

Introduction to Self-stabilization

Swan Dubois / Alexandre Maurer / Franck Petit /
Maria Potop Butucaru / *Sébastien Tixeuil*

UPMC

Firstname.Lastname@lip6.fr

Outline

Self-stabilization

Hypothesis

Atomicity

Scheduling

Proof Techniques

Transfer Function

Convergence stairs

Conclusion

Example

- ▶ $U_0 = a$
- ▶ $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- ▶ $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd

Example

- ▶ $U_0 = a$
- ▶ $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- ▶ $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd

n	0	1	2	3	4	5	6	7	8	9	10	11
U_n	7	11	17	26	13	20	10	5	8	4	2	1

Example

- ▶ $U_0 = a$
- ▶ $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- ▶ $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd

27	41	62	31	47	71	107	161	242
121	182	91	137	206	103	155	233	350
175	263	395	593	890	445	668	334	167
251	377	566	283	425	638	319	479	719
1079	1619	2429	3644	1822	911	1367	2051	3077
4616	2308	1154	577	866	433	650	325	488
244	122	61	92	46	23	35	53	80
40	20	10	5	8	4	2	1	...

Example

- ▶ $U_0 = a$
- ▶ $U_{n+1} = \frac{U_n}{2}$ if U_n is even
- ▶ $U_{n+1} = \frac{3U_n+1}{2}$ if U_n is odd
- ▶ Converges towards a “correct” behavior
 - ▶ 1212121212121212121212121212...
 - ▶ Independent from the initial value

Example

► Enumerator of Even Numbers

```
unsigned char x = 0;
...
for (;;)
{
    ...
    printf ("%d ", x);
    x = x + 2;
    ...
}
```

Example

- Self-Stabilizing Enumerator of Even Numbers (Overflow Control)

```
unsigned char x;  
...  
for (;;)   
{  
    ...  
    printf ("%d ", x);  
    x = ( (x = x + 2) > 254 ) ? 0 : x + 2;  
    ...  
}
```


Example

- Self-Stabilizing Enumerator of Even Numbers (Parity Check)

```
unsigned char x;  
...  
for (;;)   
{  
    ...  
    printf ("%d ", x);  
    x = (x % 2) ? x + 1 : x + 2;  
    ...  
}
```

Example

- Self-Stabilizing Enumerator of Even Numbers (Parity Check—Reset)

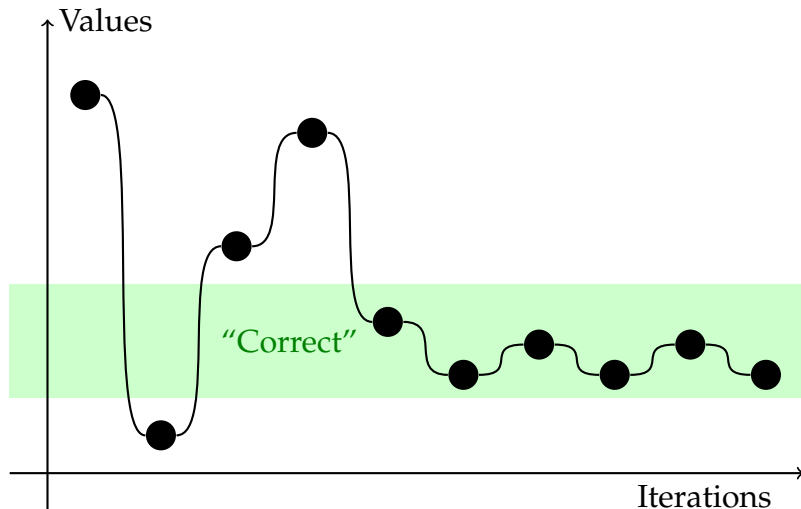
```
unsigned char x;  
...  
for (;;)   
{  
    ...  
    printf ("%d ", x);  
    x = (x % 2) ? 0 : x + 2;  
    ...  
}
```

Example

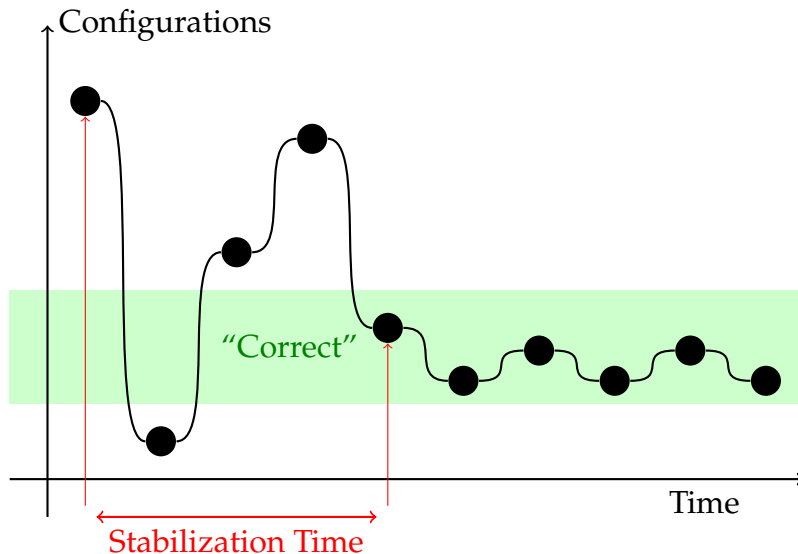
- Self-Stabilizing Enumerator of Even Numbers (Left Shift)

```
unsigned char x;  
...  
for (;;)   
{  
    ...  
    printf ("%d ", x<<1);  
    x++;  
    ...  
}
```

Example



Self-stabilization



Memory Corruption

- Example of a sequential program:

```
int x = 0;
...
if( x == 0 ) {
    // code assuming x equals 0
}
else {
    // code assuming x does not equal 0
}
```

Distributed Systems

- ▶ Locality of information

Distributed Systems

- ▶ Locality of information
- ▶ Locality of time

Distributed Systems

- ▶ Locality of information
- ▶ Locality of time
- ▶ \Rightarrow non-determinism

Distributed Systems

- ▶ Locality of information
- ▶ Locality of time
- ▶ \Rightarrow **non-determinism**

Definition (Configuration)

Product of the local states of the system components.

Definition (Execution)

Interleaving of the local executions of the system components.

Distributed Systems

Definition (Classical System, *a.k.a.* Non stabilizing)

Starting from a **particular** initial configuration, the system **immediately** exhibits correct behavior.

Definition (Self-stabilizing System)

Starting from **any** initial configuration, the system **eventually** reaches a configuration from which its behavior is correct.

Self-stabilization

Definition (Self-stabilizing System)

Starting from **any** initial configuration, the system **eventually** reaches a configuration from which its behavior is correct.

- Defined by Dijkstra in 1974

Self-stabilization

Definition (Self-stabilizing System)

Starting from **any** initial configuration, the system **eventually** reaches a configuration from which its behavior is correct.

- ▶ Defined by Dijkstra in 1974
- ▶ Advocated by Lamport in 1984 to address fault-tolerant issues

Fault Tolerance

Definition ((Distributed) Task)

A task is **specified** in terms of:

Fault Tolerance

Definition ((Distributed) Task)

A task is **specified** in terms of:

- ▶ **Safety:** *Bad actions*, which should not happen
 - ▶ At the intersection, traffic lights are green on two different axes.
 - ▶ Processes enter the critical section simultaneously.
 - ▶ *Windows* crashes.

Fault Tolerance

Definition ((Distributed) Task)

A task is **specified** in terms of:

- ▶ **Safety:** *Bad actions*, which should not happen
- ▶ **Liveness:** *Good actions*, which should (eventually) happen
 - ▶ At the intersection, if one of the traffic lights is red then, it eventually becomes green.
 - ▶ Every process eventually enter the critical section.
 - ▶ *Windows* eventually reboots.

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

- ▶ At the intersection, traffic lights are off.
- ▶ A process requesting the critical section is stuck.
- ▶ *Windows* boot loops on a blue screen with white markings.

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

- ▶ Type \rightarrow fail-stop, crash, omission, Byzantine, ...
- ▶ Duration
- ▶ Detection Rate
- ▶ Correction Rate
- ▶ Frequency

Fault Tolerance

Definition (Fault)

A fault is an action that corrupts the task specification by changing the correct functioning of a system component.

- ▶ Type \rightarrow fail-stop, crash, omission, Byzantine, ...
- ▶ Duration
- ▶ Detection Rate
- ▶ Correction Rate
- ▶ Frequency

Fault-tolerant algorithm \Rightarrow Tolerates a given class of faults

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed.

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed. Unfortunately, **[FLP]!**

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed. Unfortunately, **[FLP]!**
- ▶ **Fail-Safe FT:** *Safety* guaranteed but not *Liveness*.

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed. Unfortunately, **[FLP]!**
- ▶ **Fail-Safe FT:** *Safety* guaranteed but not *Liveness*.
 - ▶ Traffic lights are red.
 - ▶ Transactions in databases.

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed. Unfortunately, **[FLP]!**
- ▶ **Fail-Safe FT:** *Safety* guaranteed but not *Liveness*.
- ▶ **Non-Masking FT:** *Liveness* guaranteed but not *Safety*.

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed. Unfortunately, **[FLP]!**
- ▶ **Fail-Safe FT:** *Safety* guaranteed but not *Liveness*.
- ▶ **Non-Masking FT:** *Liveness* guaranteed but not *Safety*.
 - ▶ Traffic lights are flashing orange.
 - ▶ Optimistic algorithm: Data replication mechanisms.

Fault Tolerance

- ▶ **Masking FT:** Both *Safety* and *Liveness* must be guaranteed. Unfortunately, **[FLP]!**
- ▶ **Fail-Safe FT:** *Safety* guaranteed but not *Liveness*.
- ▶ **Non-Masking FT:** *Liveness* guaranteed but not *Safety*.
Self-Stabilization: *Safety* **eventually** guaranteed.

Dijkstra' self-stabilizing algorithms

- ▶ Token-passing on a ring
- ▶ Token-passing on a chain with 4 states/process

Self-stabilization

Hypothesis

Atomicity

Scheduling

Proof Techniques

Transfer Function

Convergence stairs

Conclusion

Atomicity

- ▶ An example of a “stabilizing” sequential program

```
int x = 0;
```

```
...
```

```
while( x == x ) {
```

```
    x = 0;
```

```
    // code assuming x equals 0
```

```
}
```

Atomicity

- An example of a “stabilizing” sequential program

```
int x = 0;
...
while( x == x ) {
    x = 0;
    // code assuming x equals 0
}
```

```
0  iconst_0
1  istore_1
2  goto 7
...
5  iconst_0
6  istore_1
7  iload_1
8  iload_1
9  if_icmpeq 5
```

Atomicity

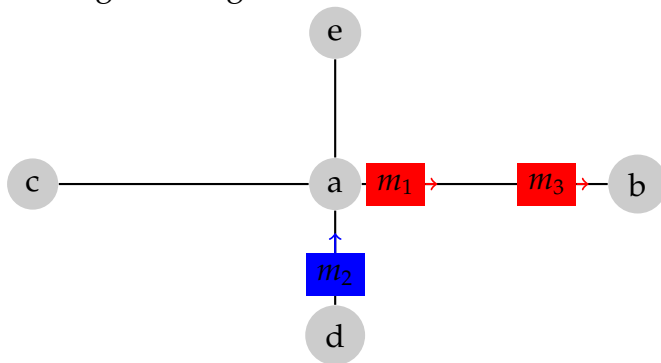
- An example of a “stabilizing” sequential program

```
int x = 0;
...
while( x == x ) {
    x = 0;
    // code assuming x equals 0
}
```

```
0  iconst_0
1  istore_1
2  goto 7
...
5  iconst_0
6  istore_1
7  iload_1
8  iload_1
9  if_icmpeq 5
```

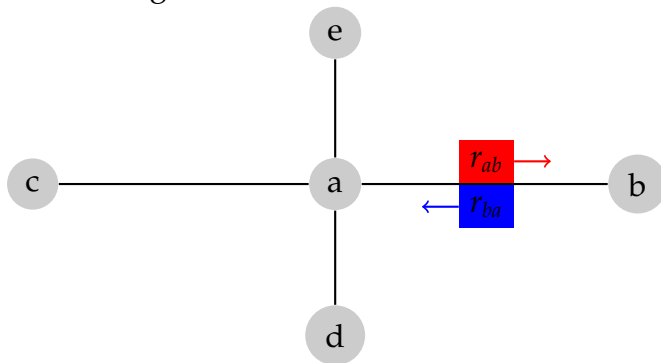

Communications

► Message Passing



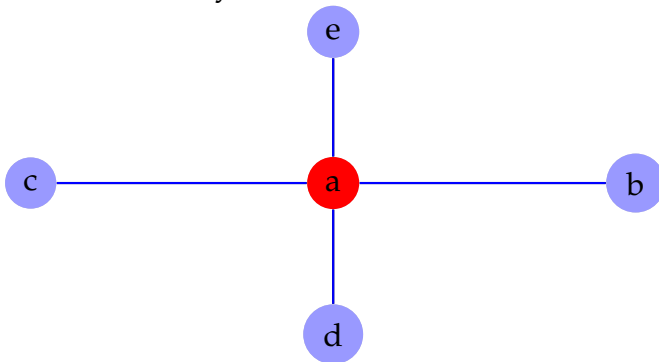
Communications

► Shared Registers



Communications

► Shared Memory



Communications

Message Passing

Shared Registers

Shared Memory

Communications

is more
difficult

to program
than

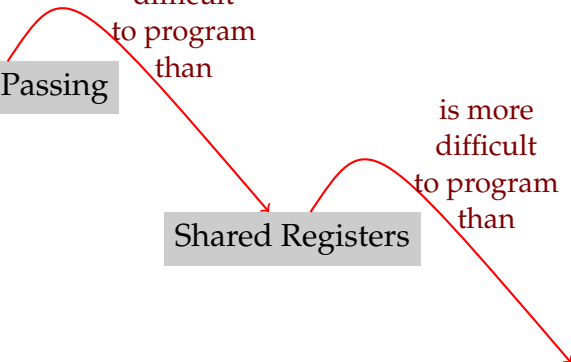
Message Passing

is more
difficult

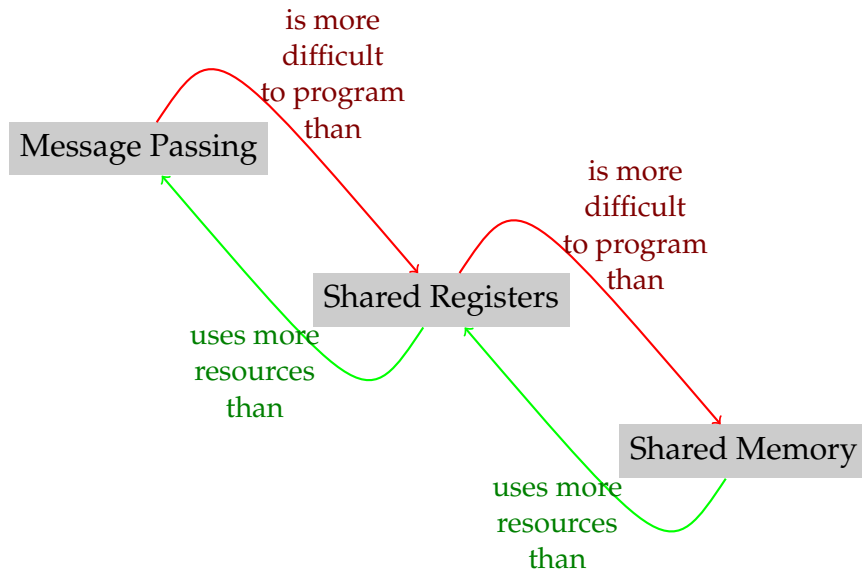
to program
than

Shared Registers

Shared Memory



Communications



Example

Definition (Shared Memory)

In one atomic step, read the states of all neighbors and write own state

Definition (Guarded command)

- ▶ Guard \rightarrow Action

Example

Definition (Shared Memory)

In one atomic step, read the states of all neighbors and write own state

Definition (Guarded command)

- ▶ **Guard** \rightarrow Action
- ▶ Guard: predicate on the states of the neighborhood

Example

Definition (Shared Memory)

