Lecture 16
self-stabilization

distributed systems
CS425 / ECE 428 / CSE 424

sayan mitra

motivation

- as the number of computing elements increase in distributed systems failures become more common
- fault tolerance should be automatic, without external intervention
- two kinds of fault tolerance
  - masking: application layer does not see faults, e.g., redundancy and replication
  - non-masking: system deviates, deviation is detected and then corrected: e.g., roll back and recovery
- self-stabilization is a general technique for non-masking FT distributed systems

self-stabilization

- technique for spontaneous healing
- guarantees eventual safety following failures

feasibility demonstrated by Dijkstra (CACM `74)

E. Dijkstra

self-stabilizing systems

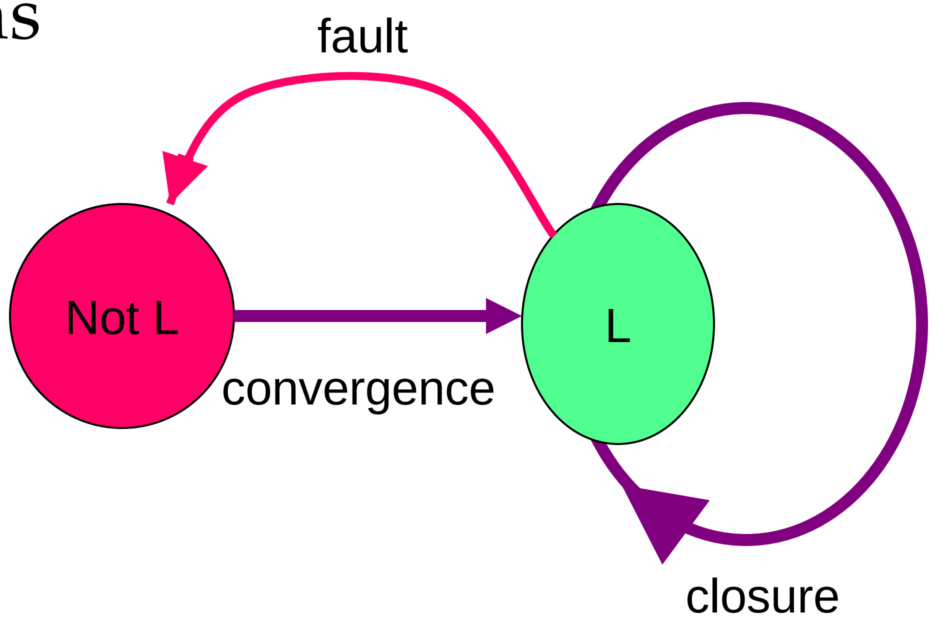
recover from any initial configuration to a legitimate configuration in a bounded number of steps, as long as the codes are not corrupted

assumption:

failures affect the state (and data) but not the the program

self-stabilizing systems

- self-stabilizing systems exhibits non-masking fault-tolerance
- they satisfy the following two criteria
  - convergence
  - closure



self-stabilizing systems

transient failures perturb the global state. The ability to spontaneously recover from any initial state implies that **no initialization is ever required**.

such systems can be deployed ad hoc, and are guaranteed to function properly in bounded time

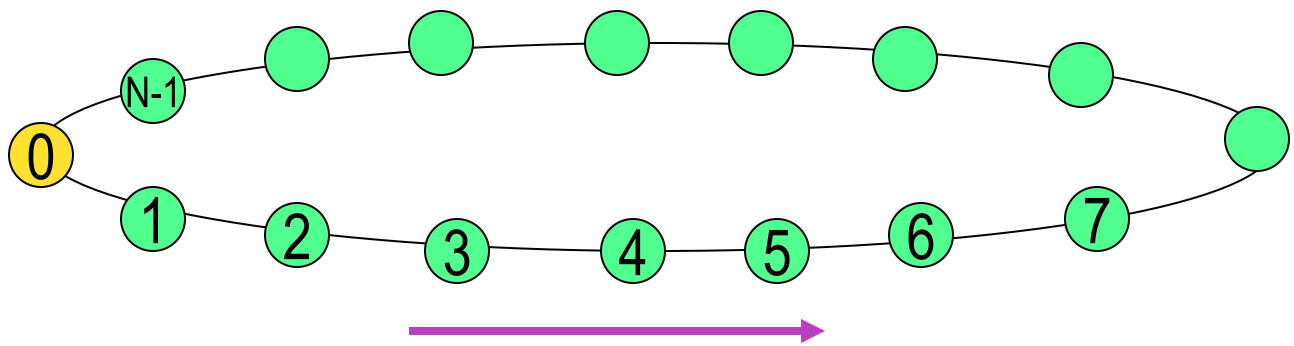
guarantees fault tolerance when the mean time between failures (MTBF) >> mean time to recovery (MTTR)

Outline

- Mutual exclusion on the ring
- Graph coloring
- Spanning tree

MUTUAL EXCLUSION ON THE RING

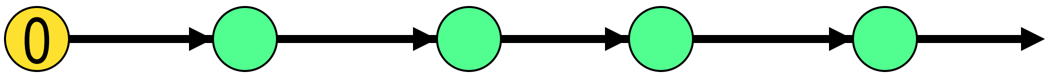
example 1: stabilizing mutual exclusion in unidirectional ring



consider a unidirectional ring of processes.
Legal configuration = exactly one token in the ring
desired "normal" behavior: single token circulates in the ring

