Principles of Distributed Computing
Sample Solution to Exercise 1

1 Vertex Coloring

If the nodes omit the "undecided" messages, then each node sends exactly two messages to each neighbor, one in the first round and one after assigning a color.

a) To express the worst-case message complexity in terms of n, we have to think of a topology that maximizes the number of neighbors of each node. Such a topology is a clique, where each neighbor has $n-1$ neighbors. Then, as each node sends 2 messages to each of its neighbors, the message complexity is $2n(n-1)$ in the worst case.

b) There are 4 messages sent over each edge: two in the first round, two after assigning a color. Hence, the total number of messages is $4m$.

2 TDMA

a) The resulting coloring is depicted in Figure 1. Note that the clique of size 3 needs at least 3 colors. Hence, our solution is optimal.

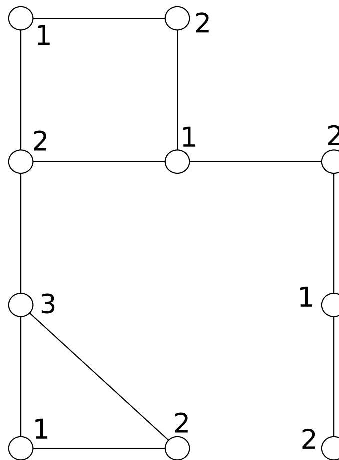

Figure 1: The slots for the wireless network

b) We will use additional edges to model the new interferences. Two neighbors of a node are not able to send messages at the same time if they have different colors. Therefore, for every node, we will add an edge between every two neighbors of that node.

The resulting coloring is depicted in Figure 2. There are multiple cliques of size 4 in the new graph; any of these needs at least 4 colors. Hence, our solution is optimal.

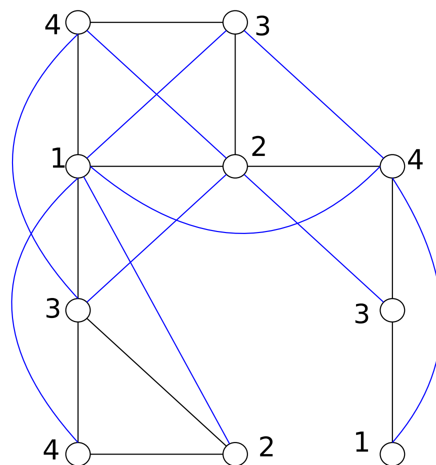

Figure 2: The slots for the wireless network with improved communication. The additional constraints are shown in blue.

c) Every pair of lectures that is selected by a student cannot take place at the same time. Thus, they cannot be colored with the same color. This leads us to the solution shown in Figure 3. The graph can be colored with 3 colors.

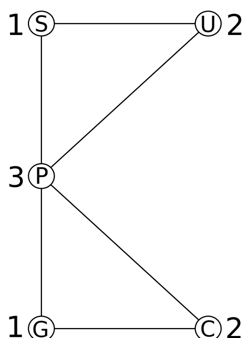

Figure 3: The resulting graph with P being shorthand for Principles of Distributed Computing, S for Statistical Learning Theory, U for Ubiquitous Computing, G for Graph Theory, and C for Cryptography. Note that 3 colors are sufficient.

3 Coloring Trees

a) The log-star algorithm for the ring is basically identical to the algorithm for trees. Nodes do not have a parent in the ring, therefore we simply define the left neighbor of any node to be its "parent". Given this definition, we can run the normal log-star algorithm. Using the same argument as for trees, it can be shown that no two neighboring nodes choose the same color. Note that we can omit the "shift" step, as all "children" (i.e., the right neighbor) always have the same color.

b) We build the algorithm step by step. The shape of the algorithm is similar to the solution of the previous task, consisting of a 6-*color* phase and a *Reduction* phase, going from 6 colors to three colors.

What happens when the nodes do not know n? Firstly, let's note the main challenge arising when n is unknown. In the log-star algorithm, the nodes precompute the number of iterations sufficient to 6-color the graph. Computing this sufficient number of iterations ($O(log^{\star}n)$) relies on the knowledge of an upper bound of n (if not n itself). Hence, the natural question is *When can nodes stop the* 6*-colors phase and start the Reduction phase?*. Note that, in order to preserve the sublinear round complexity of the log-star algorithm, nodes must make this decision without communicating with everybody else.

When can a node stop the 6-*color phase?* We can start reasoning from the following claim: *A node can stop computing new colors in the* 6*-colors phase when obtaining a color in* $\mathcal{R}$*.* Is this claim correct?

Well, *almost*. Let u and v denote two nodes and assume that u is v's left neighbor. Consider the following examples:

- Node u obtains a color, say, $2 \in \mathcal{R}$ (10 in binary), while node v's color is 16 (10000 in binary). Then, node v compute its new color in the next round as $2 \cdot 1 + 0 = 2$, which is the same color as u's, who does not change its color anymore!
- Node v obtains a color, say $1 \in \mathcal{R}$, while node u's color is 17 (10001 in binary). If the color of node u's left neighbor is 16 (10000 in binary). Then, node u computes it's new color as $2 \cdot 0 + 1 = 1$, which is the same color as v's, who doesn't change its color anymore!

So, we can overcome this issue with the help of two additional colors, ℓ and r! If node v obtains a color in $\mathcal{R}$ in some round, it stops the 6-colors phase. Then, node v will check its neighbors: if its left neighbor has obtained a color in $\mathcal{R}$, then v sets its color to ℓ, meaning "My left neighbor is done, therefore I stop this phase to avoid conflicts". Similarly, if its right neighbor has obtained a color in $\mathcal{R}$, then v sets its color to r, with the meaning "My right neighbor is done, therefore I stop this phase to avoid conflicts".