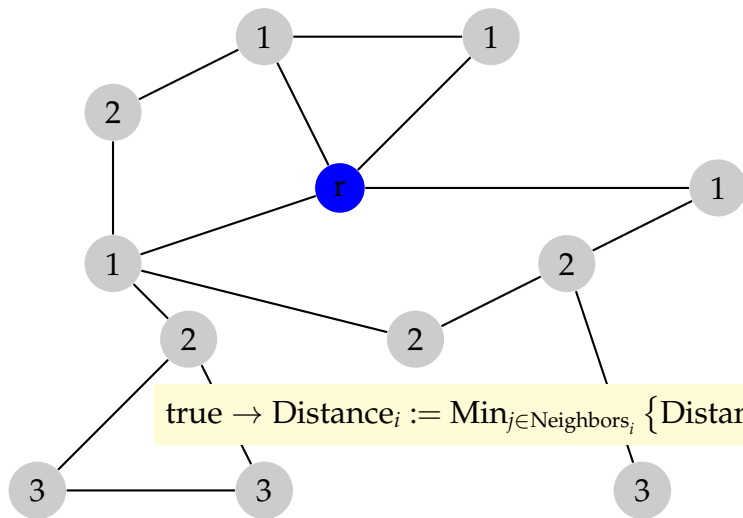
In one atomic step, read the states of all neighbors and write own state

Definition (Guarded command)

- ▶ Guard \rightarrow Action
- ▶ Guard: predicate on the states of the neighborhood
- ▶ Action: executed if *Guard* evaluates to true

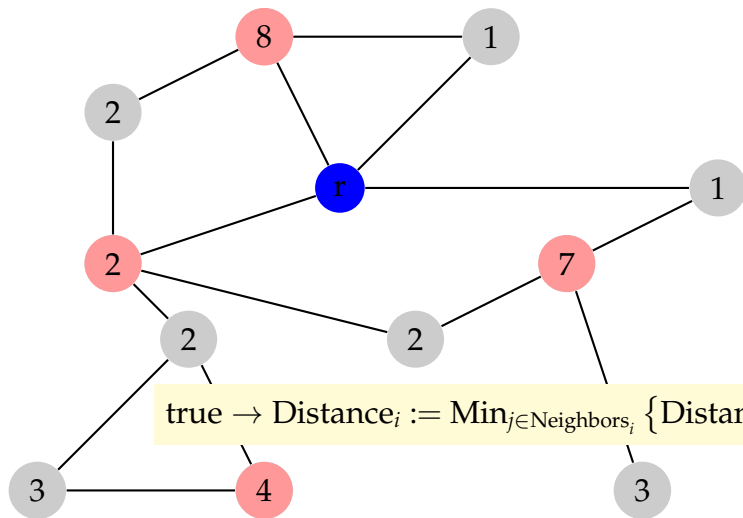
The graph has 11 nodes and 14 edges. The nodes are labeled as follows: a central blue node 'r', and ten gray nodes labeled with numbers 1, 2, or 3. The connections are as follows: Node 'r' is connected to four nodes (two labeled '1' and two labeled '2'). One node labeled '1' is connected to two nodes labeled '2'. One node labeled '2' is connected to two nodes labeled '1'. One node labeled '1' is connected to two nodes labeled '3'. One node labeled '2' is connected to two nodes labeled '3'. The remaining nodes and edges form a complex structure of interconnected cycles and paths.

Example



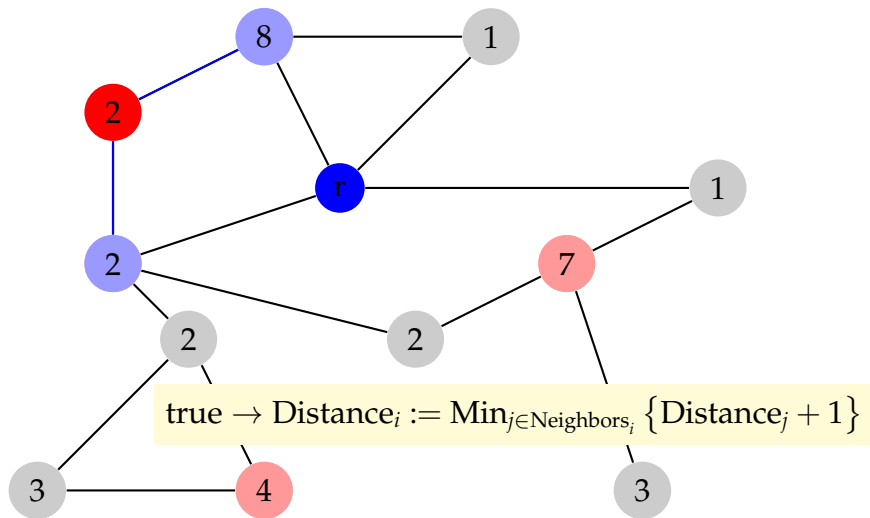
$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{ \text{Distance}_j + 1 \}$

Example

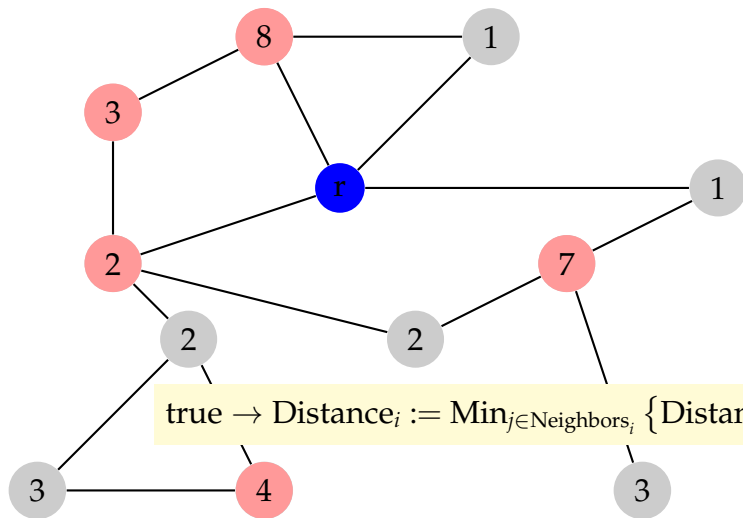


$\text{true} \rightarrow \text{Distance}_i := \text{Min}_{j \in \text{Neighbors}_i} \{ \text{Distance}_j + 1 \}$

Example

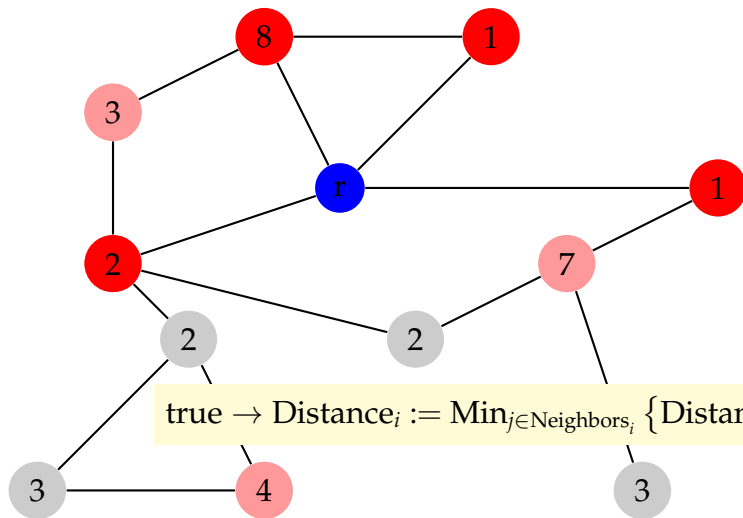


Example



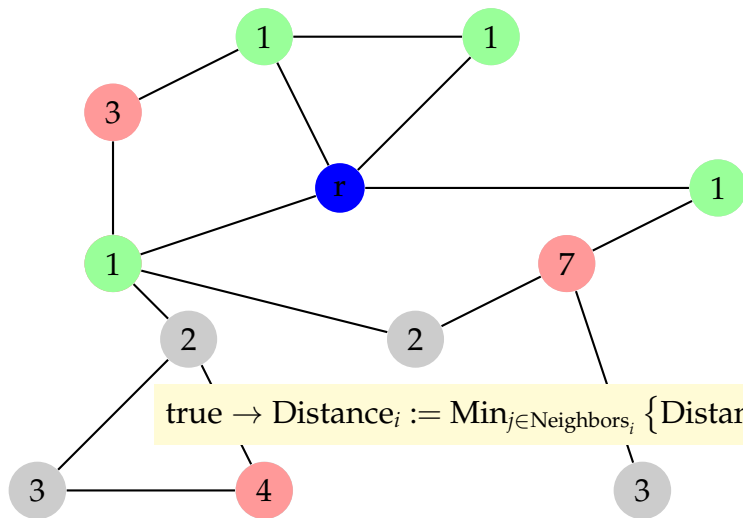
$\text{true} \rightarrow \text{Distance}_i := \min_{j \in \text{Neighbors}_i} \{ \text{Distance}_j + 1 \}$

Example

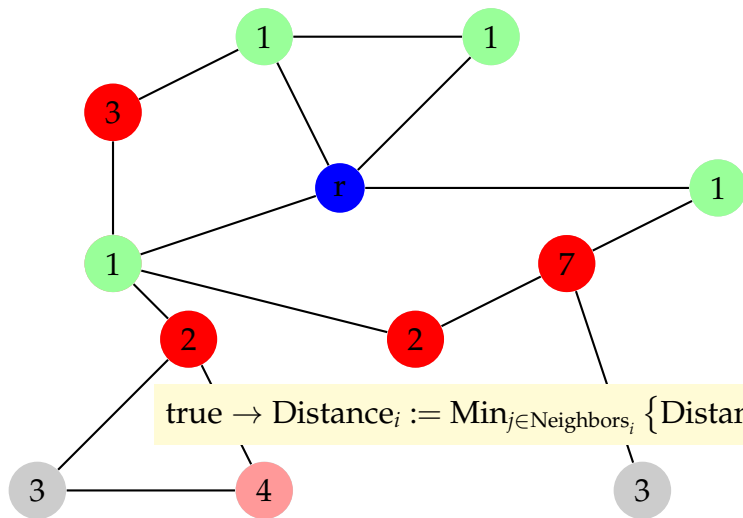


$\text{true} \rightarrow \text{Distance}_i := \min_{j \in \text{Neighbors}_i} \{ \text{Distance}_j + 1 \}$

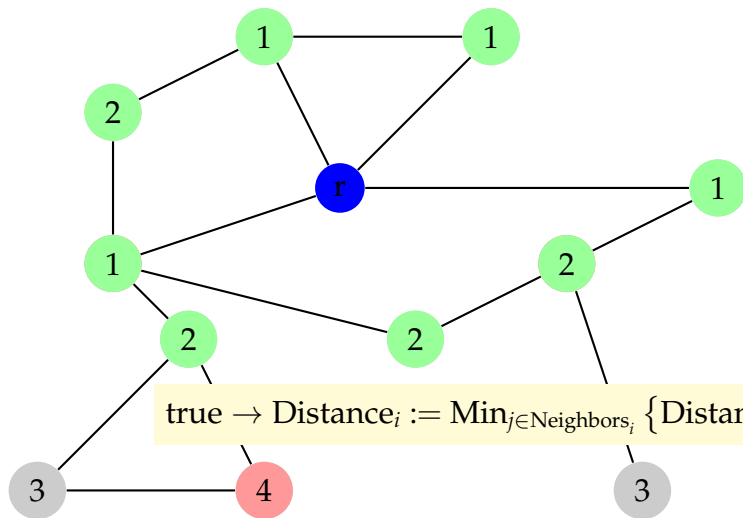
Example



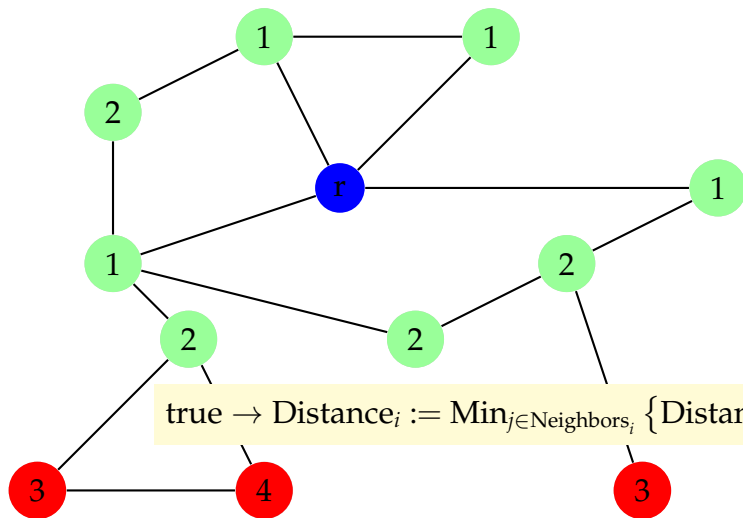
Example



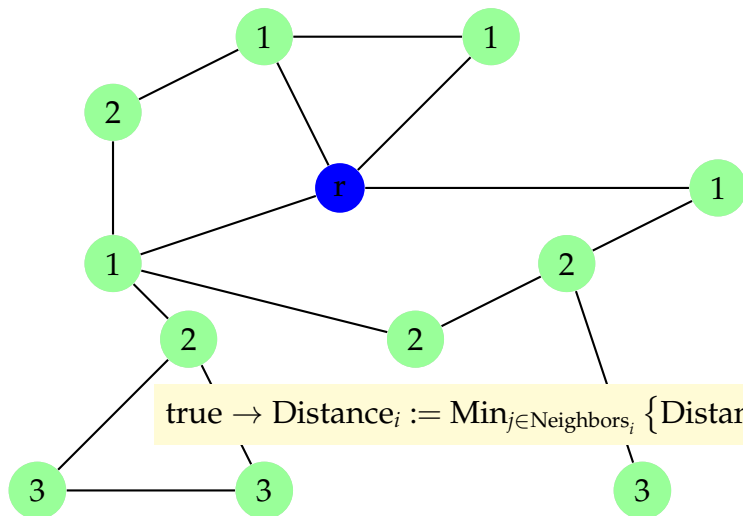
Example



Example



Example



true \rightarrow Distance_i := Min_{j ∈ Neighbors_i} { Distance_j + 1 }

Scheduling

Definition (Scheduler *a.k.a.* Daemon)

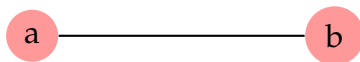
The daemon chooses among activable processors those that will execute their actions.