Dijkstra's stabilizing mutual exclusion

N processes: 0, 1, …, N-1
state of process j is $x[j] \in \{0, 1, 2, K-1\}$, where $K > N$



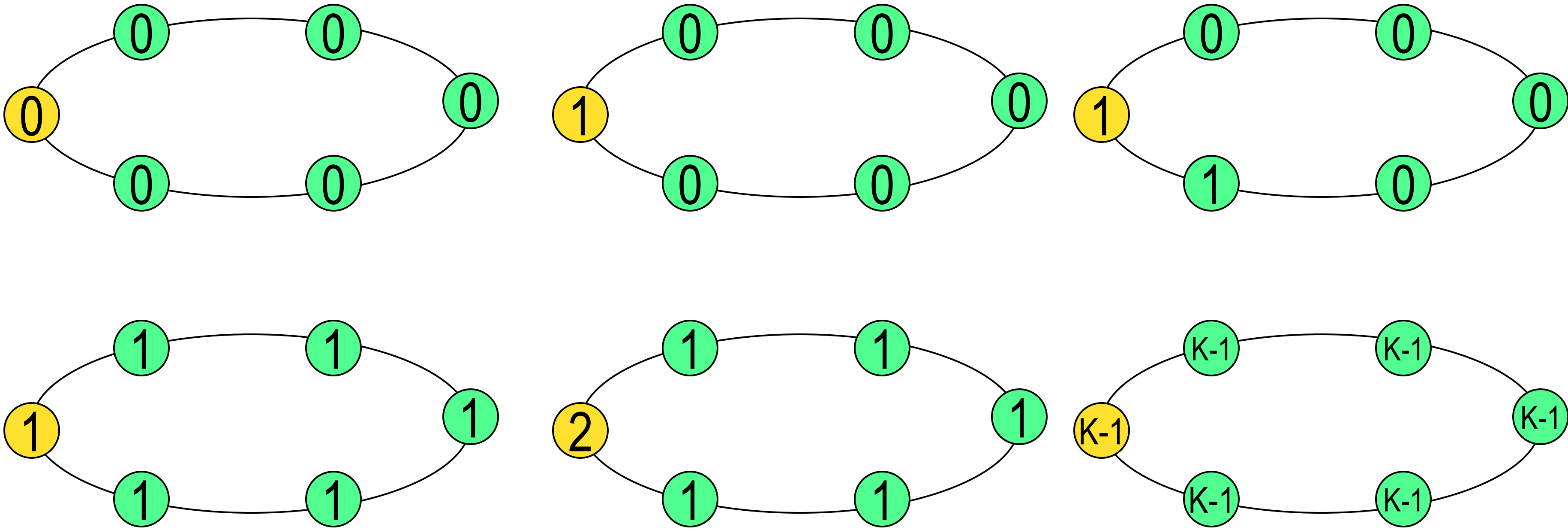
p_0 **if** $x[0] = x[N-1]$ **then** $x[0] := x[0] + 1$

p_j $j > 0$ **if** $x[j] \neq x[j-1]$ **then** $x[j] := x[j-1]$

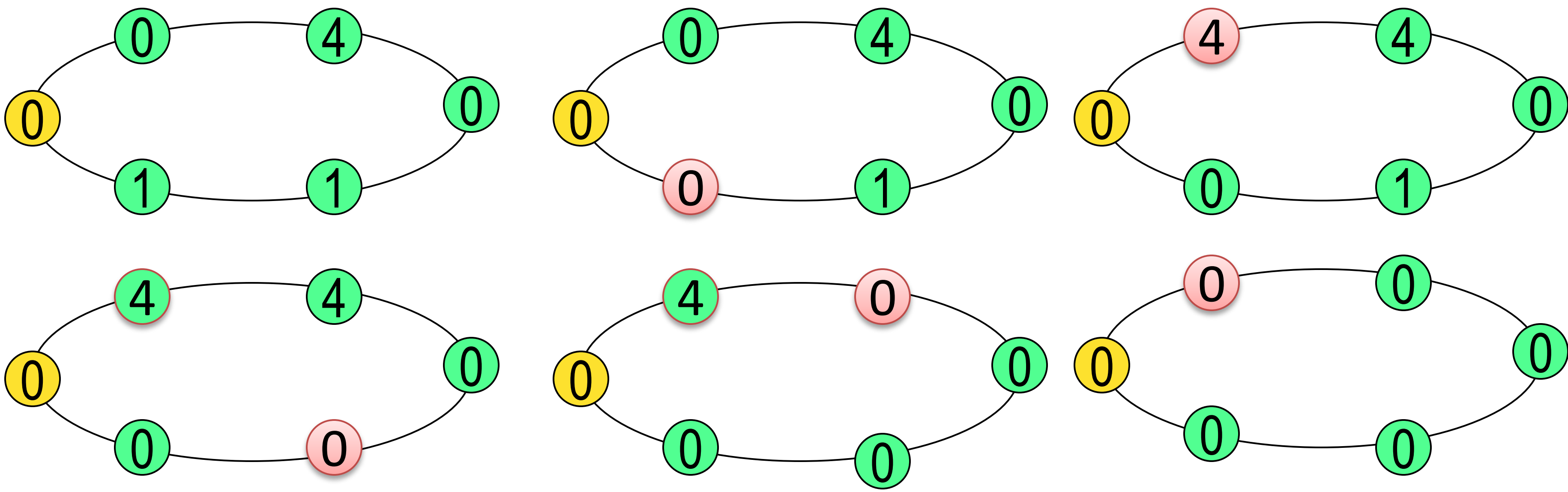
(TOKEN = if condition is true)

Legal configuration: only one process has token
start the system from an arbitrary initial configuration

example execution



example stabilizing execution



why does it work ?

