Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2024 Prof. R. Wattenhofer

Principles of Distributed Computing
Sample Solution to Exercise 9

1 Self-Stabilizing $(\Delta + 1)$-Coloring

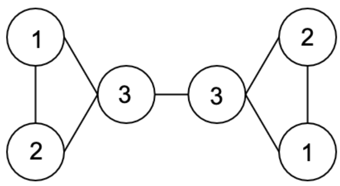
a) Every node u holds an integer c_u between 1 and $\Delta + 1$ such that $\forall v \in N(u) : c_v \neq c_u$.

b) Yes. Let u and v denote the left node and resp. the right node having initial color 3. Node u updates its color to 2, while node v can only update its color to 4.

Fun fact. Self-stabilizing algorithms are designed under different types of *schedulers*, commonly known in the literature as *daemons*. When a node wants to make a change in its state (i.e. if it needs to change its color), it gets *enabled*. The daemon may allow all enabled nodes to make a change at all times, or it may only allow a subset (so that every active node makes a step eventually). A *distributed daemon* may hence not allow both nodes u and v to make at the same time: it could be that node v is allowed to make a step first and hence it updates its color to 4, and afterwards node u maintains color 3.

Our particular input configuration leads to a stable configuration regardless of the type of daemon: either u or v are eventually allowed to make a change, or they are allowed to do so simultaneously. In both cases, the result is a proper coloring.

c) No (unless we assume a *central daemon*), the algorithm is not a self-stabilizing $(\Delta + 1)$-coloring algorithm. As a counterexample, we consider the graph and initial configuration below. We assume no transient faults occur from this point on.



The nodes having initial colors 3 may simultaneously switch to color 4. In the following round, they may simultaneously switch to color 3 again, and afterwards they may simultaneously switch to color 4, and so on. A legitimate configuration is never reached.

To fix this issue, we need to prevent neighboring nodes from updating their color at the same time. A *central daemon* prevents this by default as it only selects one enabled node to make a move at a time. In the general case, we could assume hardcoded IDs (that cannot be corrupted) and give priority to the node with a higher ID. Alternatively, we could use randomness.

2 Self-stabilizing Spanning Tree

a) It will be sufficient to prove this lower bound assuming that nodes hold hardcoded IDs, and that no transient faults occur. We assume that there is a deterministic algorithm $\mathcal{A}$ that correctly defines a spanning tree in $o(D)$ rounds in this setting. Hence, there is a D_0 such that, for any graph of diameter $D \geq D_0$, $\mathcal{A}$ takes at most $k \leq \lfloor D/2 \rfloor - 1$ rounds.

To reach a contradiction, we first run $\mathcal{A}$ on a cycle $C_{2D} := (v_0, v_1, \dots v_{2D-1})$ of $2D \geq 2D_0$ nodes, with root v_0. After k rounds, the nodes variables' p_{v_i} must define a spanning tree. This will be a path containing all edges in C_{2D} except for one edge $(v_i, v_{i+1 \mod 2D})$.

Our goal is now to define a path P containing all edges of C_{2D} except for one edge $e = (v_j, v_{j+1 \mod 2D})$ so that nodes v_i and $v_{i+1 \mod 2D}$ cannot distinguish between C_{2D} and P. Hence, we need to choose an edge e that is outside the k-neighborhoods of v_i and v_{i+1}. Note that these two k-neighborhood form a path of at most $2k+1 < D$ edges out of the $2D$ edges of the cycle, hence such an edge e exists.

Hence, when running $\mathcal{A}$ on such a path $P = (v_j, v_{j+1 \mod 2D}, \dots, v_{j-1 \mod 2D})$, nodes v_i and $v_{i+1 \mod 2D}$) behave identically to our run on C_{2D} within the first k rounds: they end up with the same variables p_{v_i} and $p_{v_{i+1}}$.Therefore the edge $(v_i, v_{i+1 \mod 2D})$ is not marked as part of the spanning tree. However, as the edge $e = (v_j, v_{j+1 \mod 2D})$ is now missing, $\mathcal{A}$ does not obtain a spanning tree within k rounds, hence we obtained a contradiction.

b) The Bellman-Ford algorithm terminates in $R(r) \in \Theta(D)$ rounds, where $R(r)$ denotes the radius of the graph at the root r. This implies that its self-stabilizing variant needs exactly the same number of rounds to stabilize, matching the bound from **a)**. Since the transformation of an algorithm running in k rounds results in an algorithm simulating k instances of the original algorithm in parallel, we need to transmit $R(r) \in \Theta(D)$ times more information in each round.

3 Crash Failures

a) In every round, nodes send 'awake' messages to their neighbors. The left-most node v starts the coloring by choosing its own color $c_v := 0$, and sends c_v to its right neighbor. When a node v receives color c_u from its left neighbor u, it sets its color to $c_v := 1 - c_u$ and sends c_v to its right neighbor. This way, we are guaranteed that non-crashing adjacent nodes obtain different colors.

If, in some round, node v does not receive a message from its left neighbor u, then node u has necessarily crashed. Now v may consider itself the left-most neighbor, set its color $c_0 := 0$ and send c_v to its right neighbor.

Algorithm 1 Crash-Resilient 2-Coloring of a Path

$c_v := \bot$. If v has no left neighbor, it sets $c_v := 0$.
In every round:
    If $c_v \neq \bot$: Send c_v to your right neighbor. Output c_v and terminate.
    Send 'awake' to your right neighbor.
    If v has received no message from its left neighbor: $c_v := 0$.
    If v has received c_u from its left neighbor: $c_v := 1 - c_u$.

b) No: now the nodes cannot distinguish between crashes and messages simply getting delayed, and they still need to output some color. Assuming such an algorithm exists, it should be able to obtain a proper coloring even when the notification between all nodes is delayed until all nodes output a color. This would imply that one can 2-color a path with no communication.