- ▶ The **daemon** can be seen as an adversary whose role is to prevent stabilization

Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j \mid j \in \text{Neighbors}_i \} \}$$

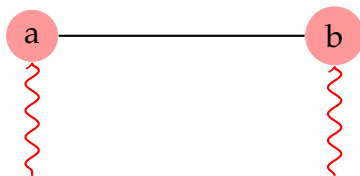
$$\Delta = \{ \textcolor{red}{0}, \textcolor{blue}{1} \}$$



Spatial Scheduling

$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j | j \in \text{Neighbors}_i \} \}$

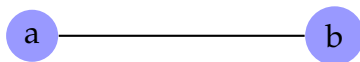
$$\Delta = \{ \text{0}, \text{1} \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j \mid j \in \text{Neighbors}_i \} \}$$

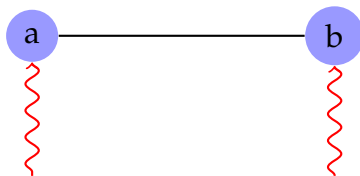
$$\Delta = \{ \textcolor{red}{0}, \textcolor{blue}{1} \}$$



Spatial Scheduling

$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j | j \in \text{Neighbors}_i \} \}$

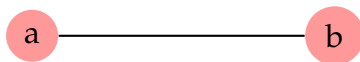
$$\Delta = \{ \text{0}, \text{1} \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j \mid j \in \text{Neighbors}_i \} \}$$

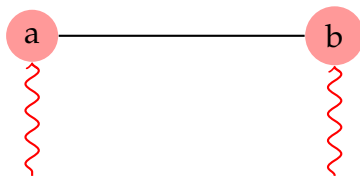
$$\Delta = \{ \textcolor{red}{0}, \textcolor{blue}{1} \}$$



Spatial Scheduling

$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j | j \in \text{Neighbors}_i \} \}$

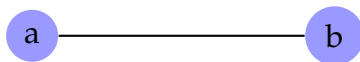
$$\Delta = \{ \text{0}, \text{1} \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j \mid j \in \text{Neighbors}_i \} \}$$

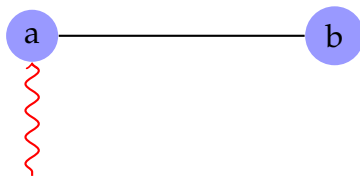
$$\Delta = \{ \textcolor{red}{0}, \textcolor{blue}{1} \}$$



Spatial Scheduling

$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j | j \in \text{Neighbors}_i \} \}$

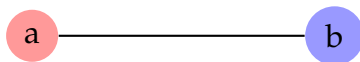
$$\Delta = \{ \text{0}, \text{1} \}$$



Spatial Scheduling

$$\text{true} \rightarrow \text{color}_i := \text{Min} \{ \Delta \setminus \{ \text{color}_j \mid j \in \text{Neighbors}_i \} \}$$

$$\Delta = \{ \textcolor{red}{0}, \textcolor{blue}{1} \}$$



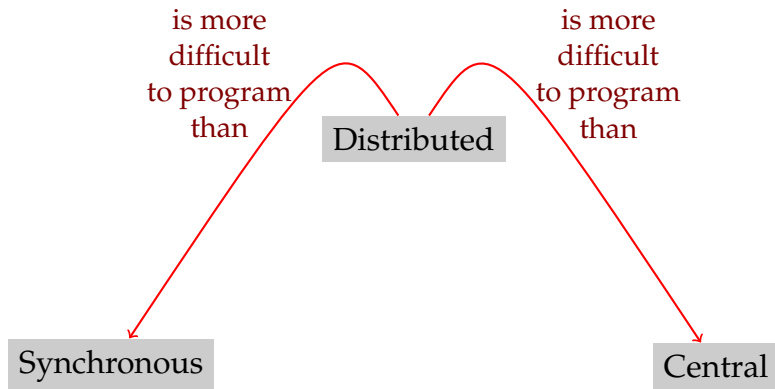
Spatial Scheduling

Distributed

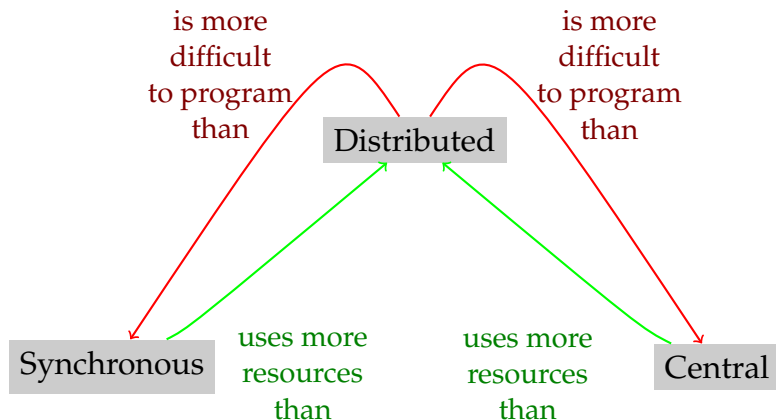
Synchronous

Central

Spatial Scheduling

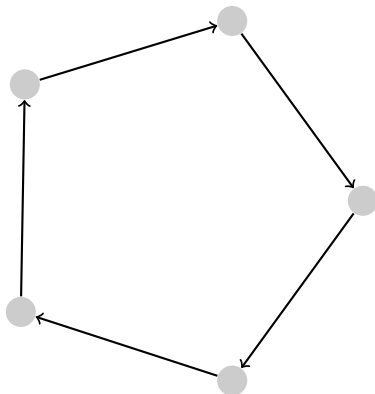


Spatial Scheduling



Temporal Scheduling

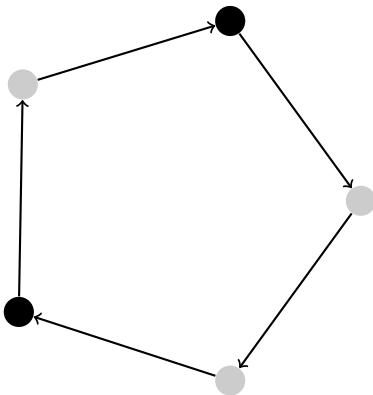
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

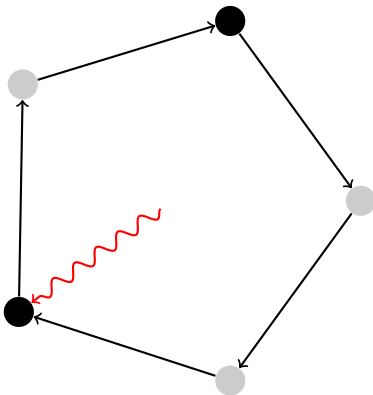
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

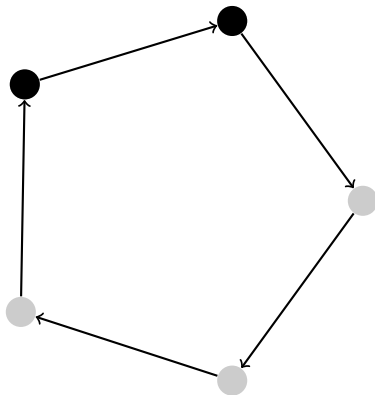
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

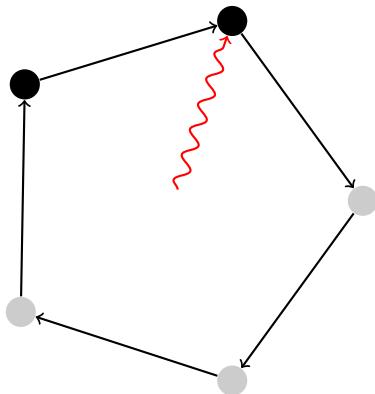
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

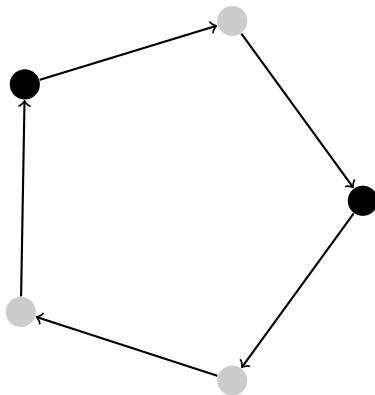
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

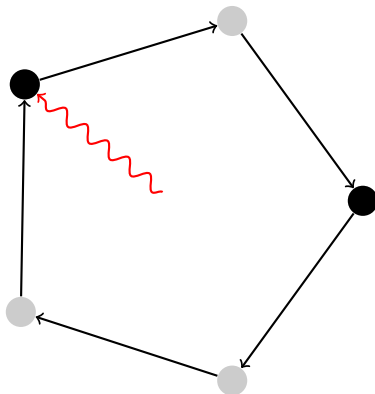
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

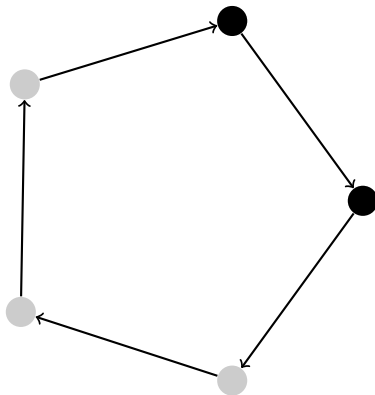
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

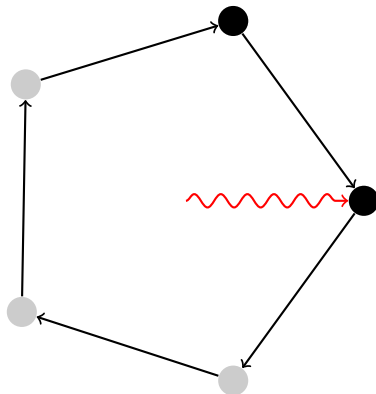
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

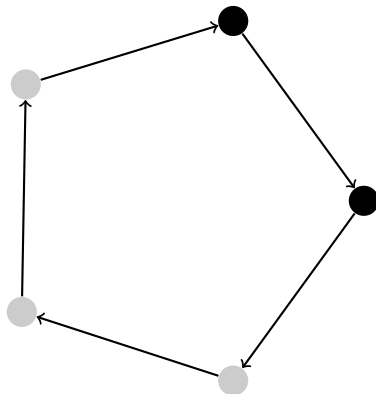
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

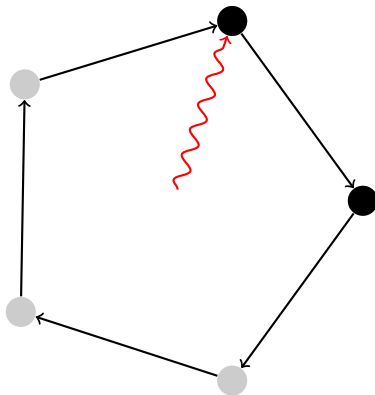
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

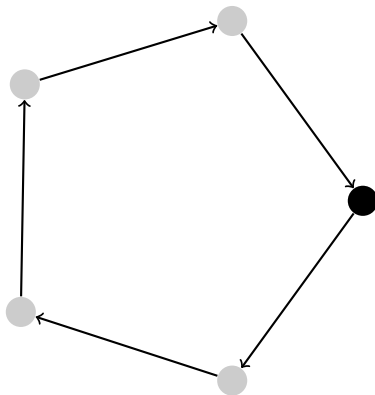
token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

token \rightarrow pass token to left neighbor with probability $\frac{1}{2}$



● = token

Temporal Scheduling

Unfair

Fair

Bounded

Temporal Scheduling

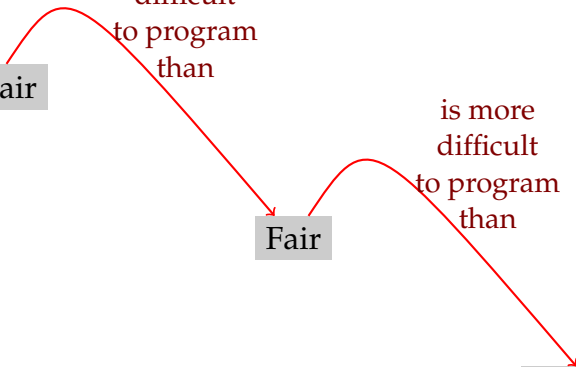
is more
difficult
to program
than

Unfair

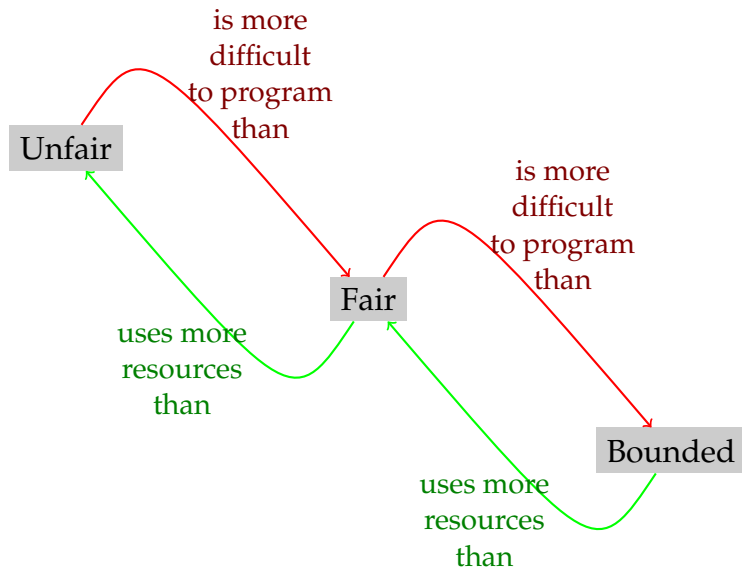
is more
difficult
to program
than

Fair

Bounded



Temporal Scheduling



Self-stabilization

Hypothesis

Atomicity

Scheduling

Proof Techniques

Transfer Function

Convergence stairs

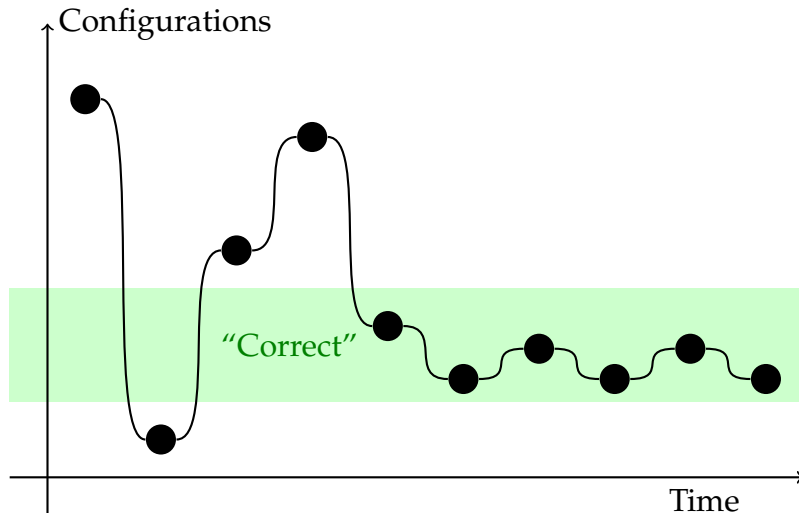
Conclusion

Transfer Function

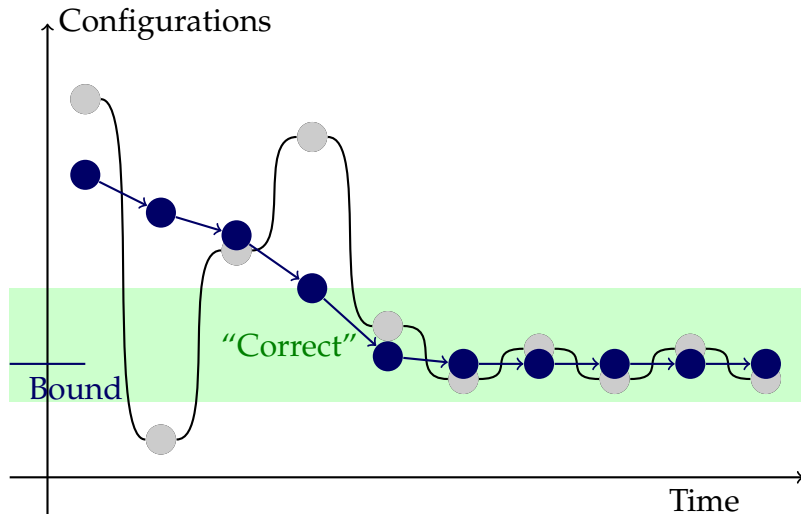
Basic Idea

- ▶ $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4 \rightarrow \dots \rightarrow c_i$
- ▶ $FP(c_1) > FP(c_2) > FP(c_3) > \dots > FP(c_i) = \text{bound}$
- ▶ Used to prove convergence
- ▶ Can be used to compute the number of steps to reach a legitimate configuration

Transfer Function

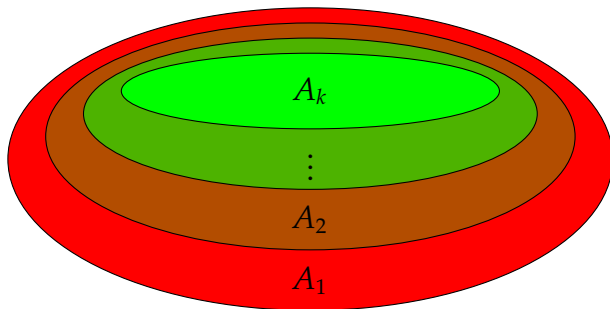


Transfer Function



Convergence stairs

- ▶ A_i is a predicate
- ▶ A_k is legitimate
- ▶ For any i between 1 and k , A_{i+1} is a refinement of A_i



Self-stabilization

Hypothesis

Atomicity

Scheduling

Proof Techniques

Transfer Function

Convergence stairs

Conclusion

Self-stabilization

Pros

- ▶ The network does not need to be initialized
- ▶ When a fault is diagnosed, it is sufficient to identify, then remove or restart the faulty components
- ▶ The self-stabilization property does not depend on the nature of the fault
- ▶ The self-stabilization property does not depend on the extent of the fault