1. at any configuration, at least one process can make a move (has token)
   - suppose $p_1,\ldots,p_{N-1}$ cannot make a move
   - then x[N-1] = x[N-2] = … x[0]
   - then p_0 can make a move

why does it work ?

1. at any configuration, at least one process can make a move (has token)
2. set of legal configurations is closed under all moves
   - if only p_0 can make a move then for all i,j x[i] = x[j] and after p_0's move, only p_1 can make a move
   - if only p_i ($i \neq 0$) can make a move
     - for all $j < i$, x[j] = x[i-1]
     - for all $k \geq i$, x[k] = x[i], and
     - $x[i-1] \neq x[i]$

   in this case, after p_i's moves only p_{i+1} can move

why does it work ?

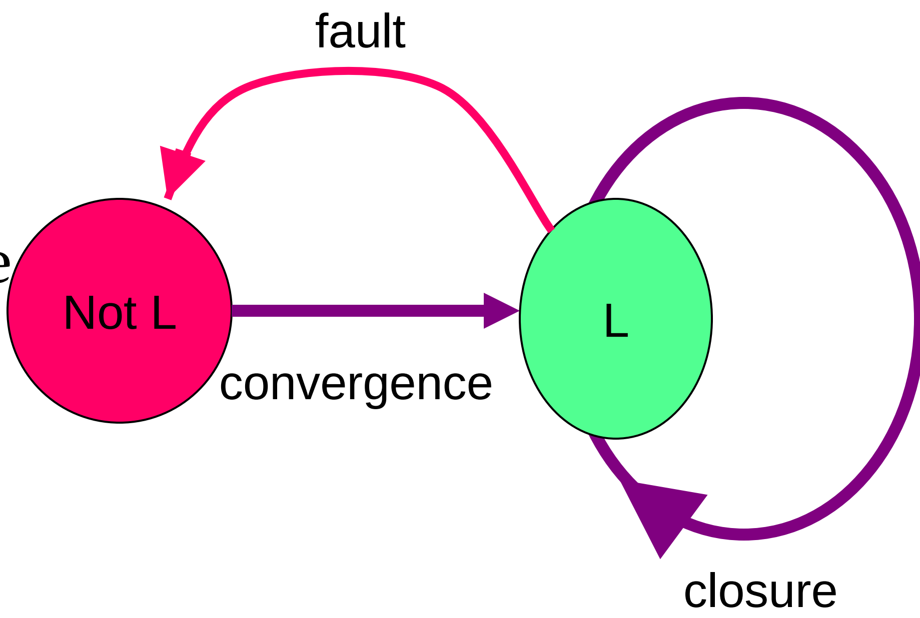
1. at any configuration, at least one process can make a move (has token)
2. set of legal configurations is closed under all moves
3. total number of possible moves from (successive configurations) never increases
   - any move by p_i either enables a move for p_{i+1} or none at all

why does it work ?

1. at any configuration, at least one process can make a move (has token)
2. set of legal configurations is closed under all moves
3. total number of possible moves from (successive configurations) never increases
4. all illegal configuration C converges to a legal configuration in a finite number of moves
   - there must be a value, say v, that does not appear in C
   - except for p_0, none of the processes create new values
   - p_0 takes infinitely many steps, and therefore, eventually sets x[0] = v
   - all other processes copy value v and a legal configuration is reached in N-1 steps

putting it all together

- Legal configuration = a configuration with a single token
- perturbations or failures take the system to configurations with multiple tokens
  - e.g. mutual exclusion property may be violated
- within finite number of steps, if no further failures occur, then the system returns to a legal configuration



mutual exclusion in bidirectional ring

N processes: 0, 1, …, N-1
state of process j, j > 0 and j < N-1 is $x[j] \in \{0, 1, 2, 3\}$
state of process 0, $x[0] \in \{1, 3\}$
state of process N-1, $x[N-1] \in \{0, 2\}$
neighbor of i = {i-1 mod N, i + 1 mod N}

$p_0\, p_{N-1}$ **if exists** neighbor j: x[j] = x[i] mod 4
then x[i] := x[i] + 1 mod 2

p_j 0<j<N-1 **if exists** neighbor j: x[j] = x[i] mod 4
then x[i] := x[j]

Exercise: show that this 4 state protocol stabilizes to a legal state in a finite number of steps.

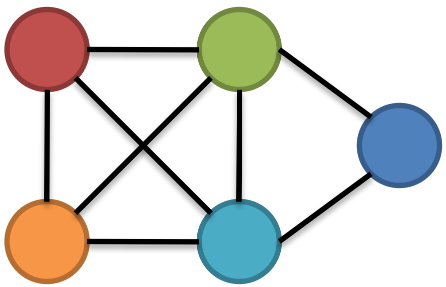
GRAPH COLORING

stabilizing graph coloring

- a graph coloring algorithm
- self-stabilizing graph coloring

graph coloring problem

- shared memory distributed system with N processes $p_0, \ldots, p_{N-1}$
  - induced undirected graph G = (V,E)
  - N_i: set of neighbors of p_i
  - $|N_i| \leq D$, maximum degree of any node D
  - set of all colors C, $|C| = D + 1$



