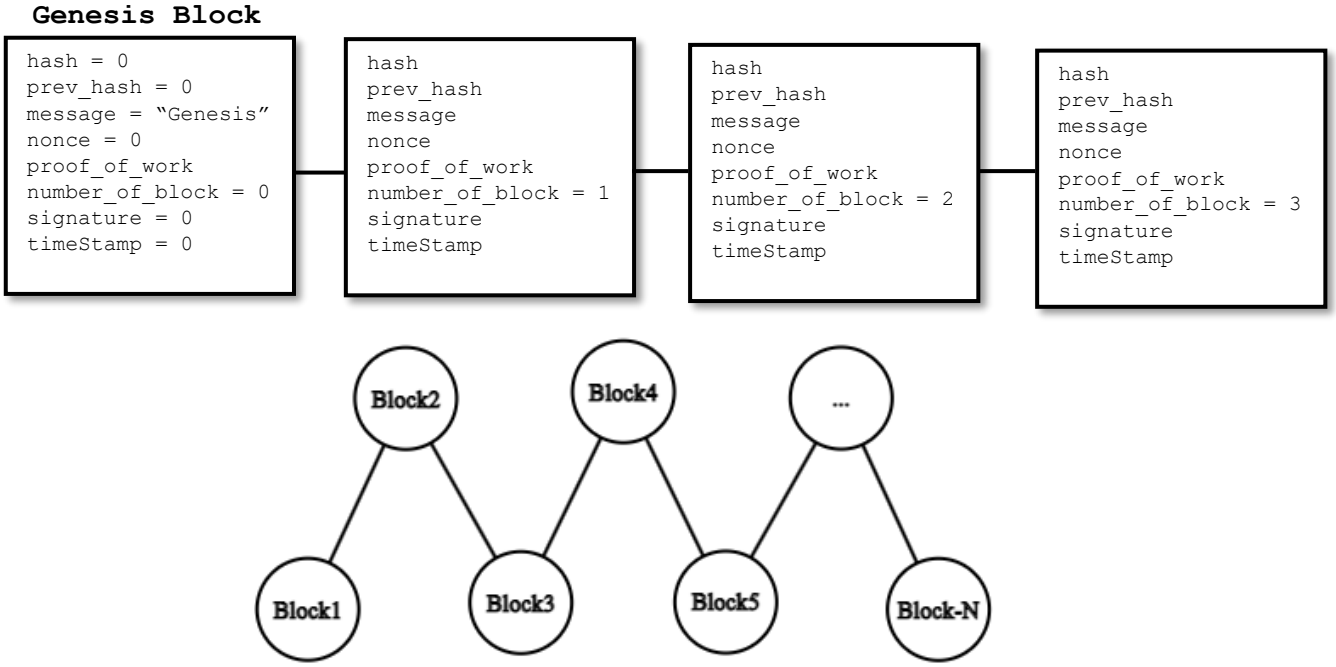
Theoretical Analysis by Abdan Hafidz

How is graph modeling in in Peer-To-Peer Networks on Block-Chain Technology using WebSocket?

1. **Full – Node Block Chain**

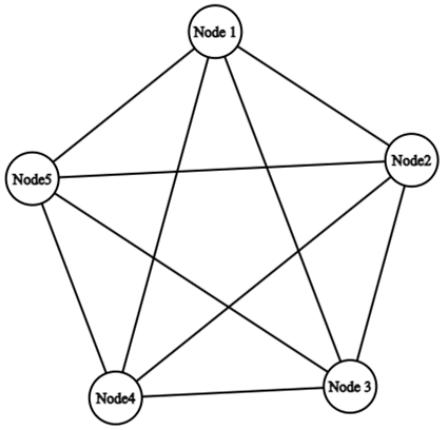
   The block model contains information from the previous block, namely `prev_hash` , so that it can be found that the blocks are connected to each other and the connection between these blocks will form a chain called a block chain (*Block-Chain)* . A block chain that has N blocks can be represented as a *Strong Connected and Directed Graph* that has $N - 1$ edges.