Hence, *when a node v gets a color in $\mathcal{R} \cup \{\ell, r\}$, it will stop the* 6*-color phase.*

There is still one detail that needs to be addressed in this phase: the additional colors ℓ and r essentially segment the ring into rooted trees (more precisely, directed paths). Node v computes its color based on its left neighbor's color. If this is ℓ, then v acts as a root: it computes its new color based on an arbitrary color.

We present the formal code of the updated 6-colors phase below. For simplicity, each node will use the subroutine presented in Algorithm 1 for computing a new color (lines 5–8 of the 6-color Algorithm in the lecture notes).

Algorithm 1 `new_color`(c_ℓ, c_v)

1: interpret c_ℓ and c_v as bit-strings
2: let i be the index of the rightmost bit b where c_v and c_ℓ differ
3: return $c_v = 2i + b$

We now present the updated "$(6, \ell, r)$-Coloring" phase.

Algorithm 2 $(6, \ell, r)$-Coloring Phase

1: send c_v to both neighbors
2: **while** $c_v \notin \mathcal{R} \cup \{\ell, r\}$ **do**
3:   **if** $c_\ell \in \mathcal{R}$ **then**
4:     $c_v := \ell$;
5:   **else if** $c_r \in \mathcal{R}$ **then**
6:     $c_v := r$;
7:   **else if** $c_\ell = \ell$ **then**
8:     $c_v =$ `new_color`(arbitrary color, c_v) ("root")
9:   **else**
10:     $c_v =$ `new_color`(c_ℓ, c_v)
11:   send c_v to both neighbors

The round complexity of this algorithm is $O(\log^* n)$. We need to show that the coloring is proper, i.e. two neighbors cannot obtain the same color c. Assuming that this is the case, the case when $c \in \mathcal{R}$ is covered by the analysis of the log-star algorithm from the lecture. The cases when $c = \ell$ and $c = r$ contradict the description of the algorithm.

Starting the reduction phase. Let us continue with the following claim: *Once v obtains a color in $\mathcal{R} \cup \{ell, r\}$, it can immediately start the Reduction phase.* Is this claim true?

Well, *no*. Let us say that a node ends up with color ℓ, and enters the Reduction phase immediately. This node may end up setting its color to, say, 2, while its right neighbor, still acting as a root in the 6-Colors phase, also sets its color to 2 in the same round.

To avoid this challenge, when exiting the 6-Colors phase, each node waits until its neighbors obtain colors in $R \cup \{\ell, r\}$ as well.

From $6 + 2$ to 3 colors. At this point, the nodes can reduce the colors from $\mathcal{R} \cup \{\ell, r\}$ to $\{0, 1, 2\}$. Algorithm 3 presents outlines the *Reduction phase*. Note that since every node has two neighbors, there is always a color available in $\{0, 1, 2\}$.

Algorithm 3 Reduction Phase

```
1: wait until both neighbors' colors are in R ∪ {ℓ, r}
2: while c_v ∉ {0, 1, 2} do
3:   if my_turn() then
4:     choose smallest available color c_v ∈ {0, 1, 2}
5:     send c_v to both neighbors
```

There is still a question left: *When is it* `my_turn()` *to pick a color in* $\{0, 1, 2\}$*?* Since adjacent nodes may start reducing at different times, the approach of the lecture's log-star algorithm may fail. That is, if we simply iterate through the colors (e.g. nodes with color ℓ go first, then nodes with color r, afterwards nodes with color 5 etc.), a node and its neighbor, although they end up with different colors in the 6-colors phase, might choose a color in $\{0, 1, 2\}$ at the same time. They might even choose the same color! To avoid this, we will use the `round_number`: since the nodes operate synchronously, they have the same `round_number` at any time, even if they are in different phases of the algorithm. We will define the subroutine `my_turn()` as follows.

Algorithm 4 `my_turn`(`round_number`, c_v)

```
1: x := (round_number mod 5) + 3
2: if (c_v = x) then
3:   return true
4: else if x = 6 and c_v = l then
5:   return true
6: else if x = 7 and c_v = r then
7:   return true
8: else
9:   return false
```

Putting it all together. Algorithm 5, presented below, is the final solution.

Algorithm 5 Synchronous 3-Coloring on Ring

```
1: send c_v to both neighbors
2: while c_v ∉ R ∪ {ℓ, r} do
3:    if c_ℓ ∈ R then
4:       c_v := ℓ;
5:    else if c_r ∈ R then
6:       c_v := r;
7:    else if c_ℓ = ℓ then
8:       c_v = new_color(arbitrary color, c_v) (root)
9:    else
10:      c_v = new_color(c_ℓ, c_v)
11:   send c_v to both neighbors
12: wait until both neighbors' colors are in R ∪ {ℓ, r}
13: while c_v ∉ {0, 1, 2} do
14:   if my_turn(round_number, c_v) then
15:      choose smallest available color c_v ∈ {0, 1, 2}
16:      send c_v to both neighbors
```

`new_color`(c_ℓ, c_v)

```
1: interpret c_ℓ and c_v as bit-strings
2: let i be the index of the rightmost bit b where c_v and c_ℓ differ
3: return c_v = 2i + b
```

`my_turn`(`round_number`, c_v)

```
1: x := (round_number mod 5) + 3;
2: return (c_v = x) or (x = 6 and c_v = l) or (x = 7 and c_v = r)
```