- initially nodes are assigned arbitrary colors
- design an algorithm such that for all i, j
  - if $j \in N_i$ then $color_i \neq color_j$
- application: choosing broadcast frequencies in a wireless network in order to reduce interference

simple coloring algorithm

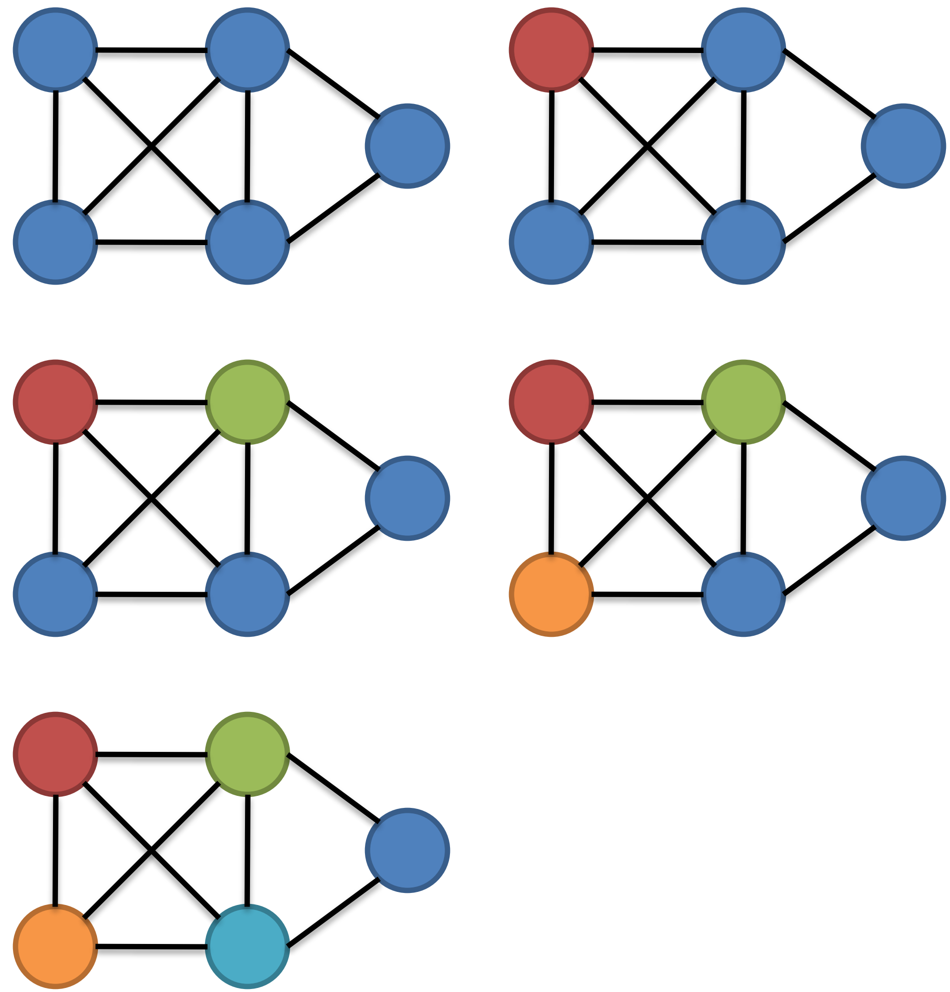
- program for process pi
  - NC = {c $\in$ C | exists $j \in N_i$, $color_j$ = c}
  - **if** there exists $j \in N_i$ such that $color_i = color_j$

    then $color_i$:= choose from C \ NC

- shared memory program: p_i can read $color_j$, $j \in N_i$ and set $color_i$ in a single atomic step

correctness of simple coloring (SC)

- each action resolves the color of a node w.r.t. its neighbors
- once a node gets a distinct color, it never changes its color
- each node changes color at most once, algorithm terminates after N-1 steps



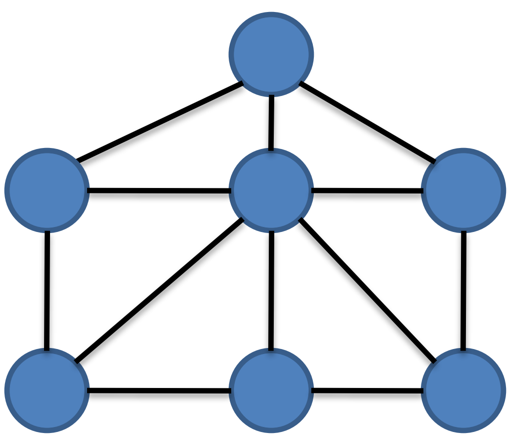
properties of SC

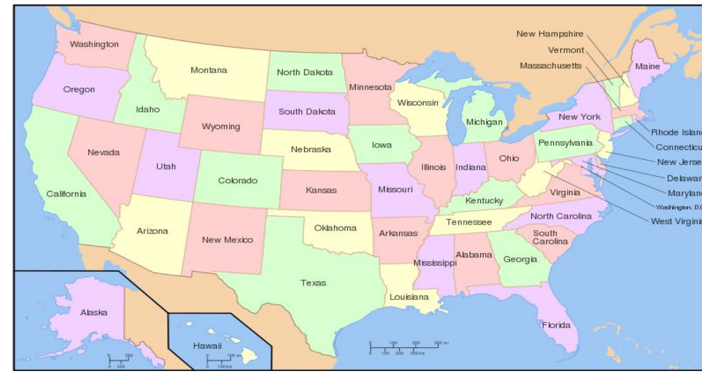
- Legal configuration = for all i, j, if $j \in N_i$ then $color_i \neq color_j$
- is SC self-stabilizing?
  - YES, does not require any initialization
  - from any initial coloring converges to a legal configuration, i.e., with correct coloring, in N-1 steps

