SS 2004 Prof. R. Wattenhofer / Fabian Kuhn / Regina O'Dell

Principles of Distributed Computing
Exercise 1: Sample Solution

1 Vertex Coloring

a) In the lecture, we have seen that Algorithm 1.9 ("Reduce") of the lecture notes needs M rounds to complete when started with a valid initial coloring of colors between 1 and M. If the initial colors are unique node IDs, this is $O(n)$ if all IDs are in $O(n)$. However, if we for example assume that IDs are arbitrary $O(\log n)$-bit numbers, the number of possible IDs can be any polynomial in n and the time complexity of Algorithm 1.9 is not linear any more.

Algorithm 1.9 works because it guarantees that no two neighbors in the graph G assign a new color simultaneously. If we are able to design an algorithm for which this condition still holds but which assigns a new color in every communication round, we are done. Algorithm 1 which is synchronously executed by all nodes fulfills these requirements.

Algorithm 1 "$\Delta + 1$"-Coloring in $O(n)$ Rounds

```
send node ID to all neighbors.
while no color assigned do
  if ID is lowest among all un-colored neighbors then
    choose smallest possible color
    send chosen color to all neighbors
  end if
end while
```

In each (but the first) round at least the un-colored node with the lowest ID in the graph assigns a color. Therefore, the algorithm terminates after at most $n + 1$ rounds.

b) Each node sends exactly two messages to each neighbor, one in the first round and one after assigning a color. Therefore, the total number of messages is $4 \cdot m$ where m denotes the number of edges in the graph.

c) Yes, the algorithm still works, it could be reformulated in the following way (we assume that each node knows its degree):

Algorithm 2 Asynchronous "$\Delta + 1$"-Coloring

```
send node ID to all neighbors.
wait until all neighbor IDs have been received and all neighbors with a lower ID have chosen
  a color
choose smallest possible color
send chosen color to all neighbors
```

Algorithm 3 Counting Nodes I

wait until receiving a request to count the nodes of sub-tree (originator of this request is the parent node)
if I am a leaf **then**
    send 1 back to the parent node
else
    send request for counting to all children
    wait until all children have sent the sizes of their sub-trees
    send 1 + sum of the sizes of the children sub-trees to the parent node
end if

2 Counting the Nodes of a Tree

a) For convenience, we define v as the root of tree T. v starts the algorithm by asking all of its children about the sizes of their sub-trees. Each node then performs the above Algorithm 3.

The "request" messages have to travel all the way down to the leafs of the tree and after arriving there, the "result"-messages travel all the way up to the root v of the tree. The time complexity of this algorithm is therefore $2 \cdot h$ where h is the height of the tree. This holds for the synchronous and for the asynchronous variant of the algorithm.

b) Essentially, we can simultaneously execute the second phase of the above algorithm for all possible root nodes. The algorithm can be formulated as follows:

Algorithm 4 Counting Nodes II

if I am a leaf **then**
    send 1 back to the parent node (the only neighbor)
else
    wait until all but one neighbors have sent the sizes of their sub-trees.
    send 1 + sum of the sizes of the sub-trees to the neighbor u which has not yet sent the size of its sub-tree.
    wait until the last neighbor u has sent the size of its sub-tree
    for all neighbors w except u **do**
        send 1 + sum of the sizes of the sub-trees of the other neighbors to w
    end for
end if
Calculate the number of nodes as 1 + the sum of all received sub-tree sizes

Each node sends exactly one message to each neighbor, the message complexity is therefore $2(n-1)$ ($n-1$ is the number of edges of a tree). The time complexity is $O(diameter(G))$.

c) First, we prove that no neighbor of v can have a sub-tree whose size is greater than $n/2$ (note that having size exactly $n/2$ is not possible because we defined n to be odd). For the sake of contradiction, assume that v has a neighbor w whose sub-tree has a size $s_w > n/2$. When dividing T at v, we get a $s_w : (n - s_w - 1)$-partition. When dividing T at w, we can get a $(s_w - 1) : (n - s_w)$-partition which is better.

Second, we prove that there is a unique node v for which all neighboring sub-trees are smaller than $n/2$. Such a node v exists because all other nodes have a neighbor which achieves a better partition of the tree. There must be at least one optimal node. Further v is unique because for all neighbors w of v, the sub-tree rooted at v has a size which is greater than $n/2$.

The worst that can happen is that v has three equal neighbors. For the partition, we then get a $1 : 2$ ratio.