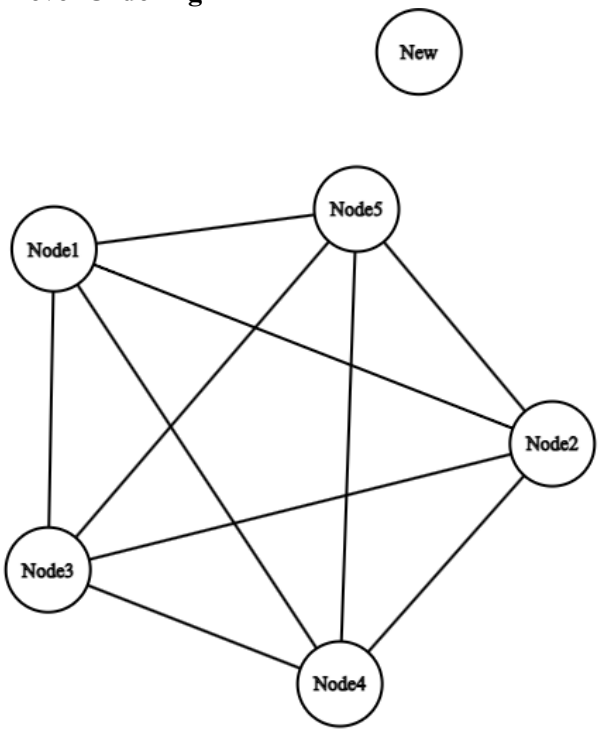
2. **Complete Graph Distributed Peer-To-Peer Network**

   Peer-To-Peer Network is the basis of the decentralization discussed, where with this network method someone is able to communicate with other people without having to go through an intermediary or third party. Someone in the network or we as nodes can be represented as graph nodes.

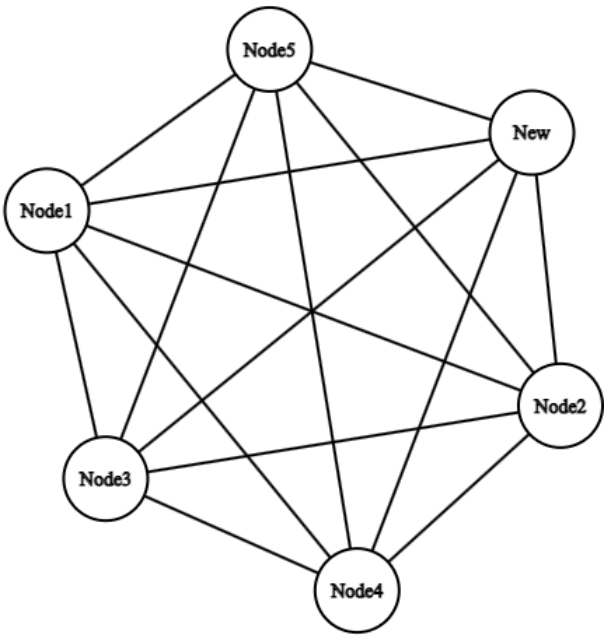
   Everyone in the network is connected to everyone else , so for a network of N people, each person will be connected to N – 1 other people. This results in the formation of a complete graph with $\frac{N(N-1)}{2}$edges. The initiation of a node in the graph or a person joining the network is based on the person's *host-IP address.*



3. **Node Initiation with Level Ordering**



When the node just joined the network, then it will connect itself to all existing nodes, then the node will request synchronization of block-chain data and other node information to the nearest node. The connection process to other blocks is carried out using the level-ordering method, so that it can connect all nodes at the same time without having to take turns, in this case a concurrent programming method with asynchronous functions is required.