- requires D+1 colors!
  - very suboptimal

“Four colors suffice”

- any **planar** graph can be colored with 4 colors!
- any 2D map can be colored with 4 colors
- this is the (famous) 4 color theorem
- proposed in 1852 when Francis Guthrie (to De Morgan), while trying to color the map of counties of England





Kenneth Appel and Wolfgang Haken (at UIUC!) announced to much acclaim that they had proven the four color theorem

their proof reduced the infinitude of possible maps to 1,936 reducible configurations which had to be checked one by one by computer and took > 1000 hours

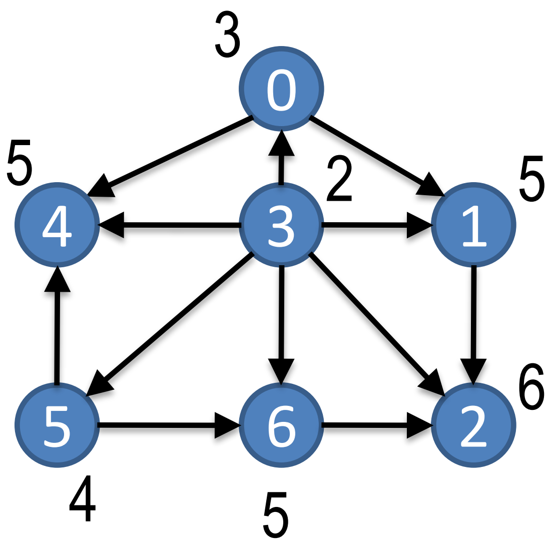
planar graph coloring

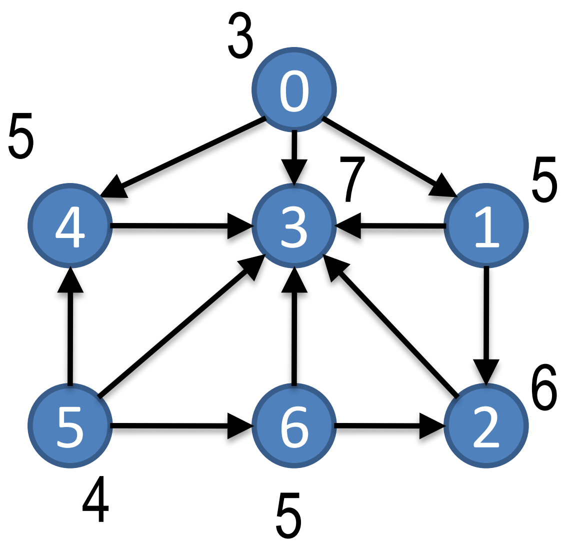
- with at most 6 colors
- key idea:
  - transform G to a directed acyclic graph (DAG) for which the degree of any node is at most 5
  - execute simple coloring algorithm on DAG

DAG generating algorithm

- process pi
  - integer variable x_i
    - $i \rightarrow j$ iff $x_i < x_j$ or $x_i = x_j$ and $i < j$
    - $i \leftarrow j$ otherwise
    - x_i's induce a directed acyclic graph (DAG)
  - succ(i) = { j | there exists <u>directed</u> edge (i,j) }
  - $sx_i = \{x_j \mid j \in succ(i)\}$
- how to ensure that the number of outgoing edges for every i is at most 5?
- program for p_i
  - if |succ(i)| > 5 then xi = max $\{sx_i\}$ + 1
- again, assuming large grain atomicity

example execution





correctness of DAG generation

Legal configuration = for all i, outdegree(i) ≤ 5

- in any planar graph $|V| > 2$ implies
  $|E| \leq 3|V| - 6$ (Euler's formula)
- **Corollary 1**. in any planar graph there is at least one node with degree ≤ 5

correctness of DAG generation

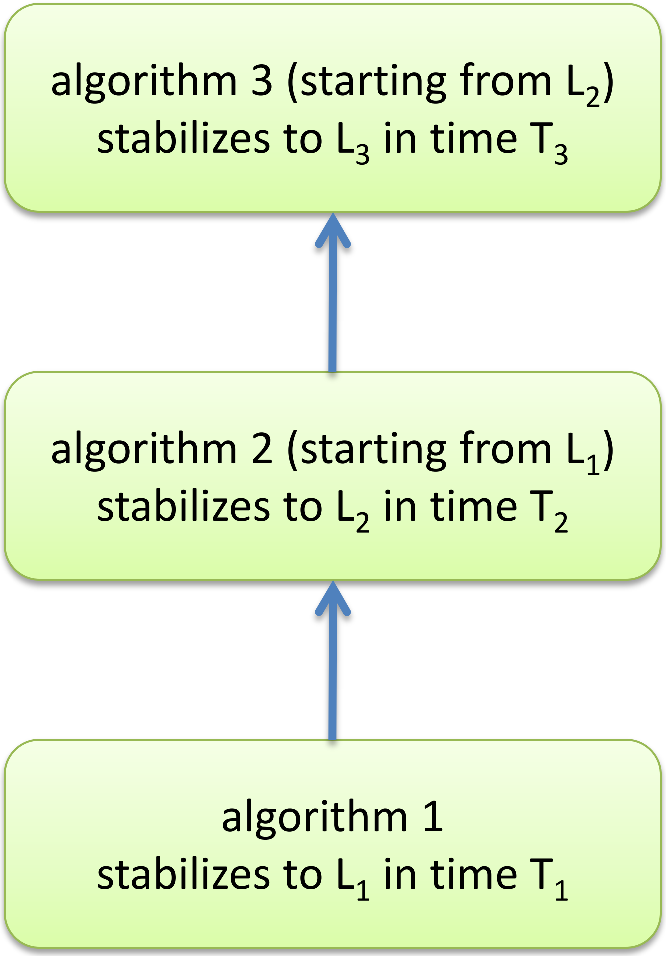
Legal configuration = for all i, outdegree(i) $\leq$ 5

