Self-Stabilization

M.T. Hajiaghayi and V.S. Mirrokni

Massachusetts Institute of Technology

Introduction

- Fault-tolerant algorithms, but different from those studied in the class (the idea proposed by Dijkstra in '74)
- Guarantee a return to correct behavior within finite time, after all *transient* failures stop

We assume the programs of processes can not be corrupted,

while the contents of their memories can be corrupted

- We start the algorithm from an arbitrary initial state, guarantee eventually behave correctly

Outline

- Model definitions and model conversions
- Advantages and disadvantages of self-stabilization
- Some examples:
  - Mutual exclusion
  - Spanning tree
- Design technique: Fair composition
- A proof technique: Variant Function
  - Maximal matching
- Fault-tolerant self-stabilization

Model Definitions

We consider asynchronous read-write shared memory model

Configuration

- A tuple $c = (s_1, s_2, \cdots, s_n, r_1, r_2, \cdots r_m)$, where s_i is the state of P_i and r_j is the content of a communication register.

System

- A pair $S = (\mathcal{C}, \rightarrow)$ where $\mathcal{C}$ is a set of configurations and $\rightarrow$ is a binary transition relation on $\mathcal{C}$.

Execution

- A sequence $E = (\gamma_0, \gamma_1, \cdots)$ such that $\gamma_i \rightarrow \gamma_{i+1}$.

Self-Stabilization Requirements

System S **stabilizes to property** P **if there exists a subset** $\mathcal{L} \subseteq \mathcal{C}$ **of** *safe (or legal) configurations* **satisfying:**

- **Correctness**: Every configuration in $\mathcal{L}$ satisfies P
- **Closeness**: If $\gamma \in \mathcal{L}$ and $\gamma \to \delta$, then $\delta \in \mathcal{L}$
- **Convergence**: Every execution contains a configuration of $\mathcal{L}$

Complexity Measures

Time Complexity: the number of *asynchronous rounds* to reach a safe configuration from any initial configuration

- *Asynchronous round:* a part of execution in which each process executes at least one step (parameter l in our termination)

- This definition nullifies the speed difference of the processes

Space (memory) complexity: the number of (local and shared) memory bits used for implementation

- It is not so important in the analyze

Model Conversions

Shared memory model can be converted to message passing model [GM91]

Synchronous algorithms can be converted to asynchronous algorithms using:

- A version of α-synchronizer [AKM+93]
- A version of β-synchronizer [Dim91]

Id-based systems can be converted to uniform randomized systems [Dol94, Dol98]

Advantages of Self-Stabilization

- **Fault tolerance:** offer full and automatic protection against all transient process failures

- **Initialization:** no need of proper initialization since the algorithm recovers from any configuration

- **Sequential composition:** combining previous algorithms to obtain new ones.

- **Simplicity:** simpler than their classical counterparts for the same problem, but have the same functionality