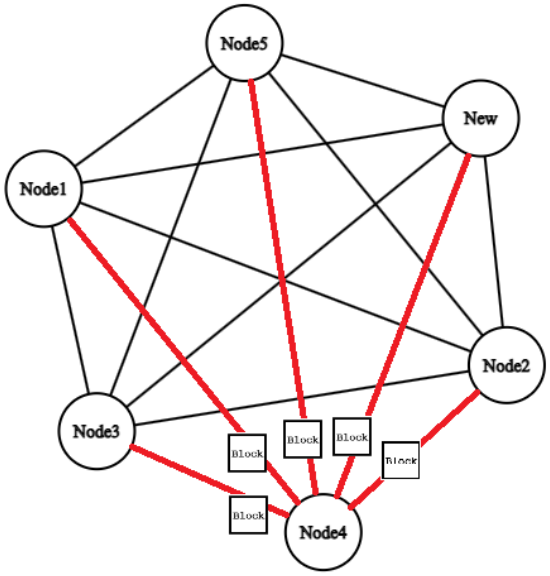


The nearest node calculation is done by determining the weight of the graph edge based on the latency of the connection between two nodes. Node synchronization will be discussed in the next section.

```
async def Peering():
    NetData = DB.GetNetworkData()
    PeerAddresses = list(NetData["Nodes"])
    tasks = [ClientHandler(uri) for uri in PeerAddresses]
    await asyncio.gather(*tasks)
```

4. Broadcasting with Level-Ordering

A node mines a block then broadcasts it to everyone through the network, from here we will understand that the block will be distributed concurrently or all nodes can receive the block at the same time without having to take turns, this is *the transverse ordering level* of a graph.



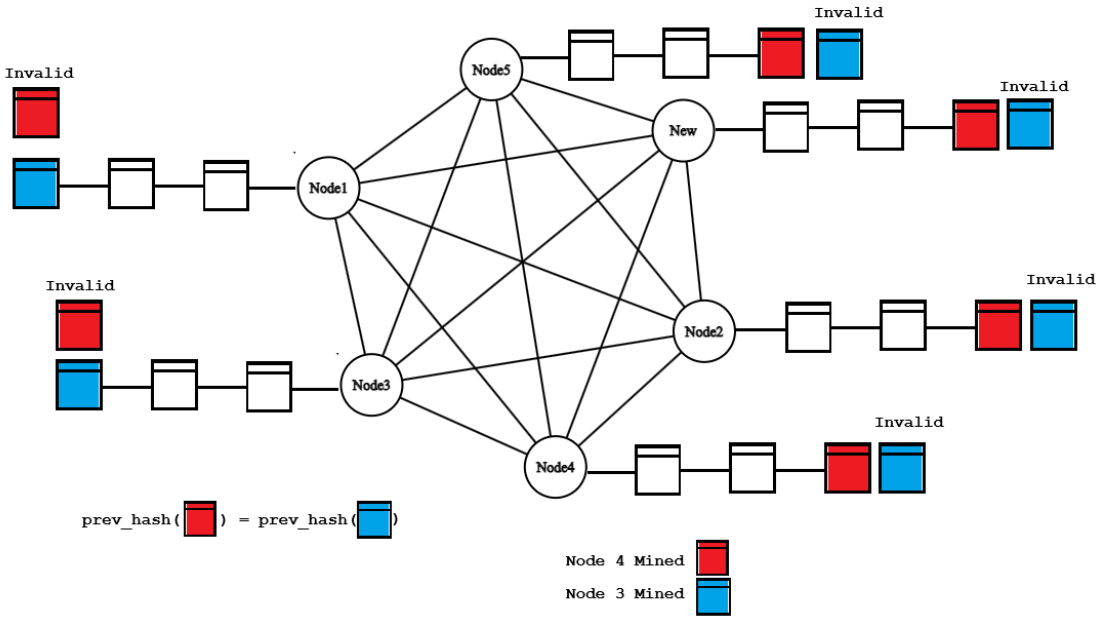

```
Node 4 Mined a block

Node 4 Siarans the Block to (Node
3, Node 1, Node 5, Node2, and
New) concurently
```

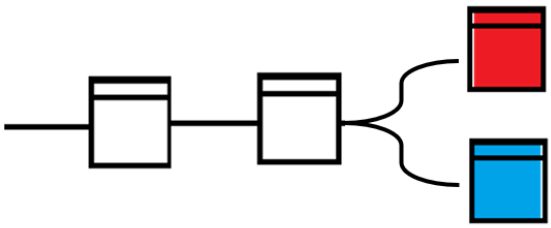
The concept of graph traversal underlies the sending of blocks in a network, the calculation of time can be done by remembering that the further the distance of *the node* , the longer the sending process will be, as well as calculations based on network latency.