Self-stabilization

Cons

- ▶ *A priori*, “eventually” does not give any bound on the stabilization time
- ▶ *A priori*, nodes never know whether the system is stabilized or not
- ▶ A single failure may trigger a correcting action at every node in the network
- ▶ Faults must be sufficiently rare that they can be considered are transient

■ Initial State

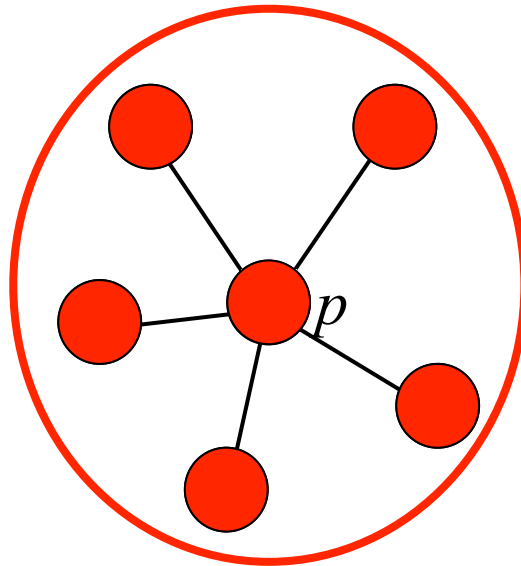
- ① Incorrect messages in channels
- ② Incorrect values in variables

*Starting from **any initial configuration**, the system eventually reaches a configuration from which its behavior is correct.*

Dijkstra 1974

State Model

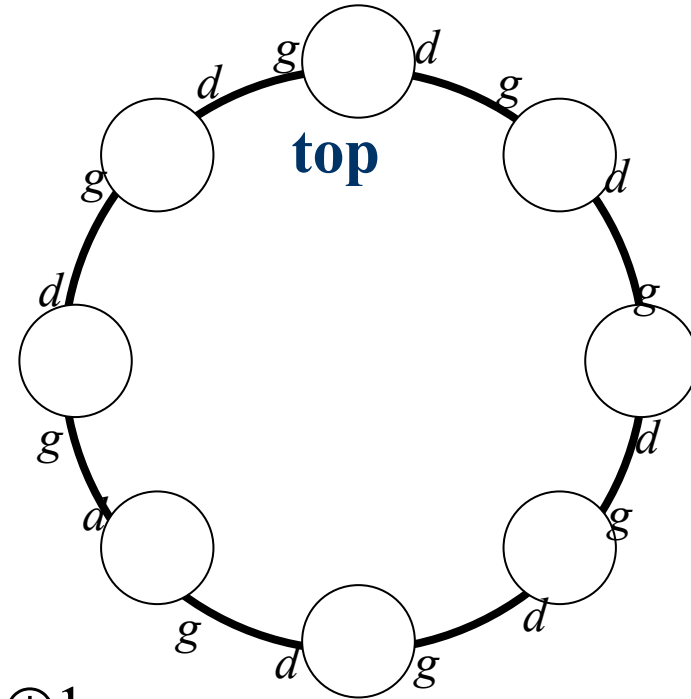
<Gard> → <Action>



Anneau à jeton

Spécification

- Sûreté : *Au plus un jeton dans le système*
- Vivacité : *Chaque processeur obtient le jeton infiniment souvent*



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

$p \neq top$

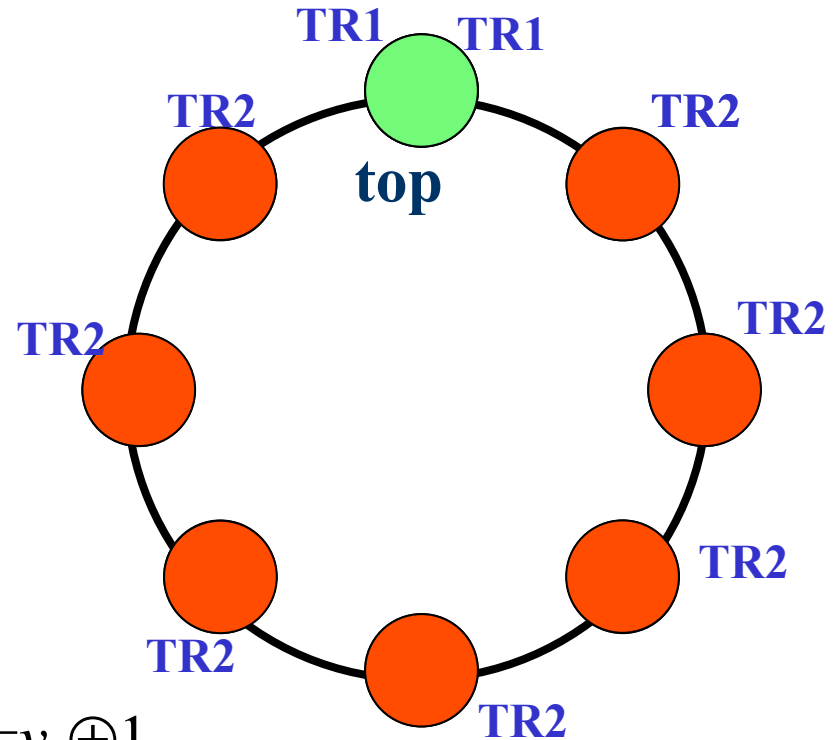
TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Auto-stabilisant ?

Specification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

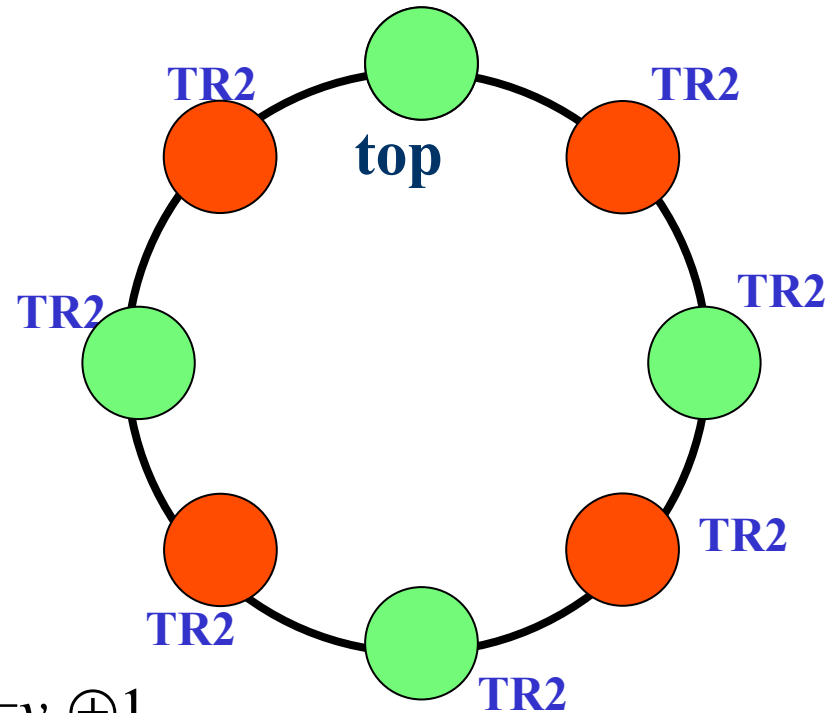
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

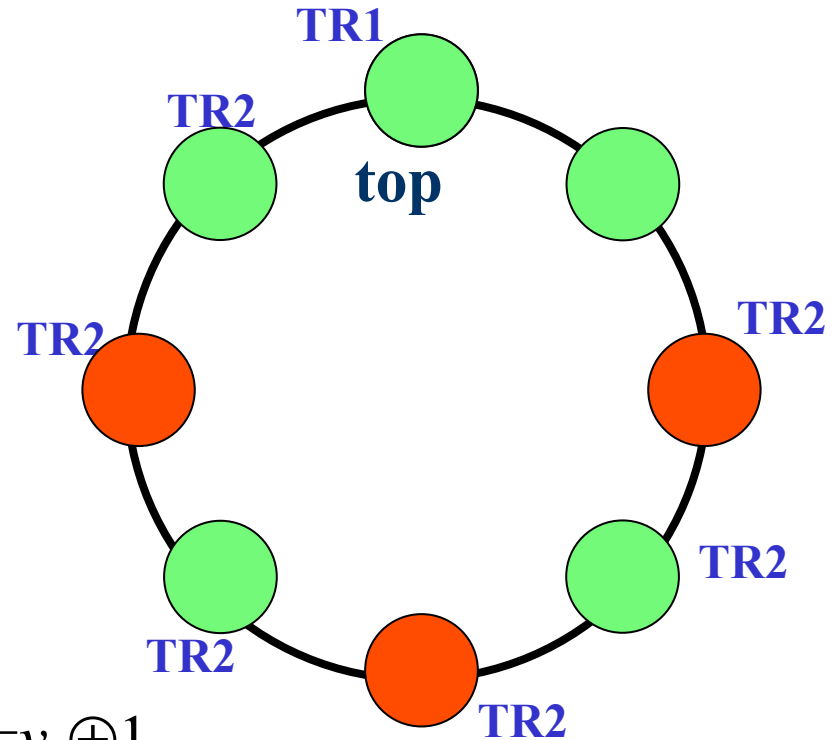
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

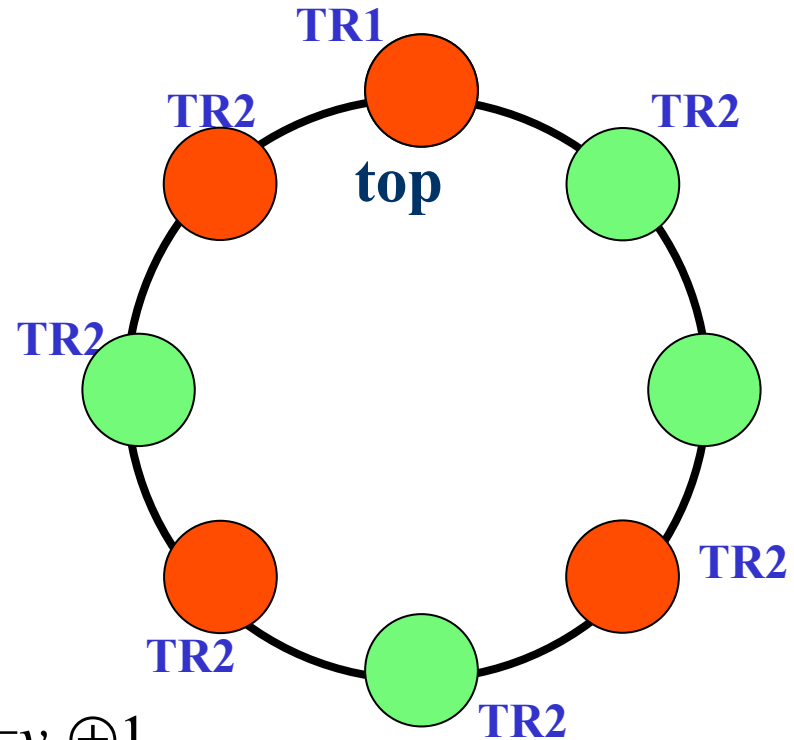
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

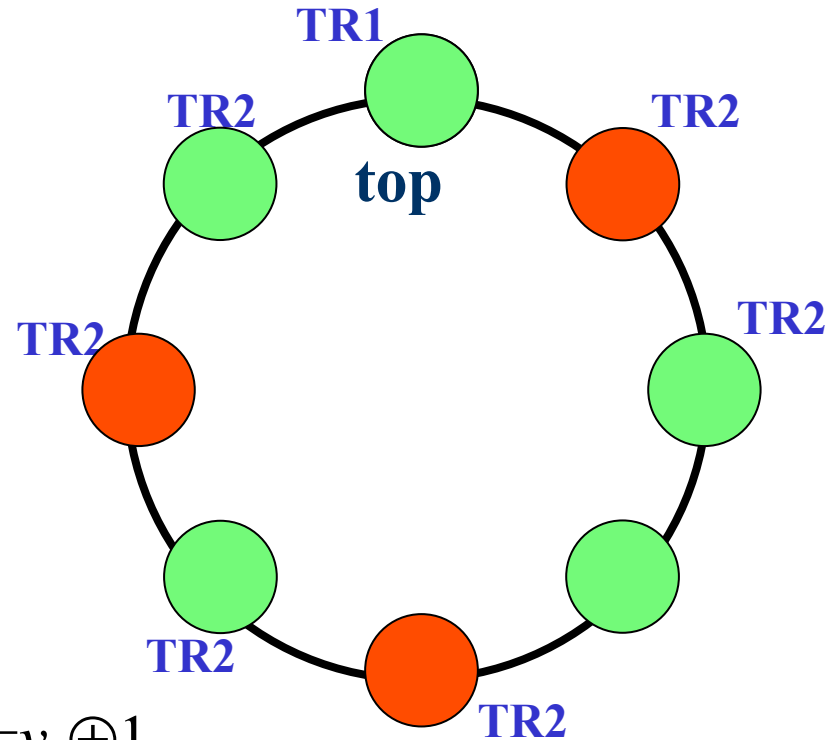
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

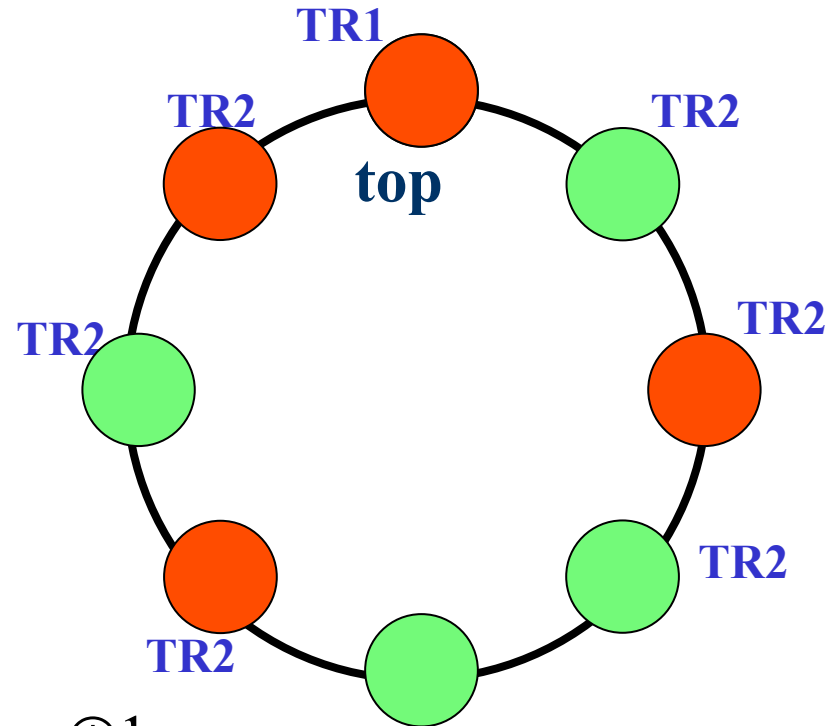
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

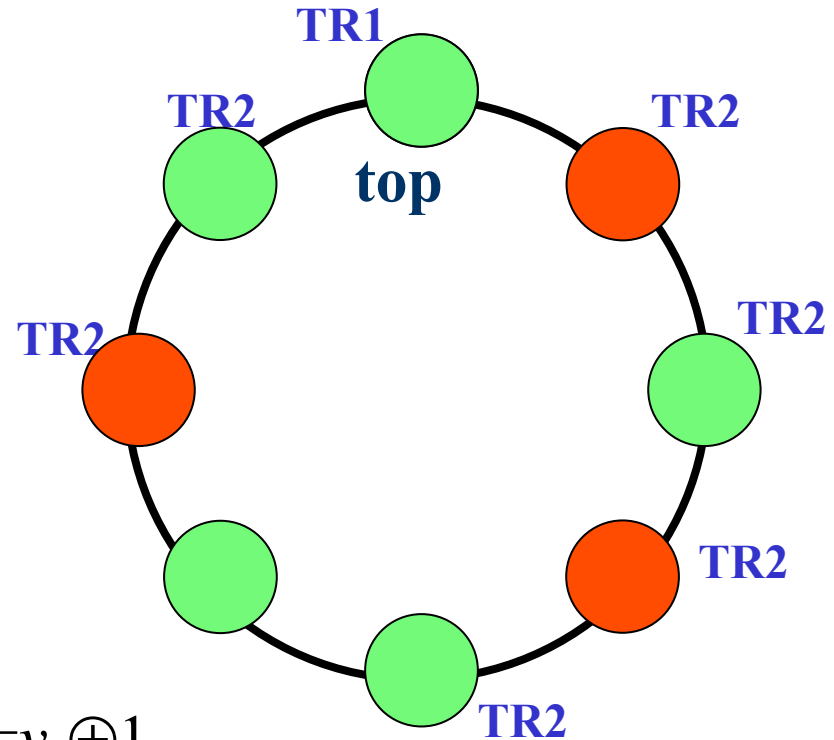
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