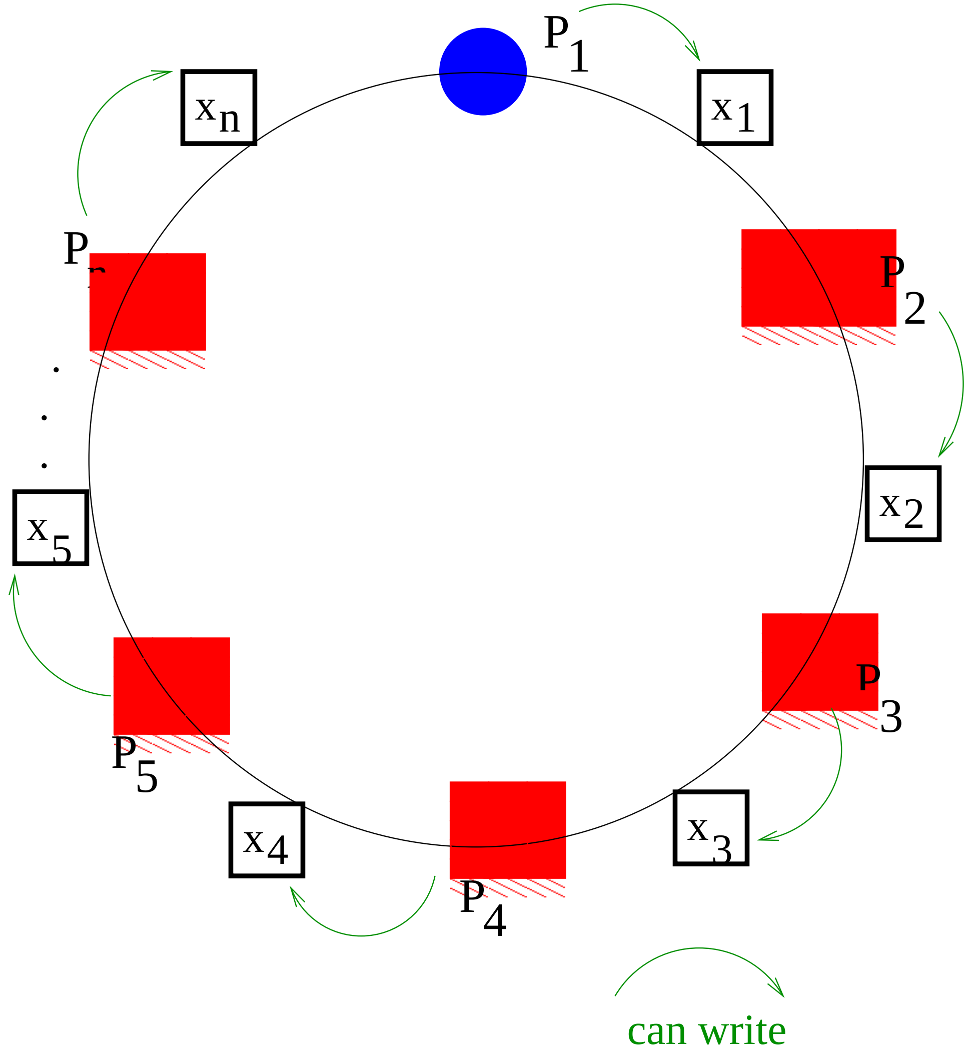
Disadvantages of Self-Stabilization

- **Initial inconsistencies:** before reaching a safe configuration, it shows an inconsistent output

- **High Complexity:** less efficient than their classical counterparts

- **Non-termination:** It's not observable within the system that a safe configuration has been reached

Dijkstra's Mutual Exclusion



P_1:

do forever

if $x_1 = x_n$ **then**

crit-reg

$x_1 := (x_1 + 1) \bmod (n + 1)$

$P_i (i \neq 1)$:

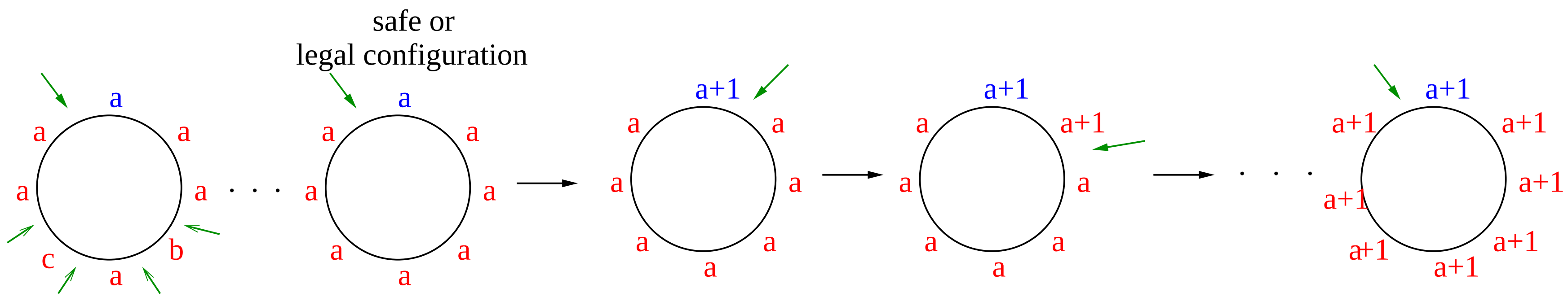
do forever

if $x_i \neq x_{i-1}$ **then**

crit-reg

$x_i := x_{i-1}$

Example: Dijkstra's Mutual Exclusion



Proof Sketch: Dijkstra's Mutual Exclusion

- A configuration c in which all x_i's have the same value is *safe*

We show in each execution, a safe configuration is reached after a finite number of rounds

- For each configuration c, there exist at least one integer $0 \leq j \leq n$ s.t. for all i, $x_i \neq j$ in c

- For each configuration c, every execution starts from c, P_1 changes x_1 in every n rounds

For each configuration c, every fair execution that starts from c reaches a safe configuration within $O(n^2)$ rounds

Root: $//r_{ij} =< j \ IsParentOf \ i, dis >$, P_i writes, P_j reads

do forever

for $m := 1$ to δ **do** **write** $r_{im} :=< 0, 0 >$

Other: // δ: the number of neighbors

do forever

for $m := 1$ to δ **do** lr_{mi}:=**read**(r_{mi})