5. Bipartite Block Forks

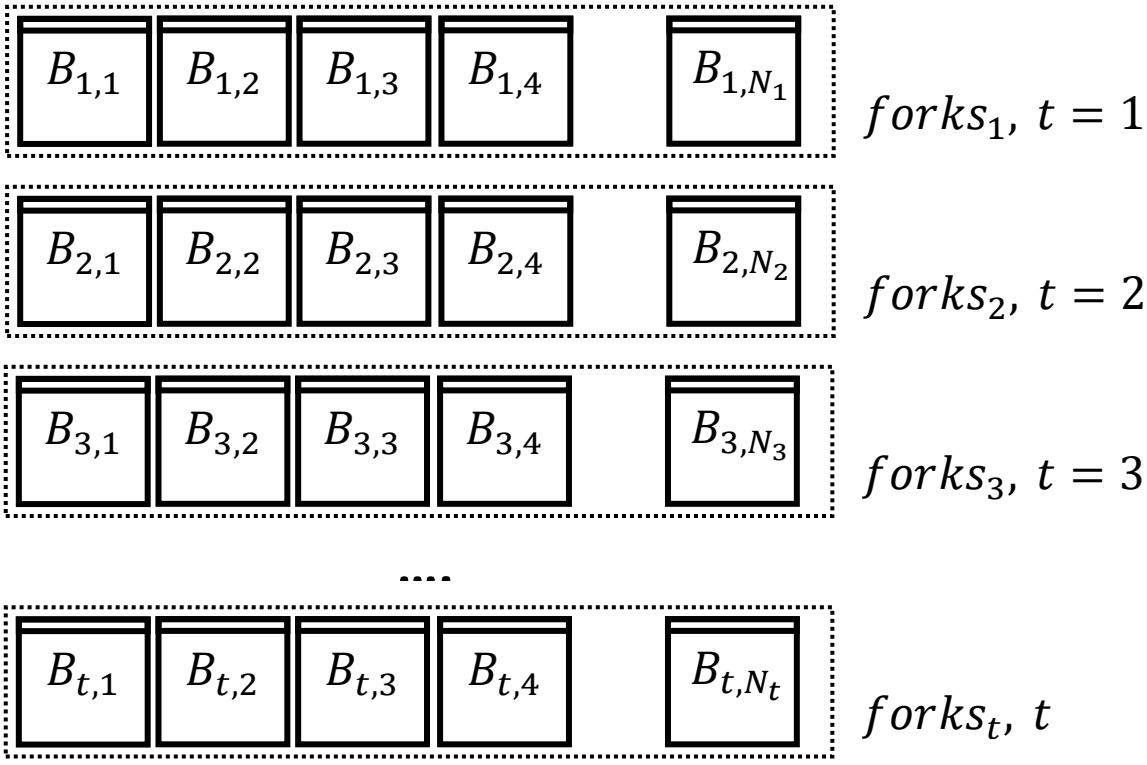
Block Forks is a condition when two or more nodes mine a block at the same time and the resulting block has the same `prev_hash value` , so that many branches will be formed before the block is included in the chain to be verified first. The previous chapter discussed how to solve this problem by sorting blocks based on the largest *proof-of-work* (PoW).