DAG generation stabilizes in finite number of steps

- assume that the algorithm does not terminate
- there is at least one j that makes infinitely many moves
- in every move, j makes all edges point inward
- so, between two successive moves of j, 6 other nodes in succ(j) must be moving
  - at least 6 nodes in succ(j) will make infinitely many moves
  - so, there exists a subgraph in which every node has degree > 5 and in which nodes move infinitely
- subgraph is also a planar graph, contradicts Corollary 1.

stack of stabilizing protocols

- DAG generation stabilizes in finite number of steps
- if DAG is stable then SC stabilizes in a finite number of steps
- thus, overall coloring stabilizes in a finite number of steps

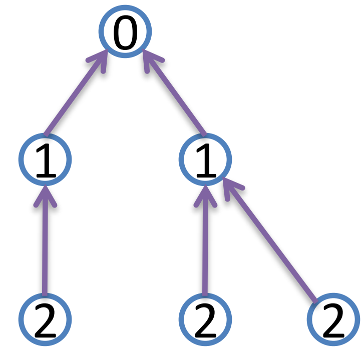


self-stabilizing spanning tree

assumptions

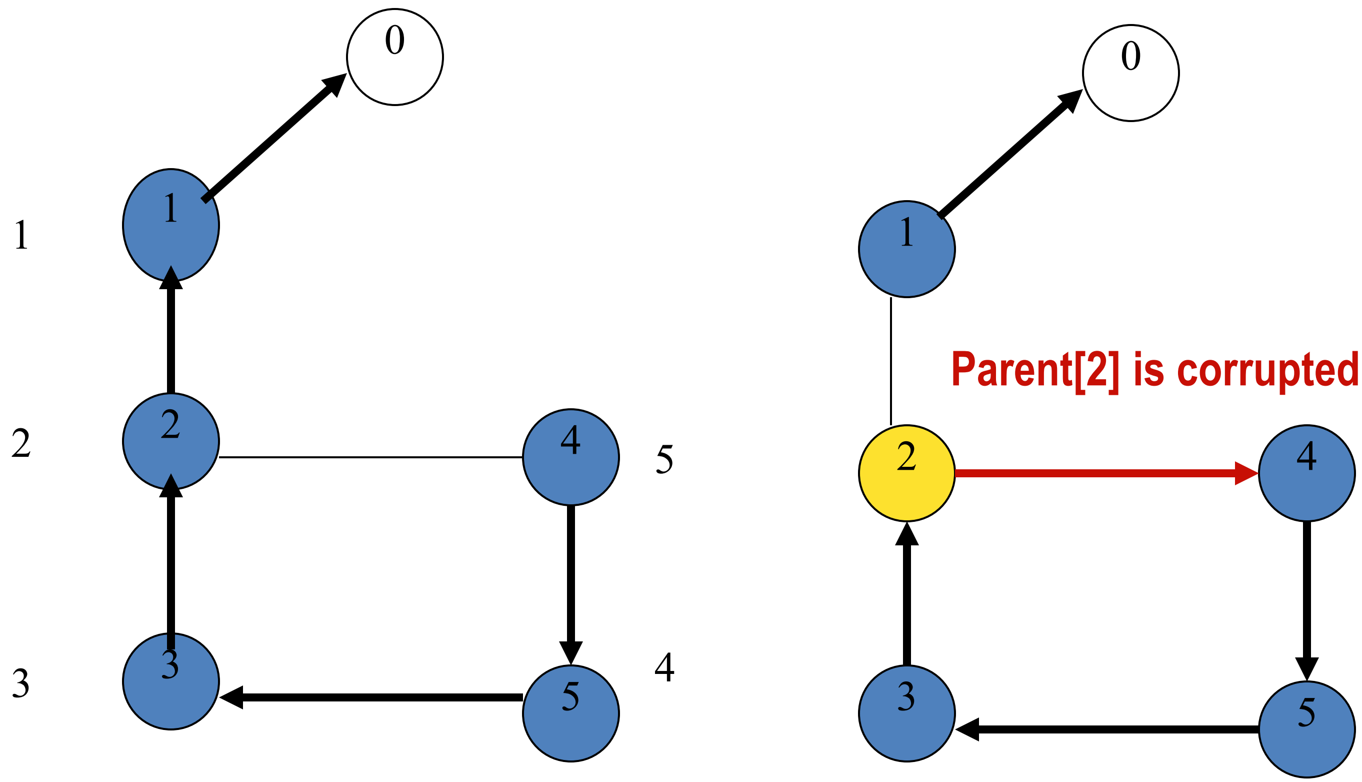
- topology is a connected graph G=(V,E)
- failures add and remove edges and vertices without disconnecting G
- failures also corrupt software state (as usual)
- let $n = |V|$
- shared memory

algorithm for spanning tree



- process *pi*
- state variables
  - parent[i]: parent pointer
  - L[i]: level
  - N[i]:set of neighbors of i
- there is a distinguished root process r (always idle)
- Legal configuration:
  - L[r] = 0, parent[r] is undefined
  - for all i, $i \neq r$ ::
    - L[i] < n and
    - L[parent[i]] < n -1 and
    - L[i] = L[parent[i]] + 1

an illegal configuration



algorithm

process p_i

if $(L[i] \neq n) \wedge (L[i] \neq L[parent[i]] + 1) \wedge (L[parent[i]] \neq n)$

then $L[i] := L[parent[i]] + 1$ (0)