dist:= $1 + min\{lr_{mi}.dis | 1 \leq m \leq \delta\}$

for $m := 1$ to δ **do**

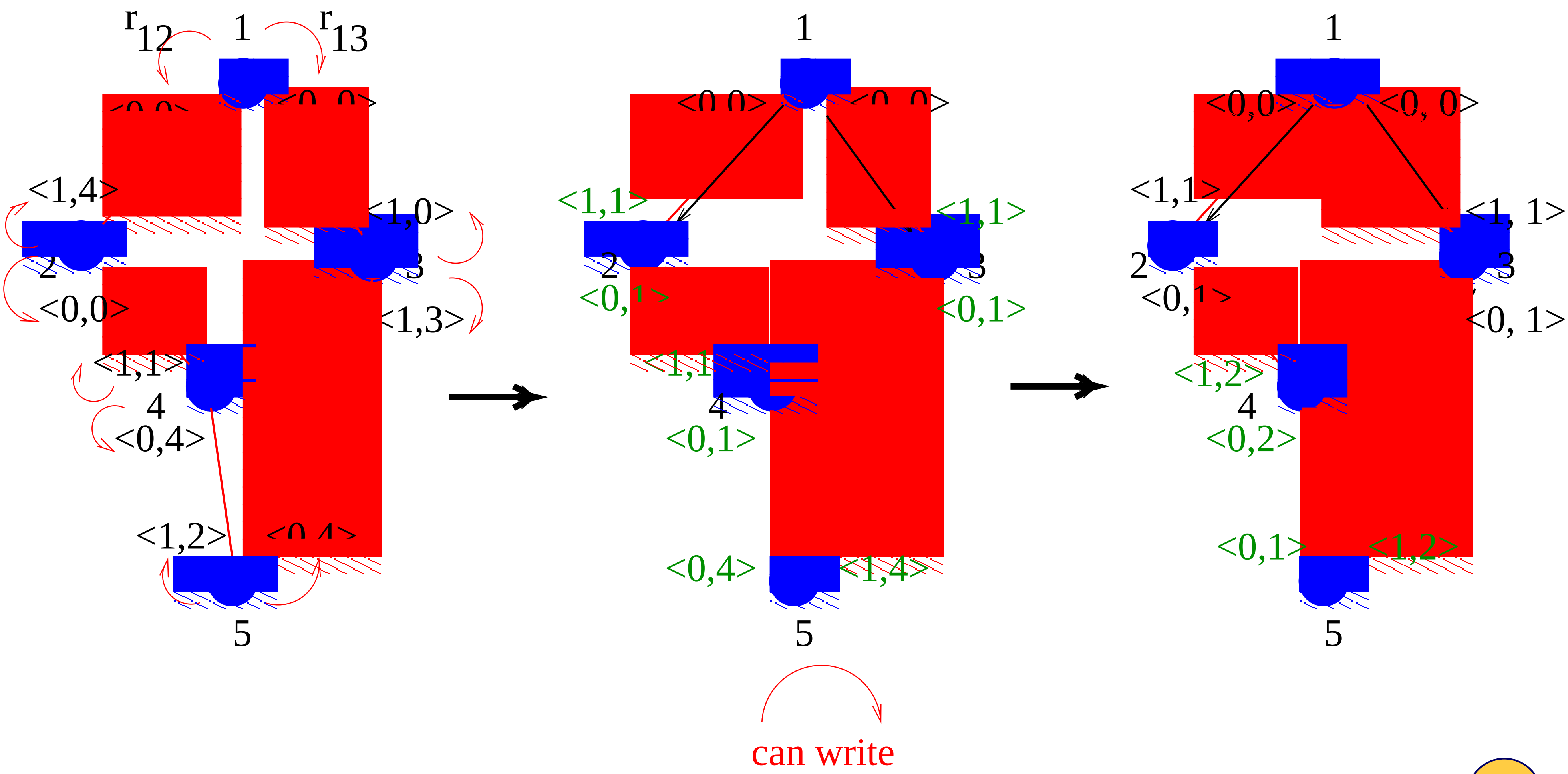
if $lr_{mi}.dis = dist - 1$ **then**

write $r_{im} :=< 1, dist >$

else write $r_{im} :=< 0, dist >$

Example: Spanning Tree

r_{ij}=<j is parent of i, dis>



Proof Sketch: Spanning Tree

- A *floating distance (FL)* in configuration c is a value in register $r_{ij}.dis$ that is smaller than the actual distance of P_i from the root
- The *smallest floating distance (SFL)* is the smallest value among the floating distances

Lemma: For every $k > 0$ and for every configuration that follows $\Delta(1+4k)$ rounds (Δ is the maximum degree)

- If there exists a FL, then $SFL \geq k$
- The values of the registers of each process within distance k from the root is equal to its distance from the root

Proof: by induction

Thus the algorithm stabilizes after $O(n\Delta)$ rounds

Fair Composition

- Two algorithms AL1 and AL2
- AL1$\bowtie$AL2: executes AL1 and AL2 alternately. (fair composition)
- Process P_i executes AL1$\bowtie$AL2.
- A_i = states of P_i in AL1.
- $S_i = A_i \times B_i$ = states of P_i in AL2.