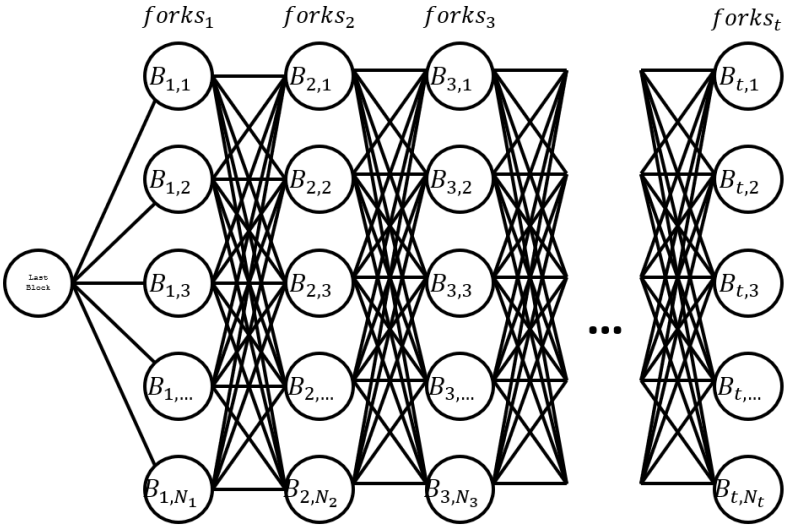
The blocks mined by both nodes will be broadcasted at the same time, then each node receives the blocks in a different order, for example in the illustration above node 4 mines the red block and node 3 mines the blue block. Both nodes simultaneously send blocks at the same time, but other nodes such as node 2, new node, and node 5 receive the red block before the blue block because they are closer to node 4, in contrast to node 3 and node 1 which receive the red block after the blue block joins the chain. Block forking occurs here, where there are two types of chains formed.



Block Forks are multinomial conditions where in any given time period we assume that in a given time span there are N_t blocks received simultaneously at each time t. We can assume F_t to be all blocks that experience forks at time t. The blocks in F_t we assume as $B_{t,i}$ for $1 \leq i \leq N_t$



We can determine that this problem can be modeled into a bipartite graph with the initiate node being the last block in the blockchain before accepting the fork and each node in F_t being connected to all members of F_{t+1} such that node $B_{i,t}$ is connected to $B_{t+1,j}$ for $1 \leq j \leq N_{t+1}$.



Nodes will choose the block with the largest `proof_of_work` on each fork they find and add it to the chain. Block searching can be done using Breadth-First-Search (BFS) transversal.

```
elif VerifyBlock.BlocksFork:
            LastBlock =  BlockChain.BlockChainData[-1]
            LastBlockProof  = LastBlock["proof_of_work"]
            LastBlockCreator =
ParseSender(LastBlock["message"])
            BlockProof = block.proof_of_work
            BlockCreator = ParseSender(block.message)
            if(LastBlockProof > BlockProof):
                if(BlockCreator ==
DB.GetUserData("PublicKey")):
                    MinedBlock =
BlockChain.mine_block(block)

JSONWorker.CreateDataJSON('Buffer/WaitBox.json',{"Draft"
:MinedBlock.__dict__})
                elif(BlockProof > LastBlockProof):
                    BlockChain.remove_block()
                    BlockChain.add_block(block)
                    if(LastBlockCreator ==
DB.GetUserData("PublicKey")):
                        MinedBlock =
BlockChain.mine_block(LastBlock)

JSONWorker.CreateDataJSON('Buffer/WaitBox.json',{"Draft"
:MinedBlock.__dict__})
```

6. Node Synchronization

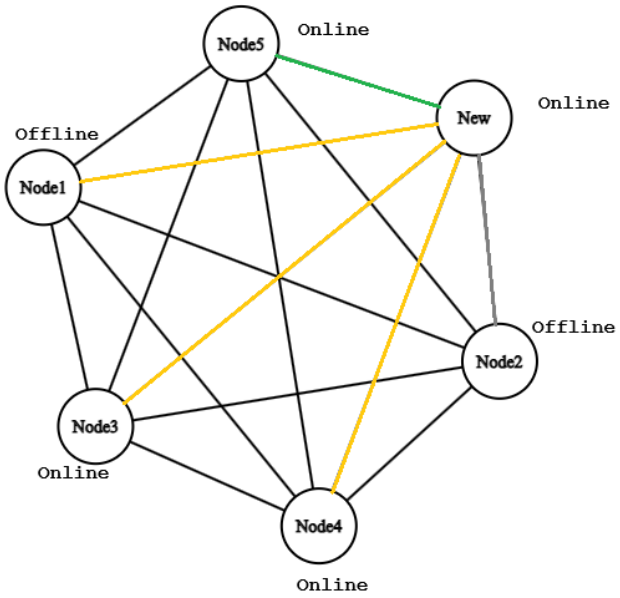
Nodes can catch up on the block lag or chain asynchronous by downloading the block chain from other nodes, then the node that gets the failed chain verification will also update the local block chain data from the download. The selection of nodes to be synchronized is based on a technique that applies the concept of neighbors in the graph.