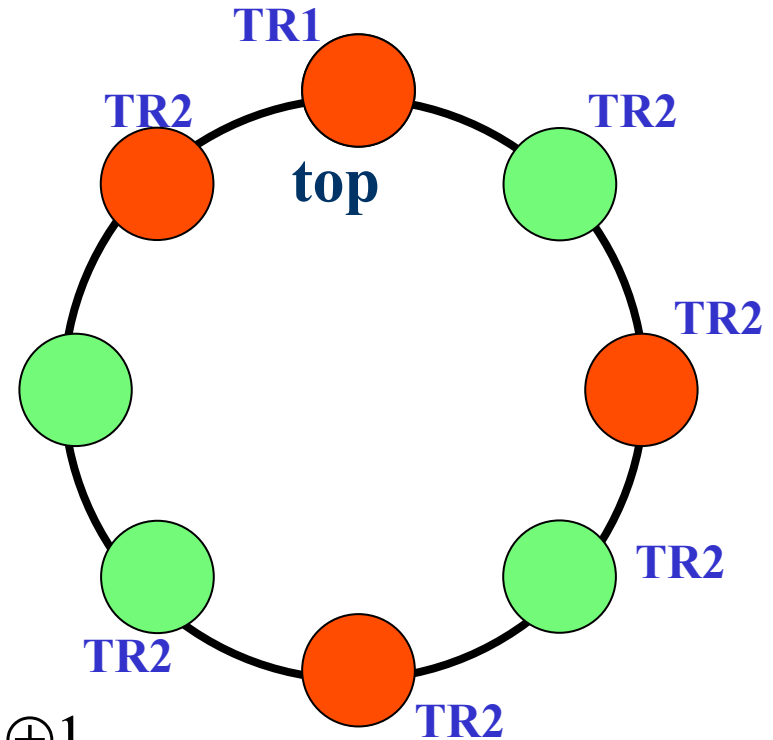
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus 1$

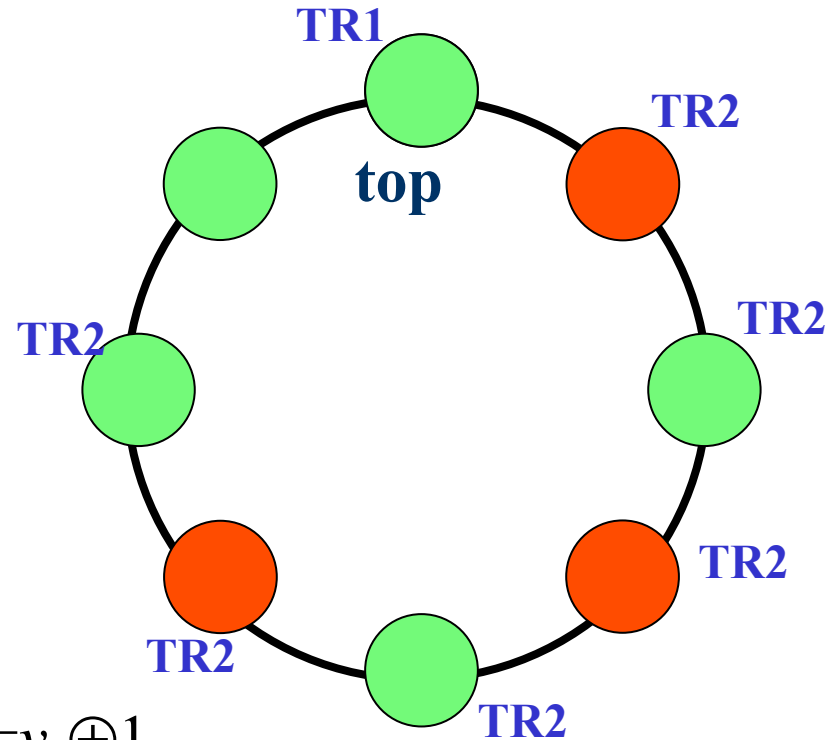
$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

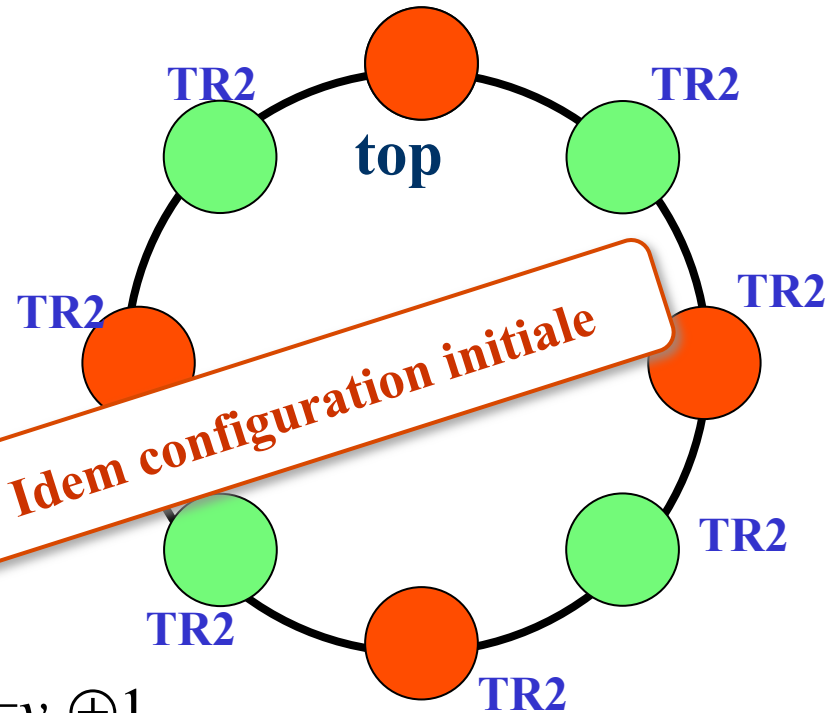
$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : Au plus un jeton dans le système
- Vivacité : Chaque processeur obtient le jeton infiniment souvent



$p = top$

TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Spécification

- Sûreté : *Au plus un jeton dans le système*
- Vivacité : *Chaque processeur obtient le jeton infiniment souvent*

Exercice : Montrer qu'il se produit la même chose avec un nombre impair de processeurs.

$p = top$

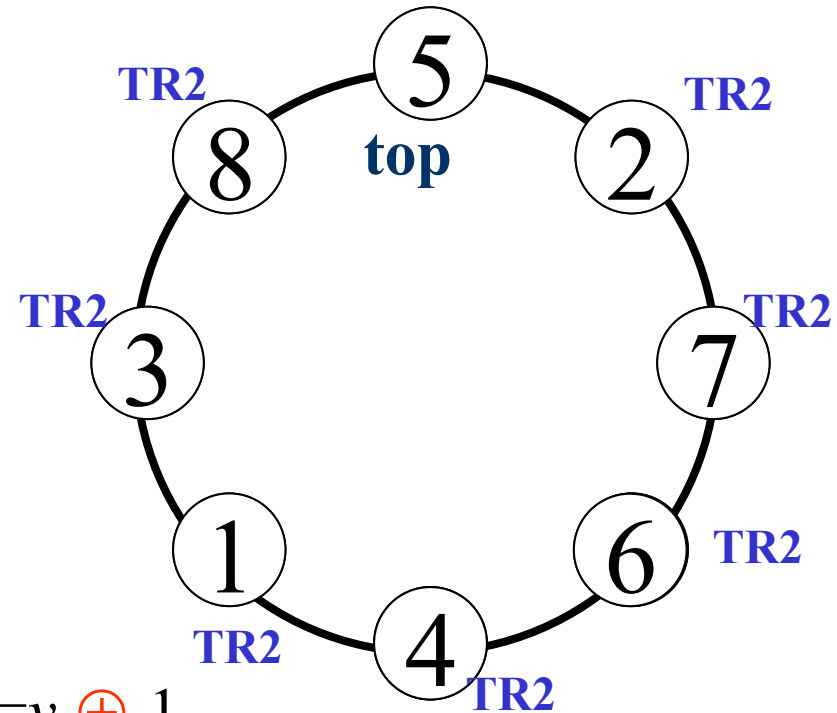
TR1 : $(v_p = v_g) \rightarrow v_p := v_g \oplus 1$

$p \neq top$

TR2 : $(v_p \neq v_g) \rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

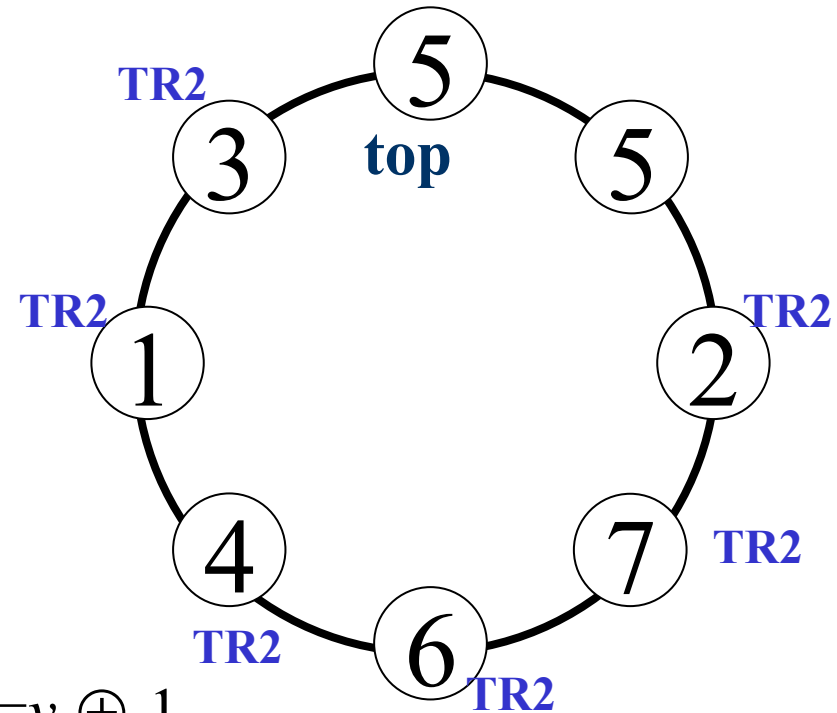
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

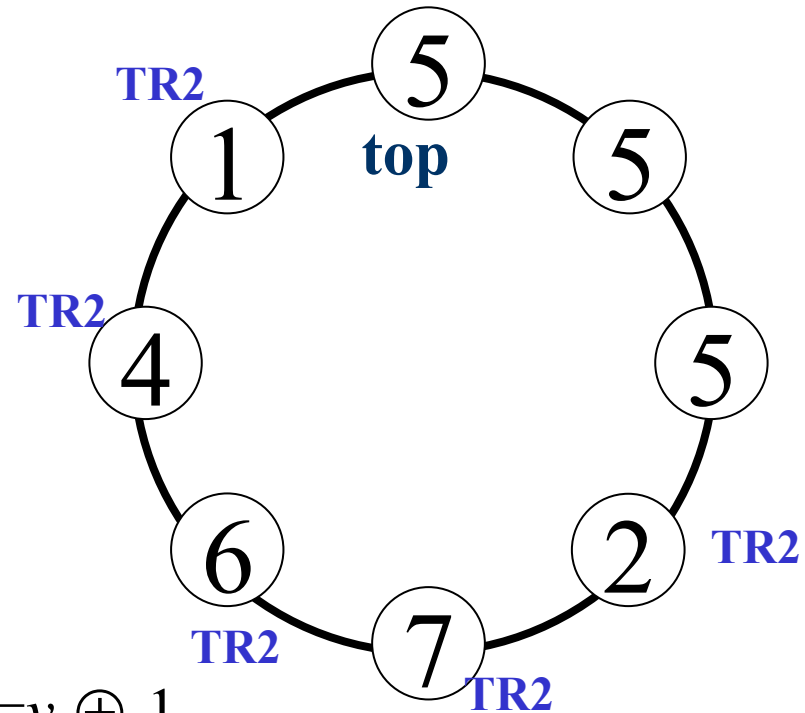
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

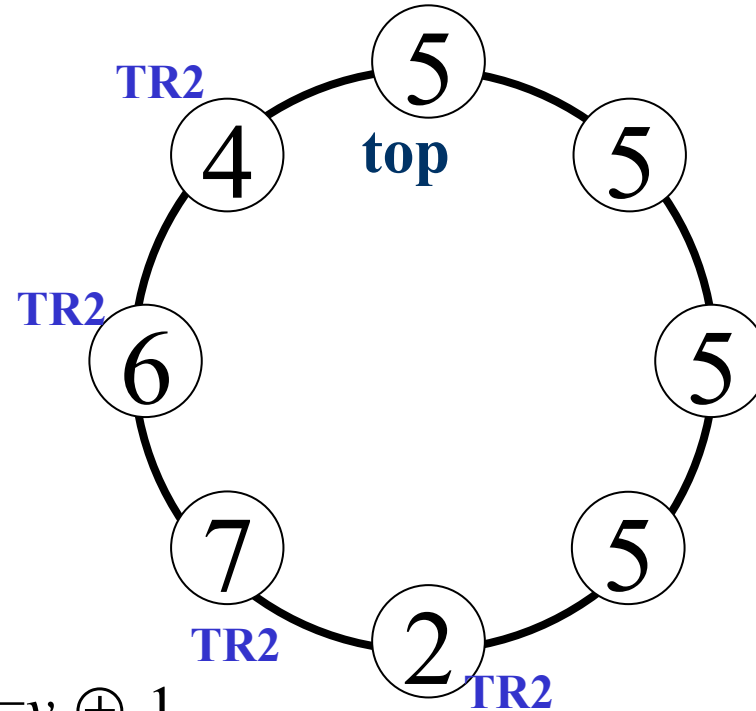
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

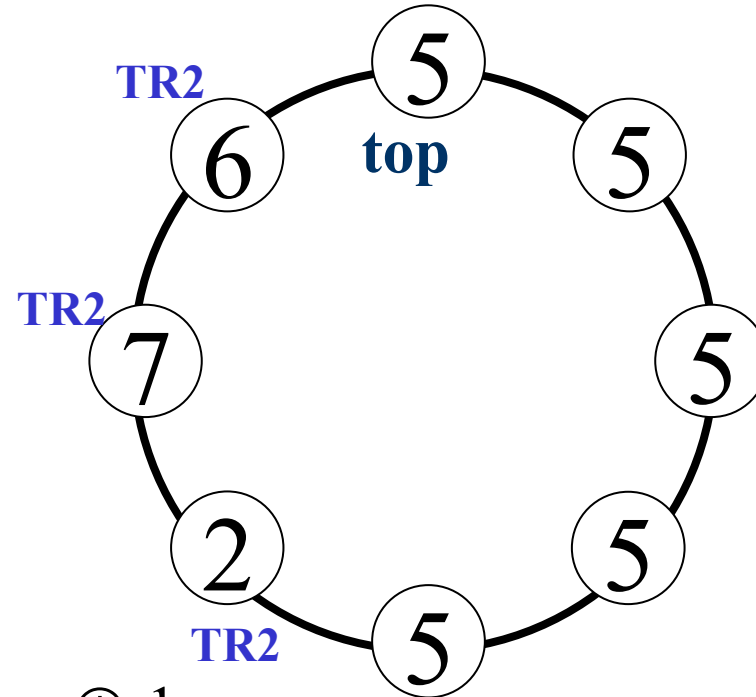
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