Fair Composition

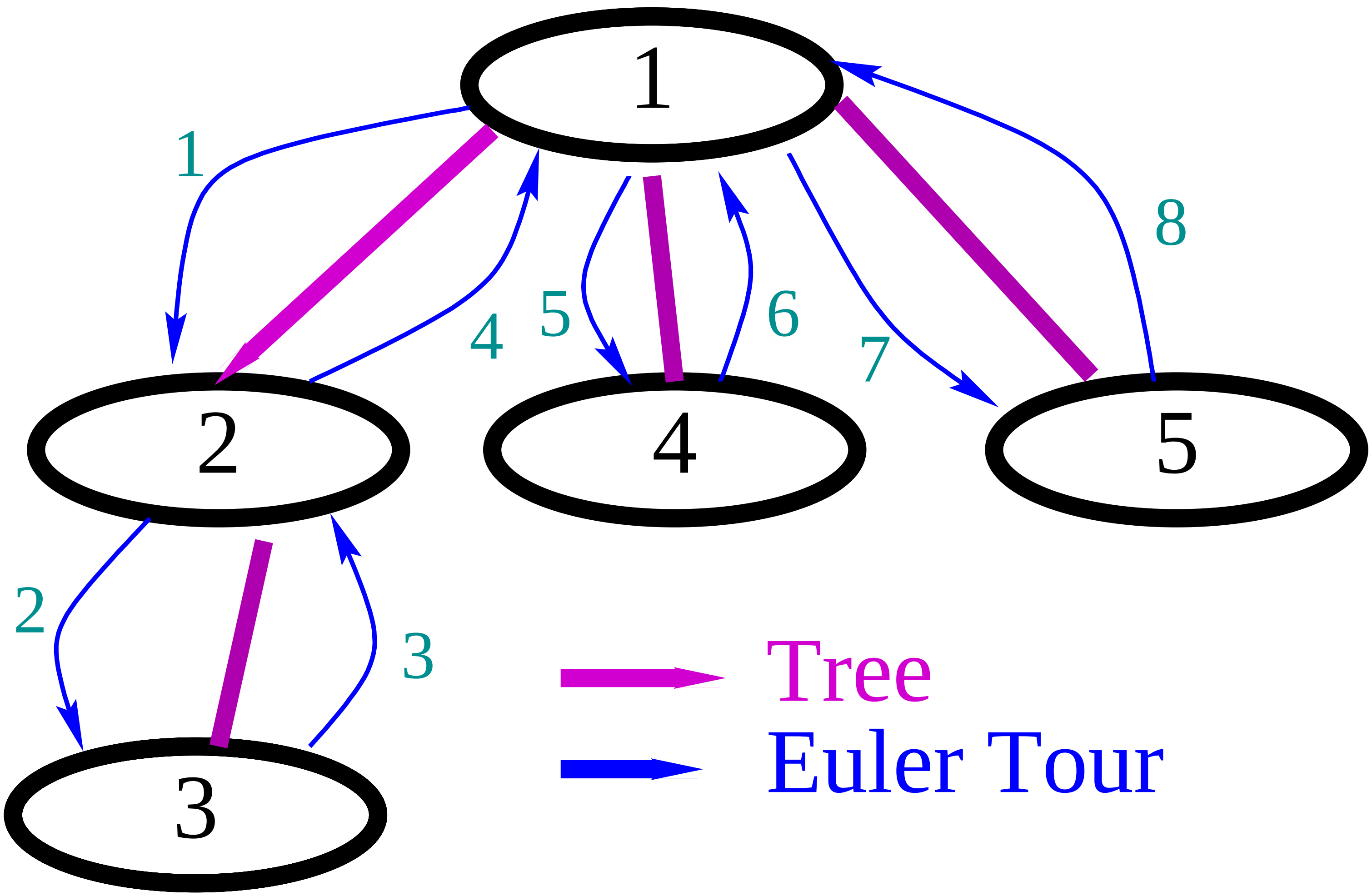
- When P_i executes AL1 $\rightarrow$ read and modify variable of A_i
- When P_i executes AL2 $\rightarrow$ read A_i and B_i, but just modify B_i
- AL1 will stabilize to θ
- When AL1 reaches θ, it does not change variables of A_i read by AL2
- AL2 will stabilize to ψ if θ holds for A_i

> Theorem: AL1$\bowtie$AL2 will stabilize to ψ.

Example: Fair Composition

- AL1= spanning tree
- constructing Euler tour from spanning tree
- AL2 = Dijkstra algorithm: mutual exclusion in a ring
- $AL1 \bowtie AL2$ = mutual exclusion in general graphs
- Idea: AL1 will provide spanning tree and an Euler tour in some variables. AL2 will use that tour as a virtual ring to perform mutual exclusion in general graphs.

Euler Tour on a Tree



Variant Functions

- **Potential function with bounded value**
- **this function monotonically decreases (increases) after each step**
- **function $<$ threshold $\Rightarrow$ safe-configuration**