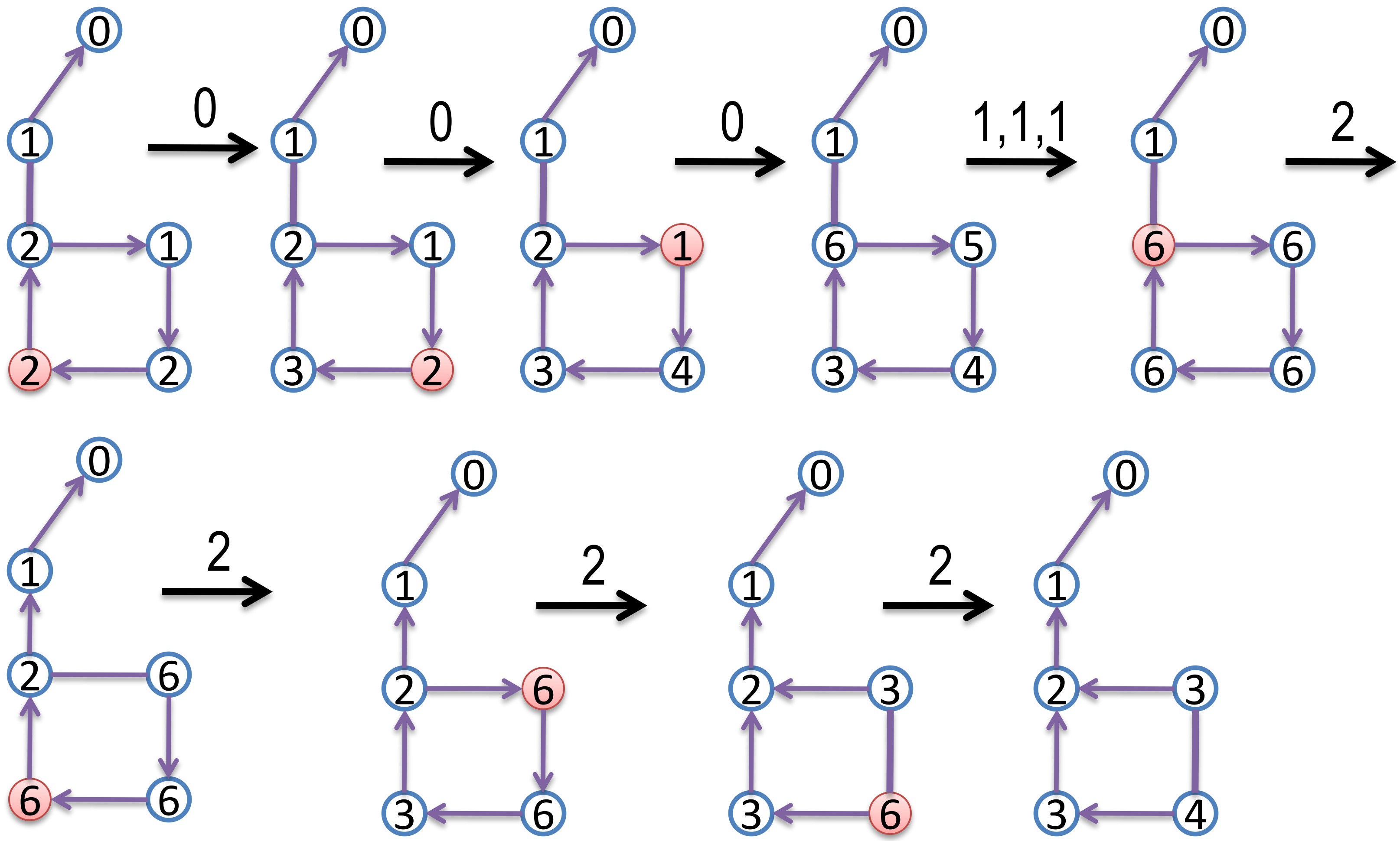
if $(L[i] \neq n) \wedge (L[parent[i]] = n)$

then $L[i] := n$ (1)

if $(L[i] = n) \wedge (\exists k \in N[i] : L[k] < n-1)$

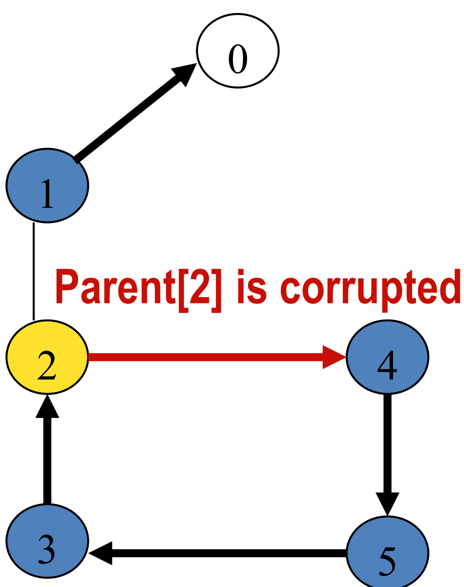
then $L[i] := L[k]+1$; $parent[i] := k$ (2)

stabilizing execution



proof of stabilization

- define an edge from **i** to **parent[i]** to be *well-formed*, when
  - **L[i] ≠ n, L[parent[i]] ≠ n** and **L[i] = L[parent[i]] +1**
- in any configuration, the well-formed edges form a *spanning forest*
  - delete all edges that are not well-formed
  - designate each tree T(k) in the forest by the *lowest value of L* in it



T(0) = {0, 1}

T(2) = {2, 3, 4, 5}

Let F(k) denote the number of T(k) in the forest.