The node will choose another node that is closest to the *Nearest Neighbor algorithm* in the connection based on the measurement of the connection latency. The selection of this approach considers the optimality of the algorithm for the *Complete Graph type of graph* . The node must also ensure that the selected neighbor is online, find another node with minimum latency besides that.

Suppose we obtain information on the network that:

```
latency(New,2) < latency(New,5)< latency(New,1) < latency(New,4) < Latency(New,3)
```

We will find that *the nearest neighbor* of the new node is node 2, but because node 2 is offline, the new node will switch to the next closest node, node 5, to perform synchronization.

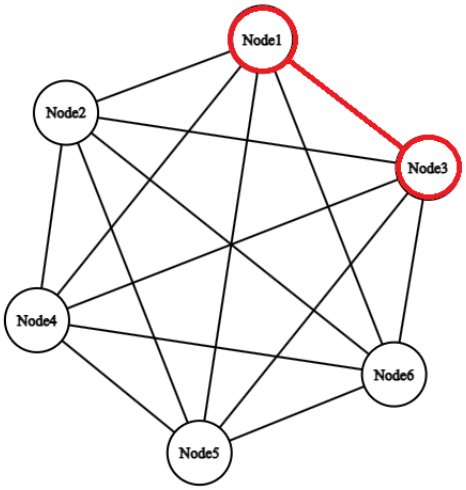


However, in terms of synchronization, several vulnerabilities were found which will be discussed in the seventh section.

7. Byzantine Fault Tolerance (BFT)

Synchronization performed by a node with another node has the potential to experience vulnerabilities such as other nodes that are asked to synchronize trying to send invalid block-chain data. This problem can be solved with one implementation that allows three solution paradigms at once.

The illustration below will show, for example, node 1 receives an invalid block-chain from node 3, so that node 1's local block-chain becomes invalid.

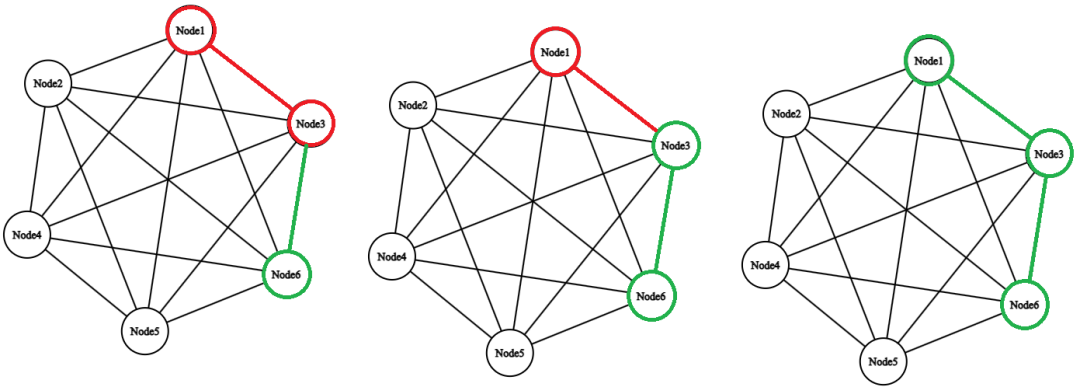


7.1. Naïve Nearest Neighbor (unmutual neighbor)

Problems can be solved by naively hoping that is when node 3 receives a block broadcast, mines a new block, or goes offline then comes back online and gets synchronization from *the nearest* trusted neighbor with the assumption of *unmutual neighbors* :