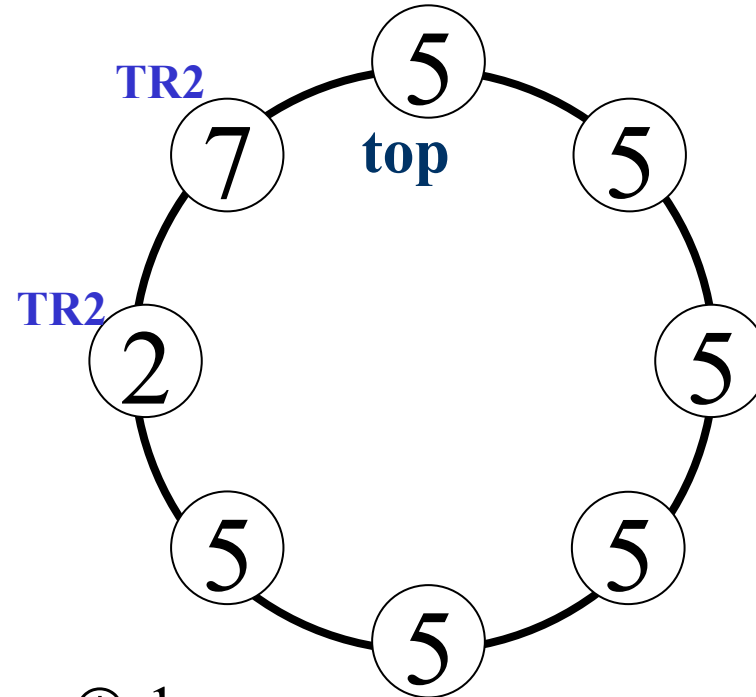
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

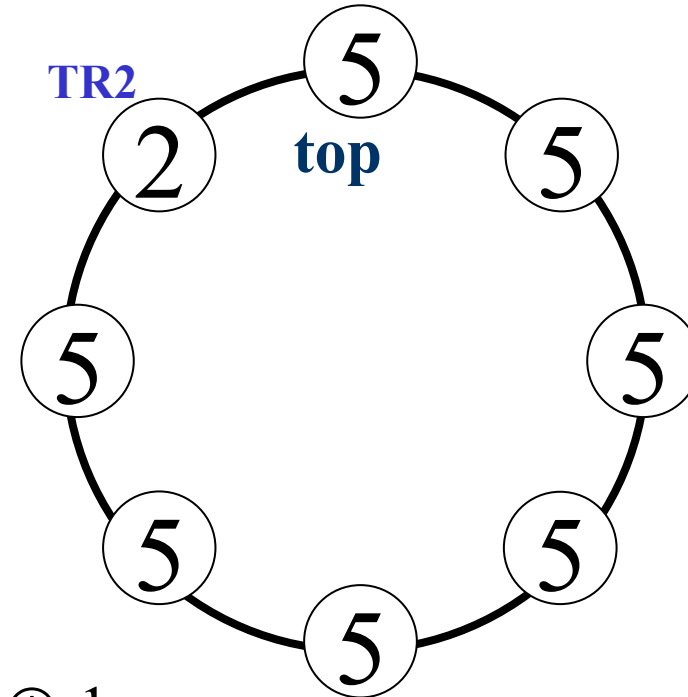
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

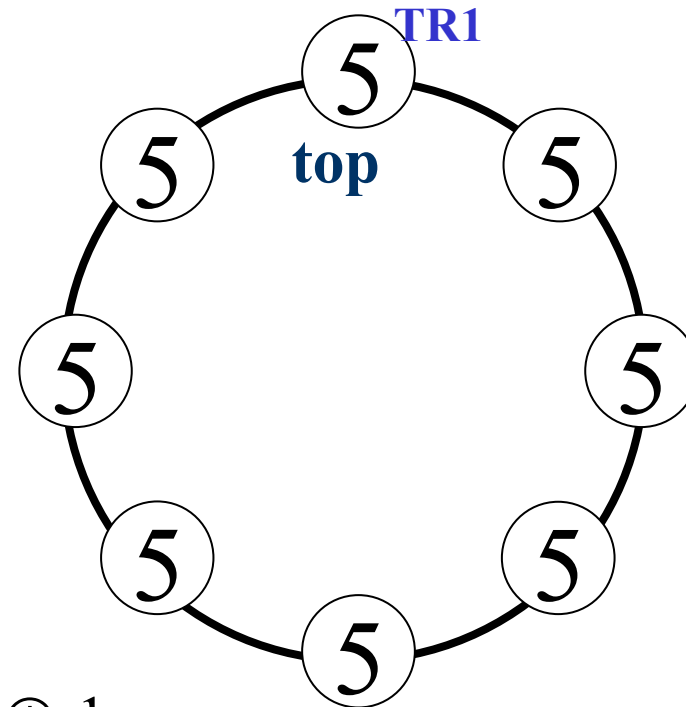
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

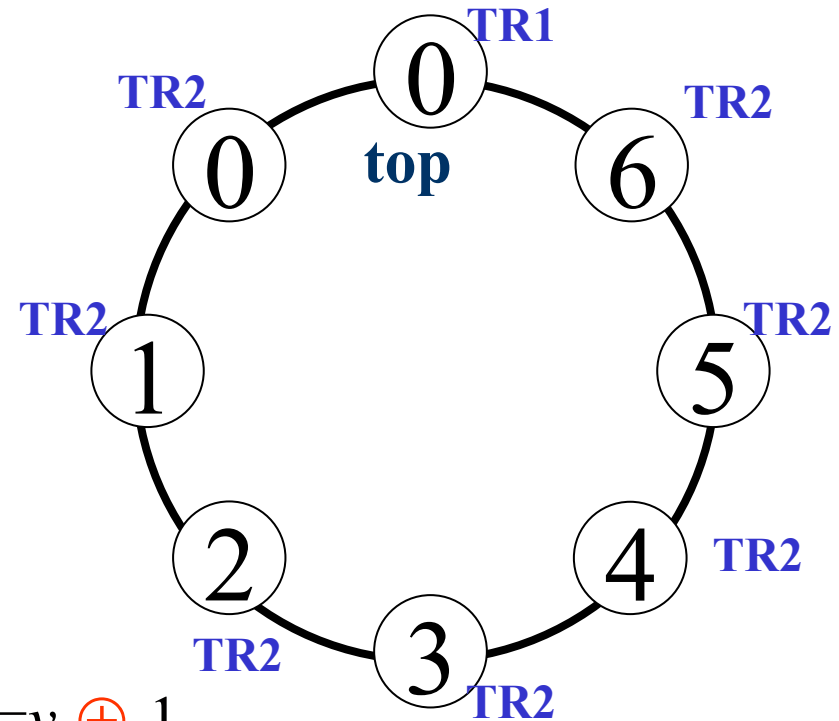
$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Valeur de k ?

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

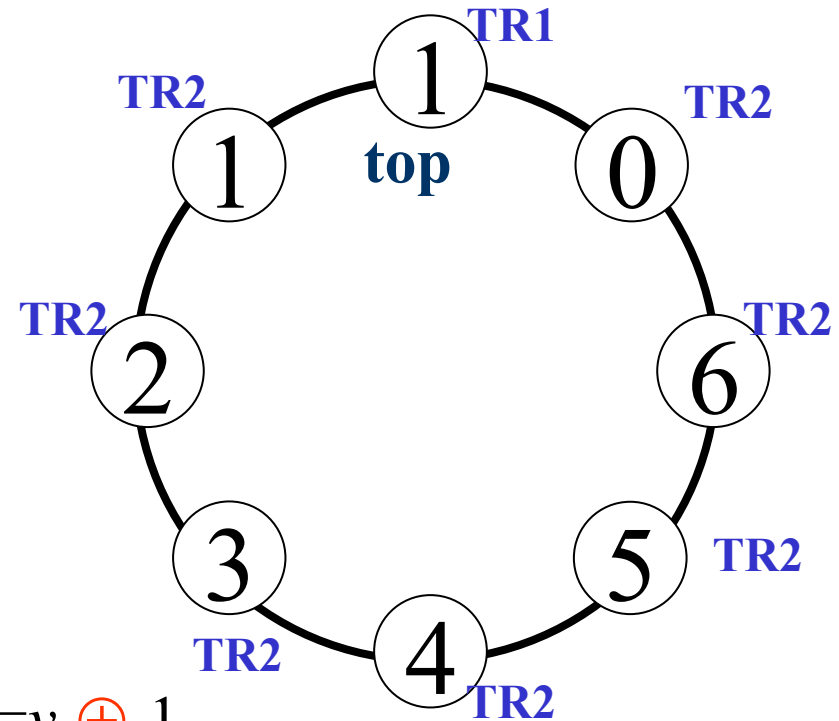
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

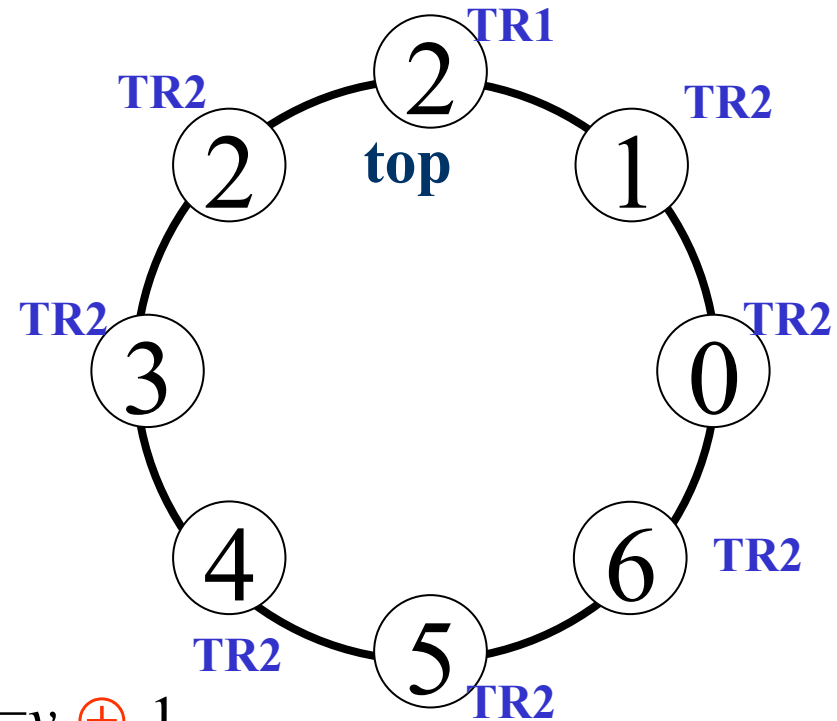
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

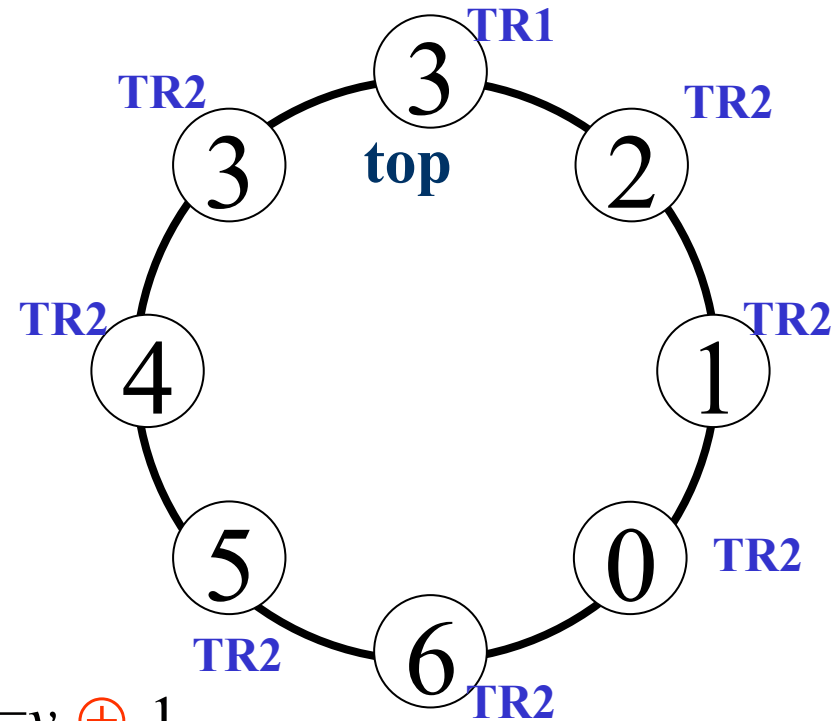
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

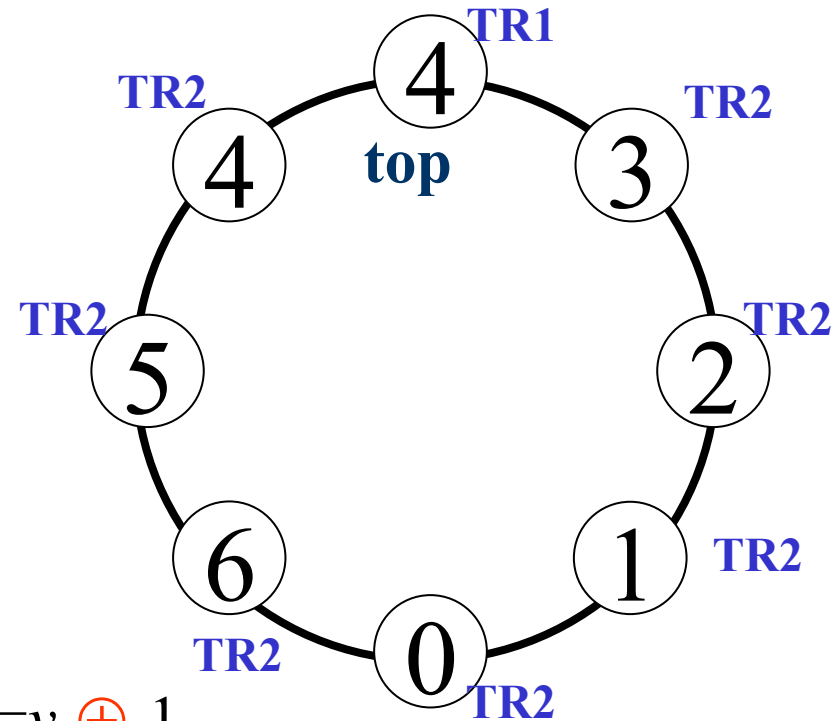
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

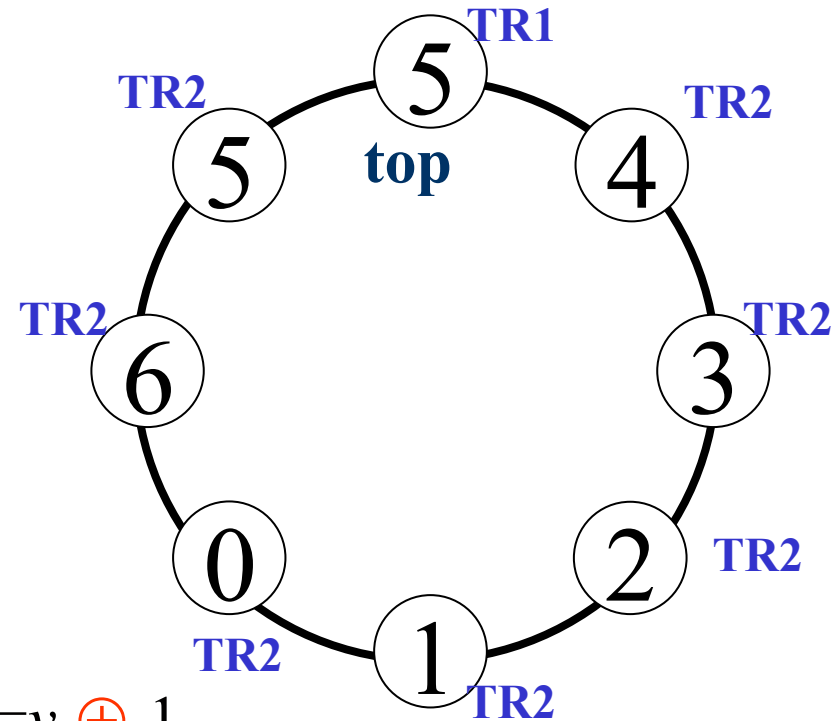
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