Maximal Matching

Matching

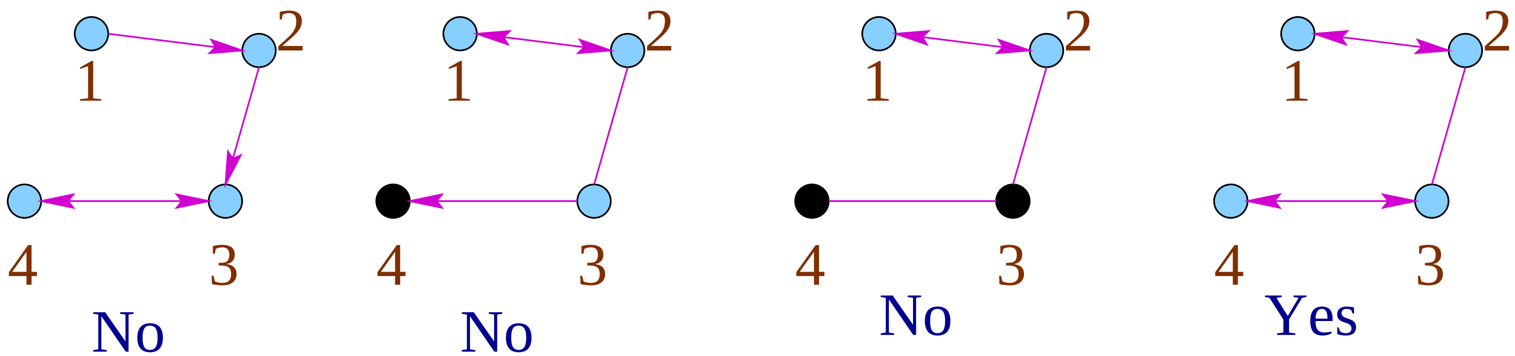
- A subset of non-adjacent edges in a graph

Maximal matching

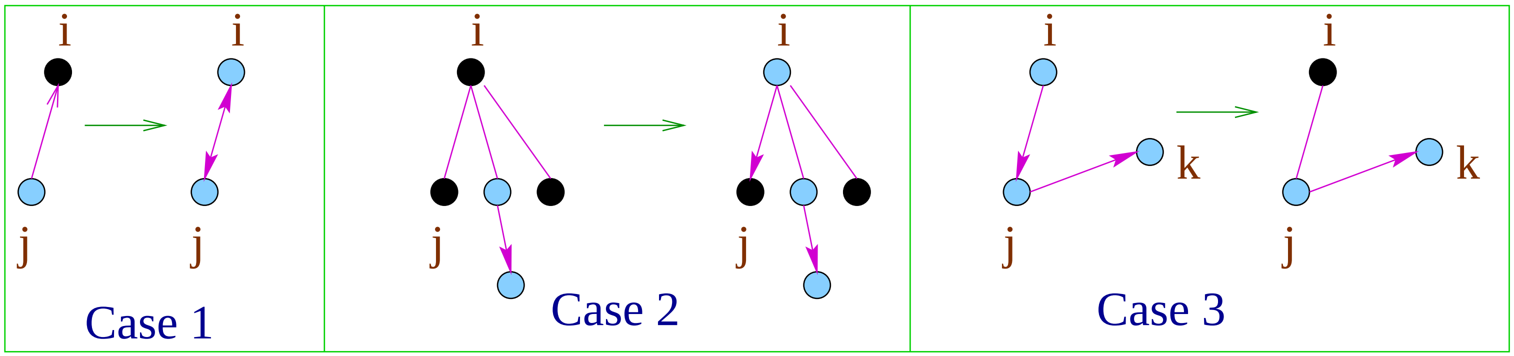
- No edge can be added to it

- Each process has a pointer to one of its neighbors:
- Pointer$_i = j \Rightarrow$ Pointer$_j = i$ and $P_j \in Neighbor(i) \leftarrow$ Matching
- Pointer$_i = null$ and Pointer$_j = null \Rightarrow (i, j) \notin E(G) \leftarrow$ Maximal