Define a tuple **F= (F(0), F(1), F(2) …, F(n)).**

For the sample graph, **F = (1, 0, 1, 0, 0, 0)** after node 2 has a transient failure.

skeleton of the proof

Minimum F = (1,0,0,0,0,0) {legal configuration}

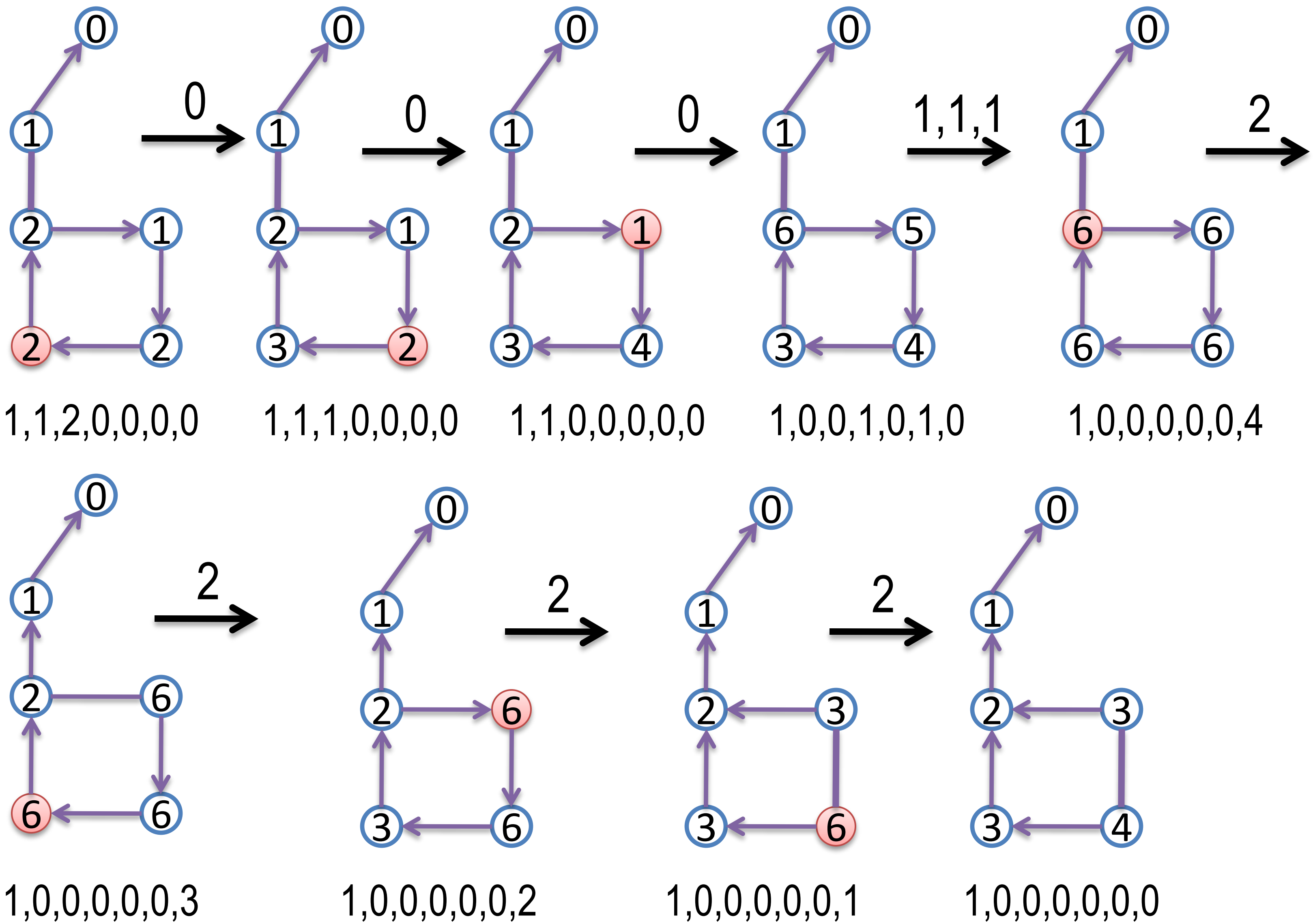
Maximum F = (1, n-1, 0, 0, 0, 0).

With each action of the algorithm, F decreases lexicographically. Verify the claim!

This proves that eventually F becomes (1,0,0,0,0,0) and the spanning tree stabilizes.

What is the time complexity of this algorithm?

stabilizing execution



other stabilizing algorithms

- see handout for a stabilizing algorithm for
  - distributed reset
  - stabilizing clock synchronization

summary

- self-stabilizing algorithms recover automatically to legal configurations after faults cease in a finite number of steps
  - assuming the program does not get corrupted
- should have two key properties
  - closure
  - Convergence
- permit compositional reasoning
- typically they maintain little state information
- examples: mutual exclusion, coloring, DAG formation, more next lecture