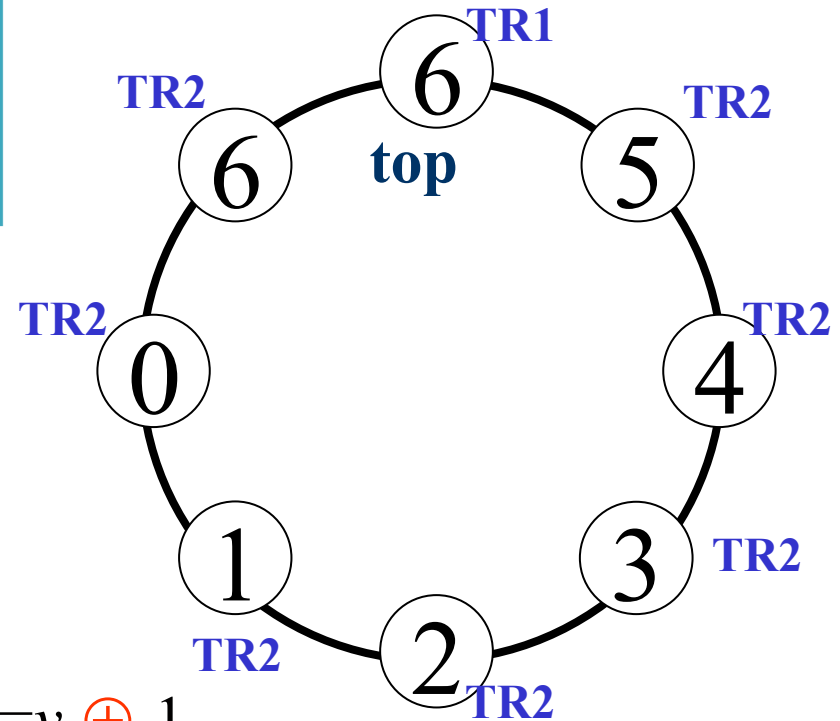
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

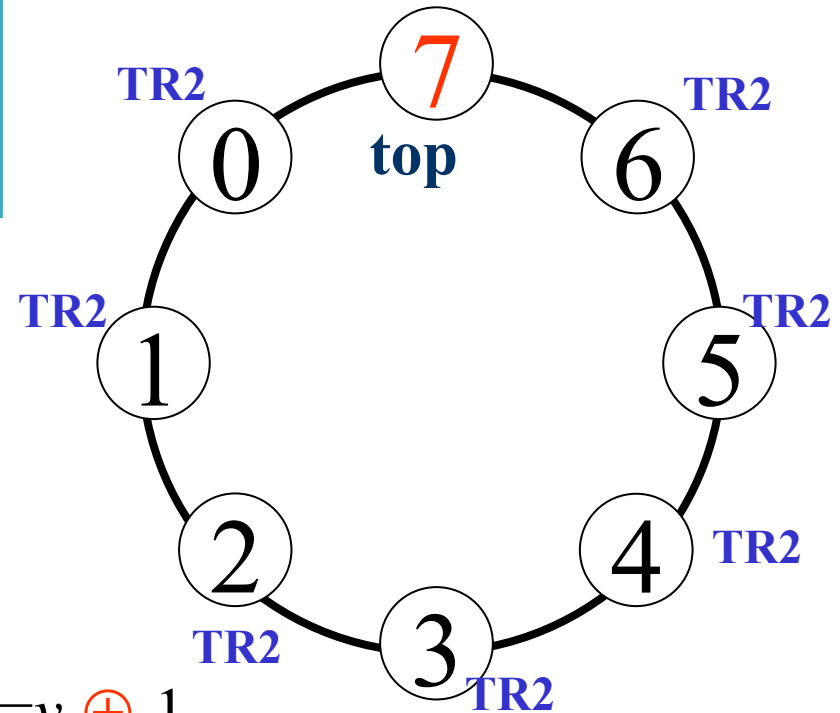
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

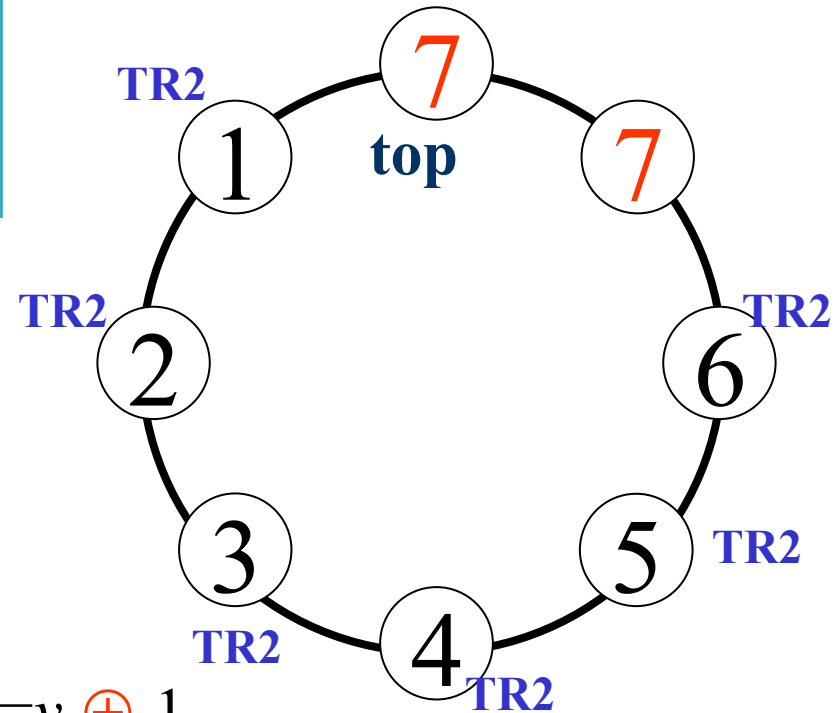
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

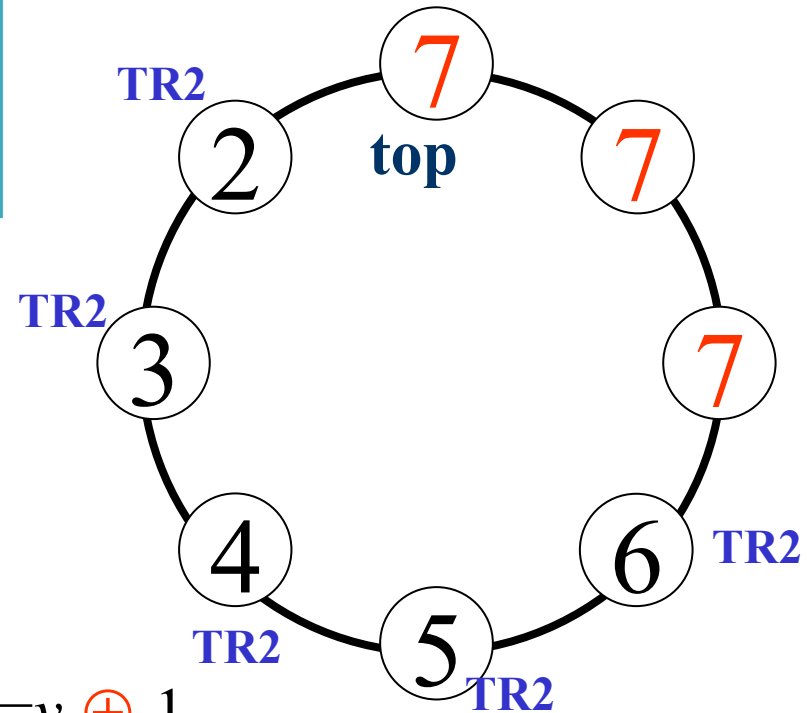
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

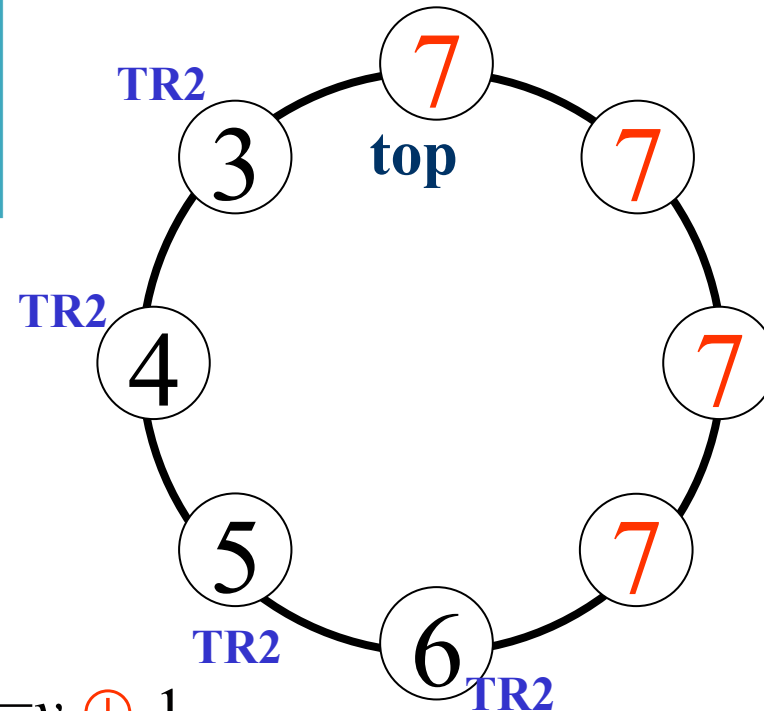
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

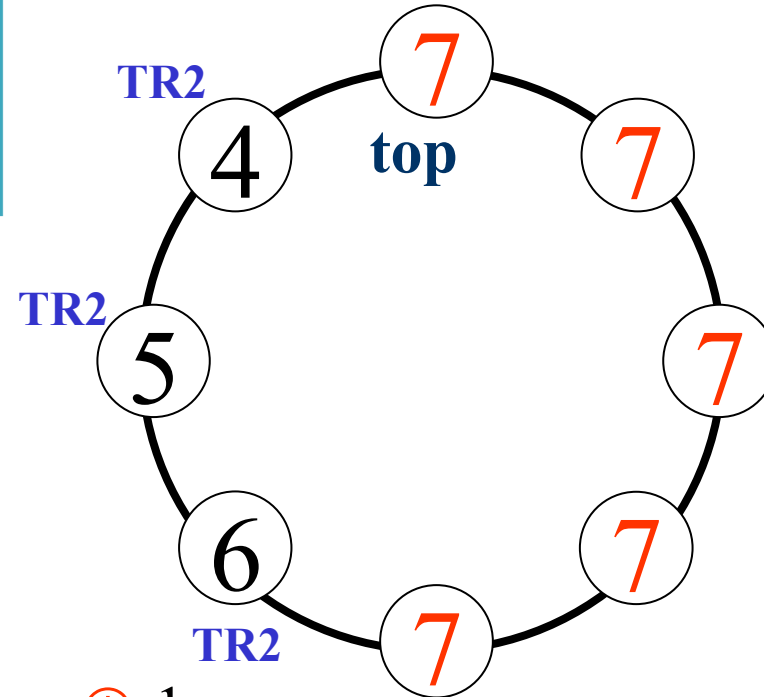
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

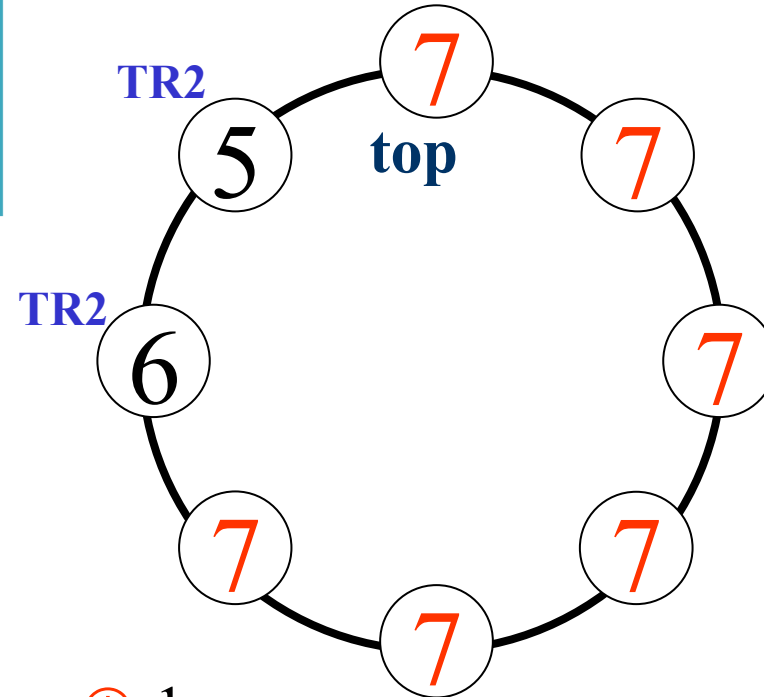
TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

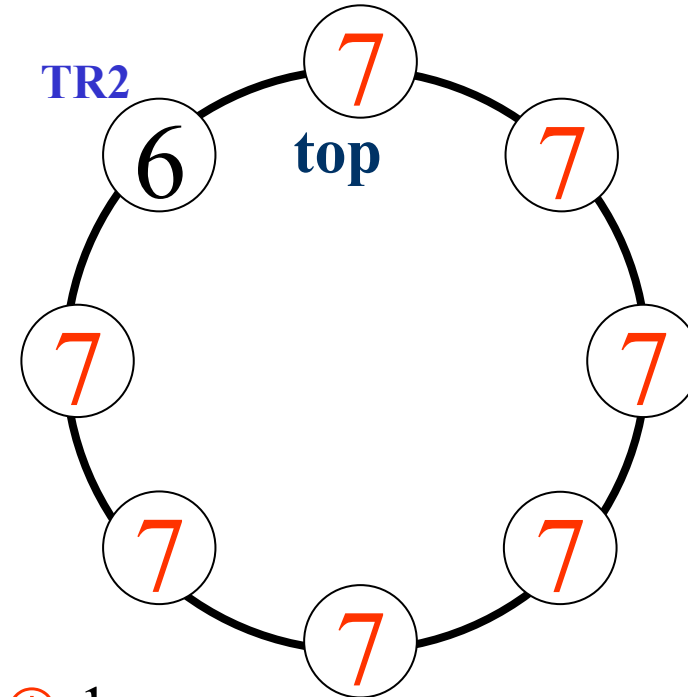
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

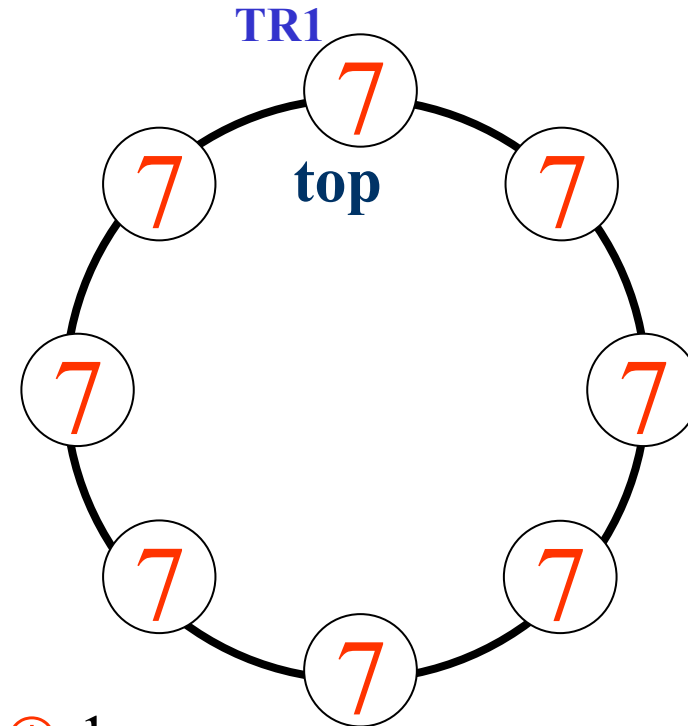
TR1 : $(v_p = v_g)$ \rightarrow $v_p := v_g \oplus_k 1$