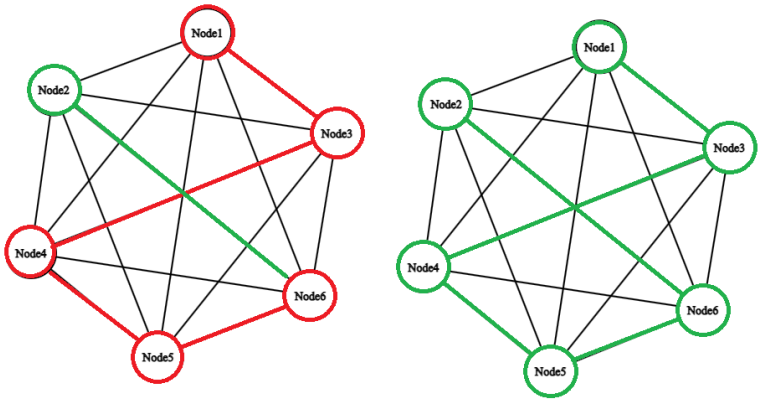
$$nearest_{neighbour(node\ u)} = node\ v$$
$$nearest_{neighbour(node\ v)} \neq node\ u$$

We can also construct Node 3 to verify its blockchain data first and then synchronize it before sending it to node 1, so that from here node 3 will get a valid block chain, so that when node 1 requests synchronization from node 3 it will also receive a valid block chain.




```
async def HandlerGetBlockChain(websocket, path):
        BlockChainSync.VerifySyncBlockChain()
        BlockChainData = DB.GetBlockChainData()
        UserData = DB.GetUserData("PublicKey")
        await websocket.send(json.dumps({
            "WalletAuthor":UserData,
            "BlockChainData" : BlockChainData}))
```

A larger case such as all nodes provide invalid synchronization, but at least one node provides a valid block-chain, then graph traversal will be performed.

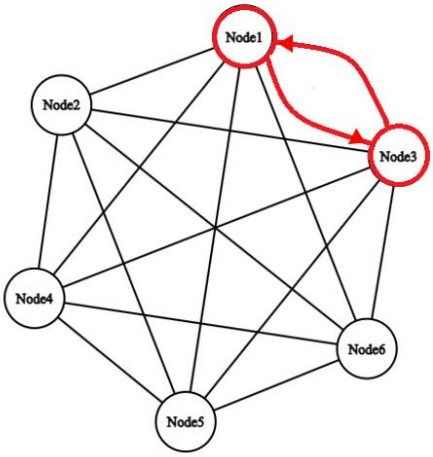


7.2. Breadth First Search with Backtracking

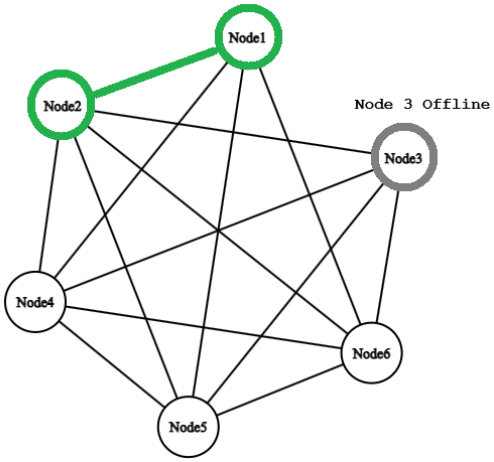
The solution using *Nearest Neighbor* is a naive solution because it has the potential for *mutual neighbor cases* where there are cases:

$$nearest_{neighbour(node\ u)} = node\ v$$
$$nearest_{neighbour(node\ v)} = node\ u$$

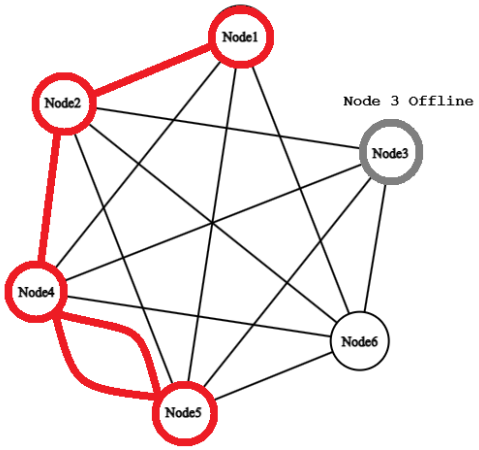
If this happens then the graph exploration will result in *an infinite loop of* only two mutual nodes.

