Assignment 3

Question 1

Algorithm

Algorithm **bfsTrees** (id)

```
In:  Processor id
Out:  Two BFS trees, one rooted at processor 1, one rooted at processor 2

mssg1, mssg2, message, ack1, ack2, parent1, parent2 <- null
Initialize children1 and children2 vector as well as a string array for data

if (processor id = 1)
     mssg1 <- send message to adjacent1 with id, REQ and root identifier
     parent1 <- null
else if (processor id = 2)
     mssg2 <- send message to adjacent2 with id, REQ and root identifier
     parent2 <- null
else
     mssg1, mss2 <- null
     parent1, parent2 <- null
ack1, ack2 <- null
int roundsLeft <- -1
Loop:
if (mssg1 != null) then send mssg1, roundsLeft <- 1
if (mssg2 != null) then send mssg2, roundsLeft <- 1
if (ack1 != null) then send ack1
if (ack2 != null) then send ack2
message <- receive message
while (message != null)
     data <- unpack data
     if (is REQ)
         if (from root 1)
             if (parent1 = null)
                 parent1 <- data[0]
                 remove parent1 from adjacent1
                 mssg1 <- send message to adjacent1 with id, REQ and root identifier
                 ack1 <- send message to parent with id, ACK and root identifier
             else
                 remove data[0] from adjacent1
         if (from root 2)
             if (parent2 = null)
                 parent2 <- data[0]
```

```
                remove parent2 from adjacent1
                mssg2 <- send message to adjacent2 with id, REQ and root identifier
                ack2 <- send message to parent with id, ACK and root identifier
            else
                remove data[0] from adjacent2
    else if (is ACK)
        if (from root 1) then add data[0] to children1
        if (from root 2) then add data[0] to children2
    message <- receive message
if (parent1 != null AND parent2 != null)
    if (roundsLeft = 0)
        printParentsChildren(parent1, parent2, children1, children2)
        return ""
    else if (roundsLeft = 1) then roundsLeft = 0
return null
```

Java implementation

See DBFS.java

Proof of termination

Once a processor sets its parent1 and parent2, it will terminate. The processors near the middle of the two BFS tree will begin to return first since they will know their parent1 and parent2 earliest. As the BFS tree continues, the rest of the processors will know its parent1 and parent2, therefore terminates. Once all processors terminates, algorithm ends.

Proof of correctness

A processor will choose as parent the id in the first message that it receives. In round 1, messages from both root were sent and received by all processors at distance 1 from the root. When a processor receives the message, it will forward to all its neighbours in the next round (except its parent), and send ACK back to its parent. Therefore, in round 2, messages were went to and received by all processors at distance 2 from the root. So on and so on, at the nth round, the message will arrive at processors at distance n from the root and these are the leaf of the BFS tree, aka the farthest processors, for both trees. In order to properly identify the trees and store the correct value, there are tree identifiers and ACKs for two trees in place. Compare to the normal BFS tree, the processor do not terminate unless both parent1 and parent2 are set.

Wenxi Chen
wchen466

Assignment 3

CS 4438
November 4, 2021

Time complexity and communication complexity

Time complexity
Since for every single rounds, the message travels for distance of 1. In order to reach to the leafs, in the worst scenario, the message needs to travel the distance of the tree, or the diameter of the tree. Therefore the number of rounds it will take to reach the leafs will equal to the diameter. Assume the diameter is n:

$f(n) = numOfRounds \leq diameter = O(n)$

Communication complexity
In this algorithm, each communication link will have 2 messages travel on them. They are REQ and ACK. For example, when parent send REQ to children, children need to send the ACK back to the parent in order to confirm the adoption. Therefore, the communication complexity will depends on how many communication links are present. Assume there are n links:

$f(n) = numOfMessages = 2 * numOfLinks = O(n)$

Will it work on an asynchronous system?

No, not really. Let's say in an asynchronous system and you have two (or more) nodes linked to their adjacent nodes with varying distance (1, 2, or even longer). If the communication is slow, theoretically the adjacent node is not guaranteed to receive from the closest node, therefore, it will not be getting the shortest path. This algorithm need to wait till it has heard from all the adjacent nodes and decide which one is the closest.

Question 2

Algorithm

Algorithm **computeRoutingTable** (id)

```
In:  Processor id
Out:  Routing table for each processor

table <- create new table
mssg, message <- null
count <- 0
if (is leaf)
     mssg <- send message to parent with string formed id, table
else mssg <- null
Loop:
if (mssg != null) then send mssg
mssg <- null
if (count = number of children)
     if (is root) then print table
     else mssg <- send message to parent with string formed id, table
     return ""
message <- receive message
while (message != null)
     count ++
     add to table with the same hop and destination
     if (we receive empty table AND not root)
        mssg <- send message to parent with string formed id, table
     else
         use every single table entry received to populate the current table
         if (not root)
             mssg <- send message to parent with string formed id, table
     message <- receive message
return 0
```

Java implementation

See ComputeRouting.java

Assignment 3

Proof of termination

For leaf nodes, since it doesn't need to process any incoming routing table, they can simply send a empty table up to their parents and then terminate. For any other processors including the root, it will and only will terminate once the number of message they receive equals to the number of children they have. In the test case, the root node will not terminate, even though there is a leaf node passing a empty table to it. Therefore, all processors will eventually terminate using this algorithm. When root terminates, the entire algorithm is terminated.

Proof of correctness

In this algorithm, the correct output is where the processors outputs correct routing tables. For each leaf node, they send an empty table to their parent. Once their parent, or any internal node receives an table (empty or not) they will first create an entry on their own table, with the next hop and destination from the source node (could be a leaf, could be another internal node). Once they done that, they will iterate through the table received (if it is not empty) and add next hops and destinations to its own routing table. After they have done that, and confirmed that it received a table for each of its children, it will send it up to its parent (unless it is root). If we are at the root, the algorithm will output the final, correct routing table and terminates.

Time complexity and communication complexity

Time complexity

Similar to convergecast, each leaf node will send the table to its parents with distance of 1. Then, its parents, and other internal nodes (except the root), will also send the table to their parent travelling a distance of 1. There is one extra round at root, where it prints out the final table and terminates. Therefore, the time complexity of this algorithm is diameter + 1. Assume the diameter of the network is n:

$f(n) = diameter + 1 = n + 1 = O(n)$

Communication complexity

Each processor except the root will send a message to its parent in this algorithm. Each processor will have only one parent to send message to, and they terminates right after sending the message. Therefore, the communication complexity will be number of processors - 1. Assume the number of processor is n:

$f(n) = numOfProcessor - 1 = n - 1 = O(n)$

Will it work on an asynchronous system?

Absolutely. Since the algorithm compares the number of children of a parent with the number of messages received by it, we can ensure the parent (assumed not the root) will not send the table to its parent **prematurely**. Even if communication is slow, as long it gets to its parent, the counter will increment by 1. The counter will not increment unless you reach the parent, doesn't matter how slow you are. Therefore, this algorithm will work on an asynchronous system.