Examples of Maximal Matching



Algorithm Idea



Algorithm

do forever

if Pointer$_i = null$ and $(\exists P_j \in N(i)|$Pointer$_j = i)$ then

Pointer$_i = j$;$\leftarrow$ Case 1

if Pointer$_i = null$ and $(\forall P_j \in N(i)|$Pointer$_j \neq i)$ and

$(\exists P_j \in N(i)|$Pointer$_j = null)$ then

Pointer$_i = j$; $\leftarrow$ Case 2

if Pointer$_i = j$ and Pointer$_j = k$ and $k \neq i$ then

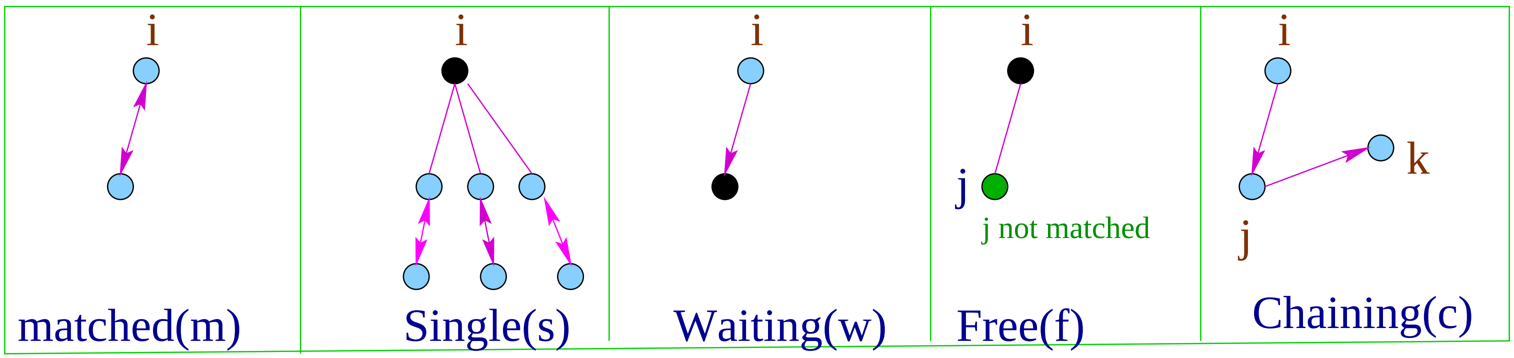
Pointer$_i = null$;$\leftarrow$ Case 3

Why will this algorithms be stabilized??

Assumption

- Processes will be activated one at a time, read its neighbor's variables and perform its code

Definition



Process Groups

We call P_i in configuration c_l,

- matched in c_l, if P_i has a neighbor P_j such that Pointer$_i = j$ and Pointer$_j = i$. $\leftarrow m$
- single in c_l, if Pointer$_i = null$ and every neighbor of P_i is matched. $\leftarrow s$
- waiting in c_l, if P_i has a neighbor P_j such that Pointer$_i = j$ and Pointer$_j = null$. $\leftarrow w$
- free in c_l, if P_i has a neighbor P_j such that Pointer$_i = j$ and Pointer$_j = null$. $\leftarrow f$
- chaining in c_l, if there exist a neighbor P_j for which Pointer$_i = j$ and Pointer$_j = k$, $k \neq i$. $\leftarrow c$

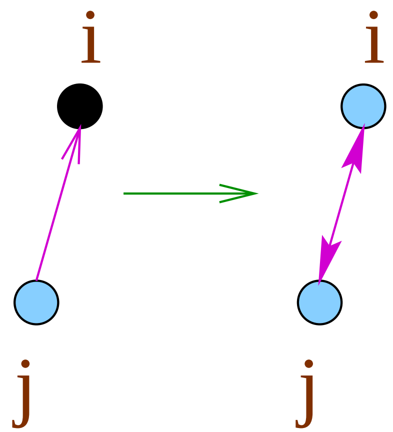
Variant Function and Proof

Potential Function

- For configuration c_l, $(m + s, w, f, c)$
- $(n, 0, 0, 0)$ is a safe-configuration
- every change in a pointer increases $(m + s, , w, f, c)$