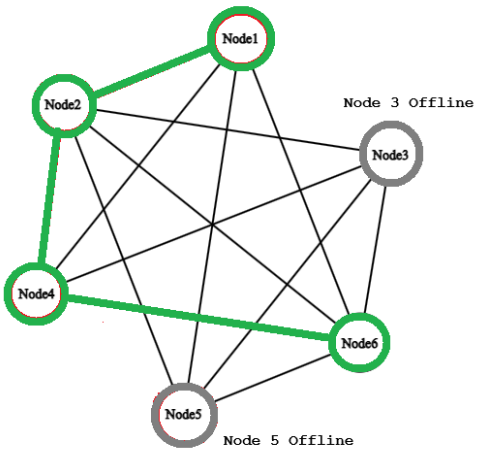


Another naive solution to this problem is that we hope that one of the nodes *is offline* so that the node will *backtrack* and then request synchronization to another valid *nearest neighbor* .

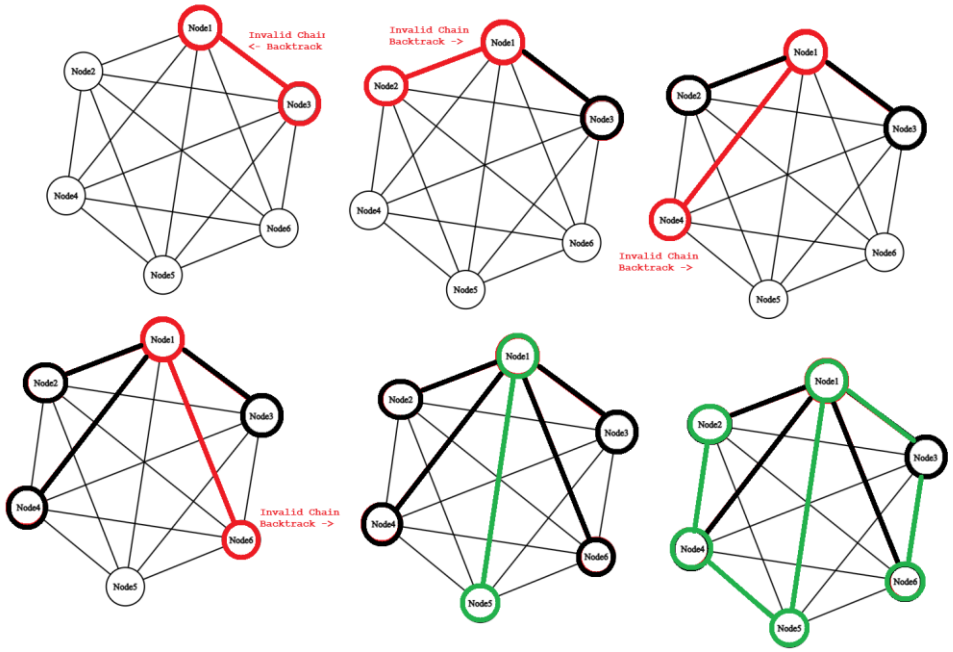


Also applies to larger cases.





We consider another case, although the probability is very small, namely where when a loop occurs, *the mutual node* will never go offline. The application of BFS Backtracking for this problem is when synchronization is performed, do not forget to verify the received blockchain, if the synchronization is invalid then *backtrack* and request synchronization to another node that has minimum latency.




```
def SynchronizeBlockChain():
    BlockChain = Blockchain()
    data  = DB.GetNetworkData()
    nodes = data["Nodes"]
    latency_nodes = {k: v for k, v in nodes.items() if
"Latency" in v}
    sorted_nodes = sorted(latency_nodes.items(), key=lambda
item: item[1]["Latency"])
    nodeIndex = 0
    while True:
        if latency_nodes:
            closest_node = sorted_nodes[nodeIndex]
            closest_ip = closest_node[0]
            ReceivedChain =
WSBlockChainSync.SynchronizeHandler(closest_ip)
            if(ReceivedChain["BlockChainData"] != '' and
ReceivedChain):

if(BlockChain.verify_chain(ReceivedChain["BlockChainData"])):

JSONWorker.CreateDataJSON('Database/BlockChainDB.json',Receiv
edChain["BlockChainData"])
```