Proof: Case 1

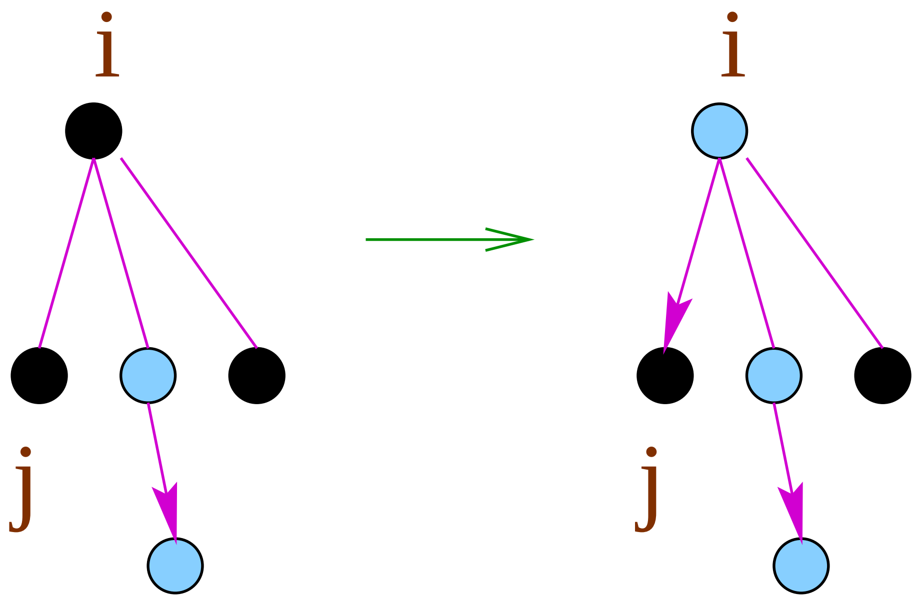
- $m = m + 2, f = f - 1, w = w - 1 \Rightarrow$ increases the function



Case 1

Proof: Case 2

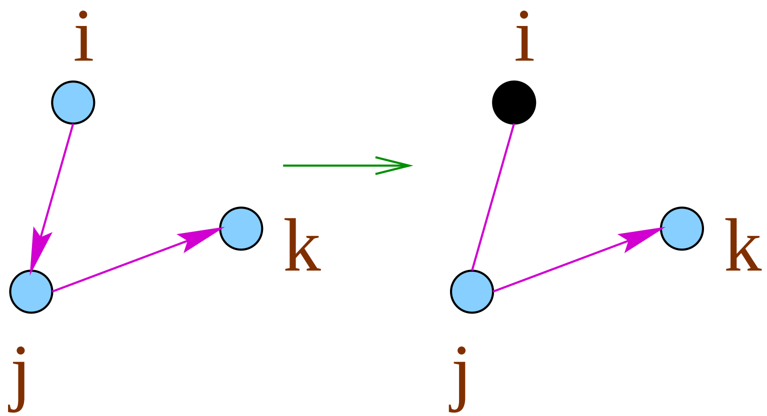
- $w = w + 1, f = f - 1 \Rightarrow$ increases the function



Case 2

Proof: Case 3

- P_i is chaining: $c = c - 1$
- $c = c - 1, (f = f + 1)$or$(s = s + 1)$ or
- $c = c - 2, w = w + 1, f = f + 1 \Rightarrow$ increases the function



Case 3

Analysis

- number of different situations for $(m+s, w, f, c)$ is $O(n^3)$ because $m+s+w+f+c \leq n$
- It takes at most $O(n^3)$ steps to reach a safe configuration.

- using function $(c+f+w, 2c+f) \Rightarrow O(n^2)$

Tolerating other Faults with stabilization

- Digital clock synchronization
- Self-Stabilization using bounded and unbounded clock
- Stopping failure
- Tolerate Byzantine faults in synchronize system
- Impossibility of stabilization in asynchronous systems

References

- Edger W. Dijkstra, Self-stabilizing systems in spite of distributed control. Communication of the ACM, 17(11):643–644, November 1974.

- Shlomi Dolev, Self-Stabilization. MIT Press, 2000.

- Gerald Tel, Introduction to Distributed Algorithms, Cambridge Univ. Press, 1994.