$p \neq top$

TR2 : $(v_p \neq v_g)$ \rightarrow $v_p := v_g$

Anneau à jeton

Algorithme Auto-stabilisant de circulation de jeton de Dijkstra



$p = top$

TR1 : $(v_p = v_g)$ $\rightarrow v_p := v_g \oplus_k 1$

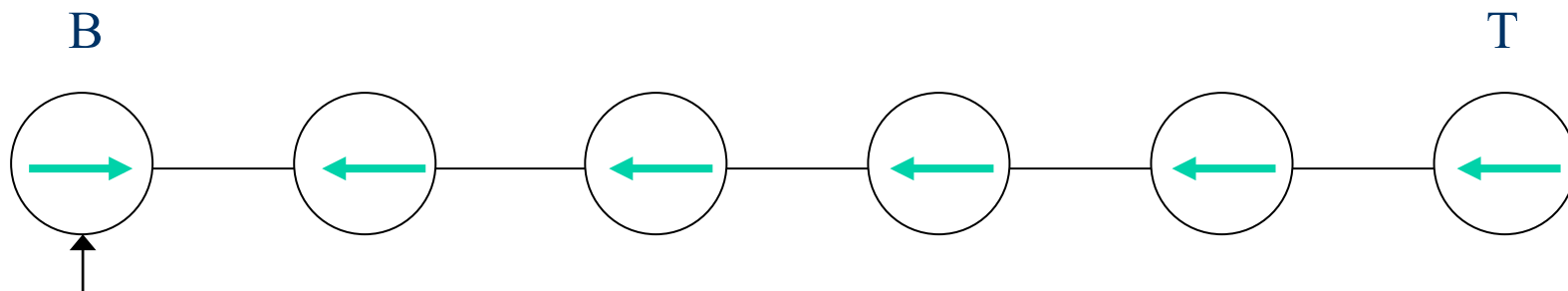
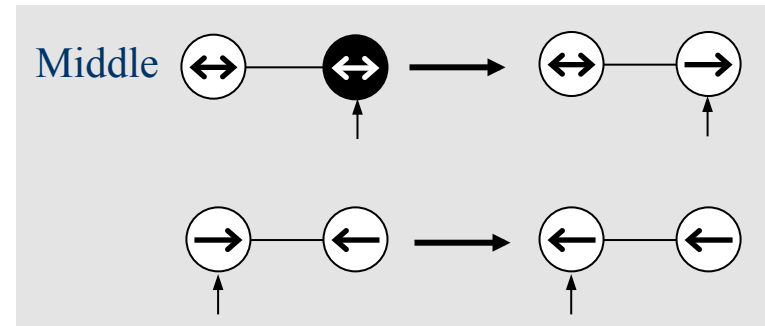
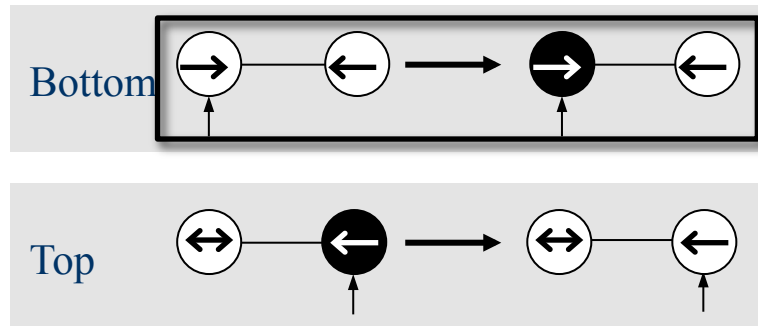
$p \neq top$

TR2 : $(v_p \neq v_g)$ $\rightarrow v_p := v_g$

Temps de Stabilisation : $O(n)$

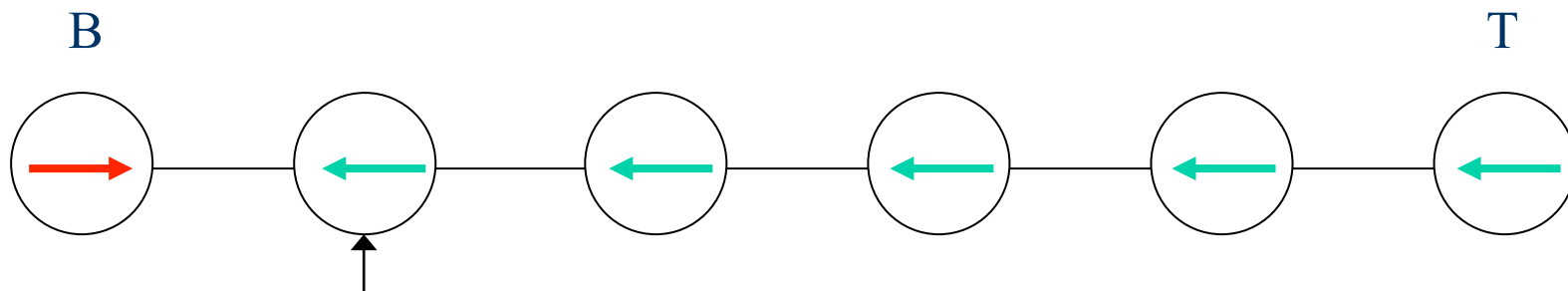
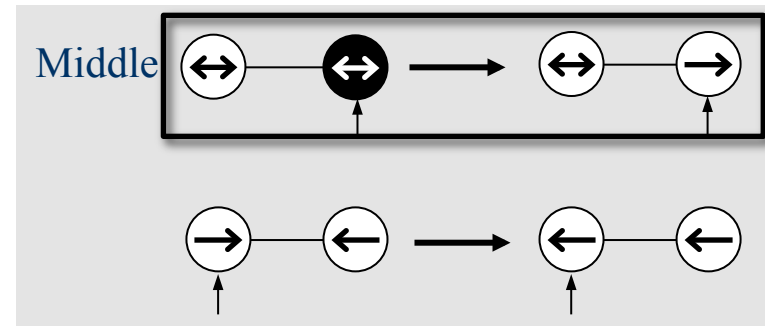
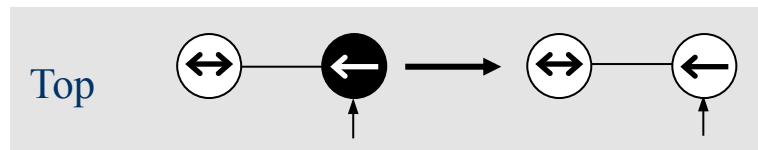
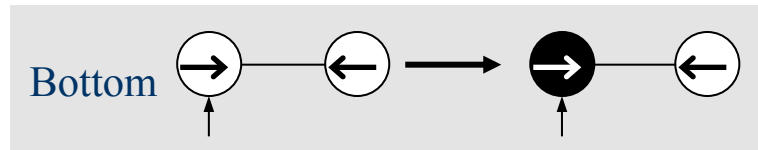


 [Dijkstra 74]



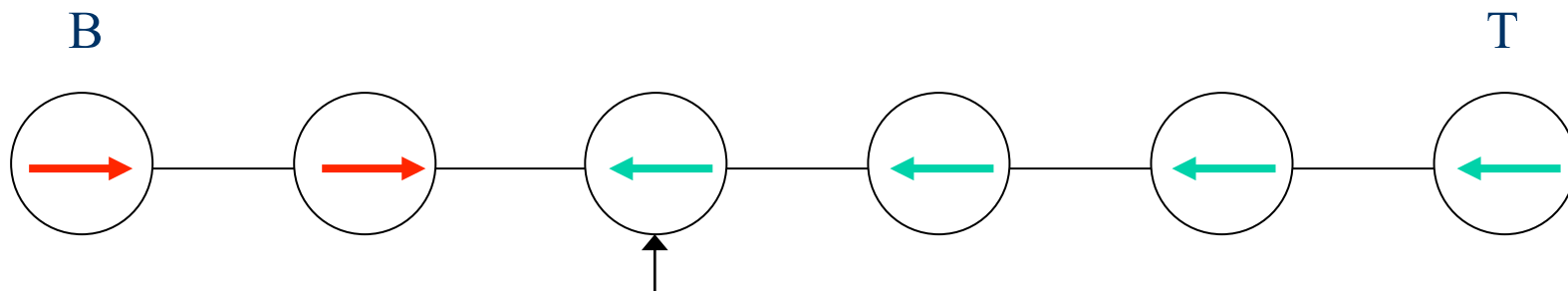
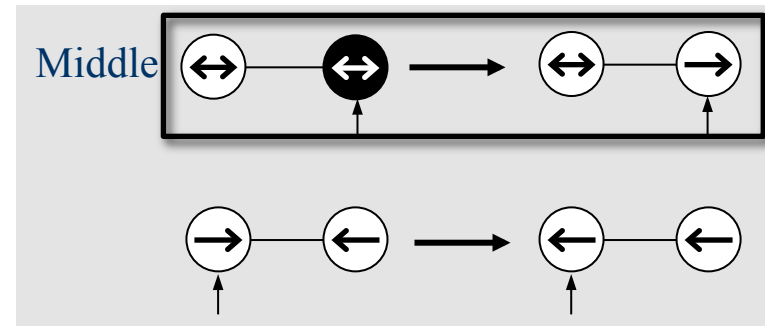
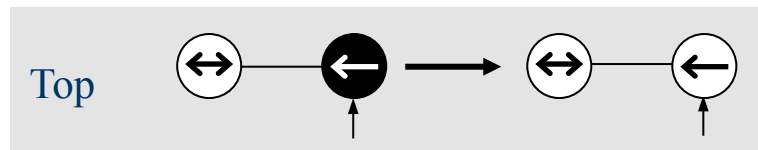
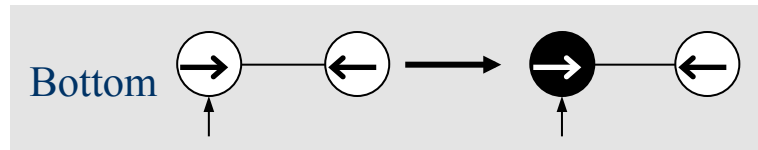


 [Dijkstra 74]



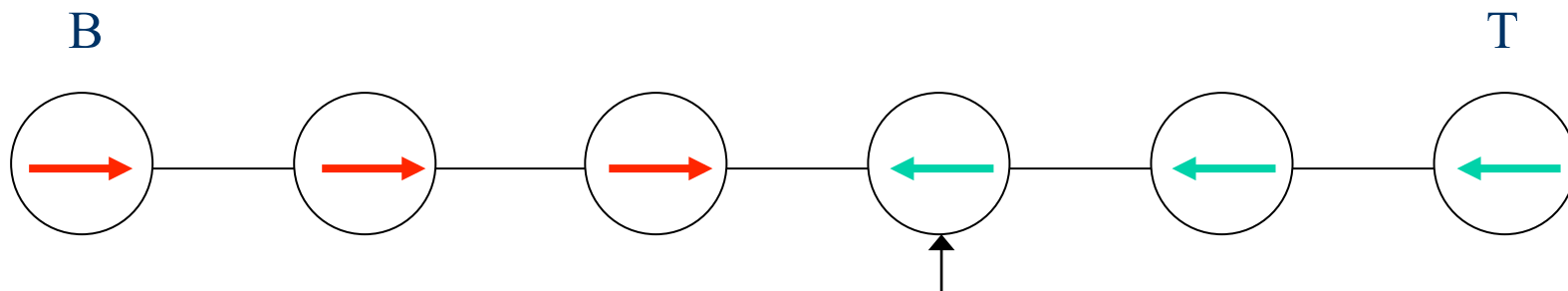
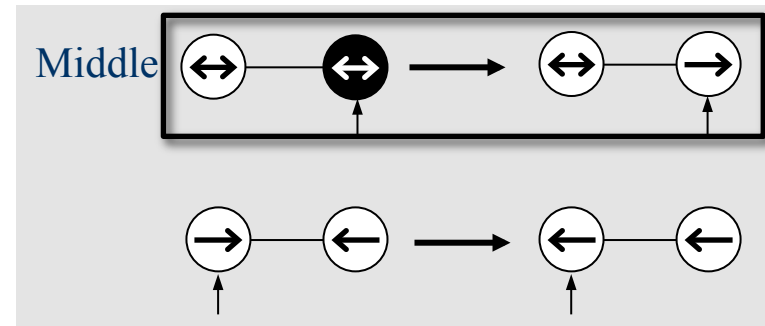
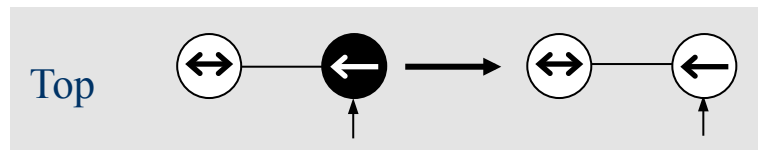
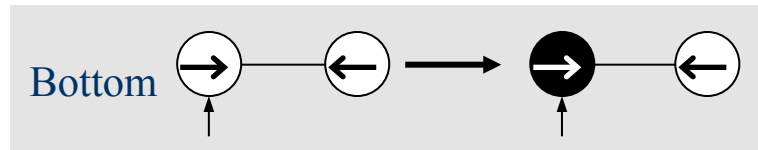


 [Dijkstra 74]



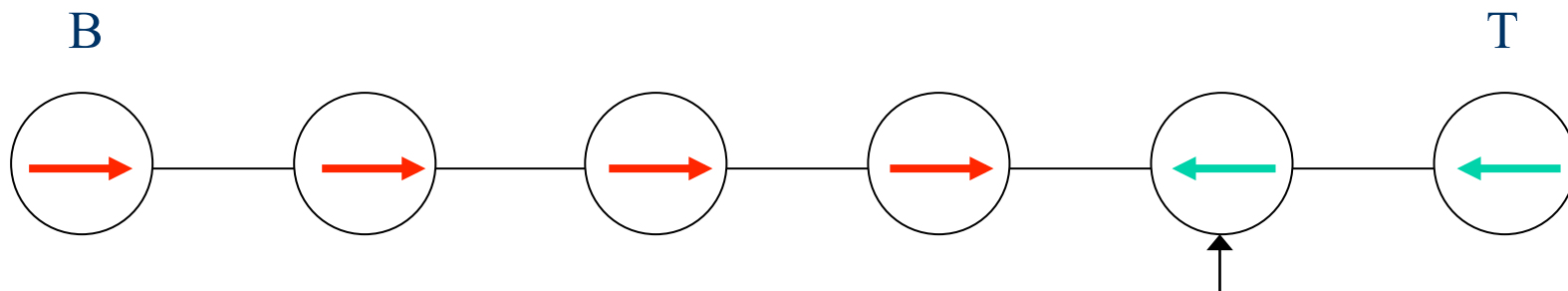
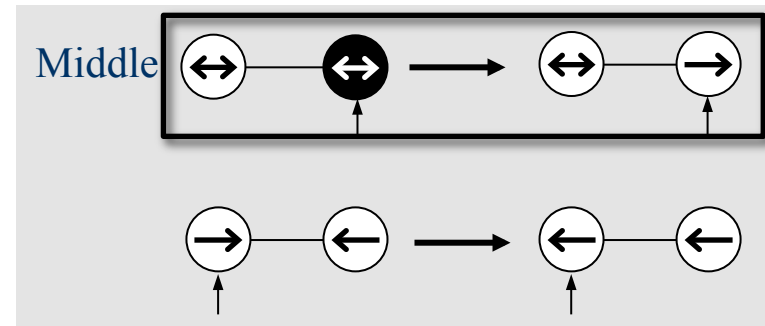
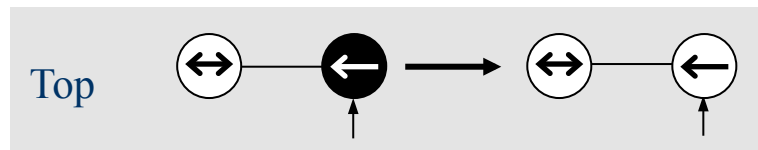
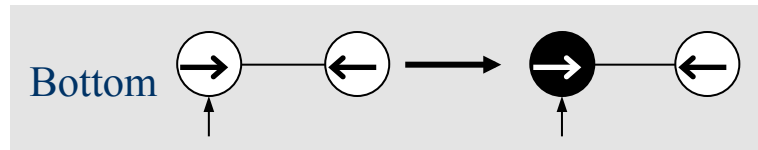


 [Dijkstra 74]



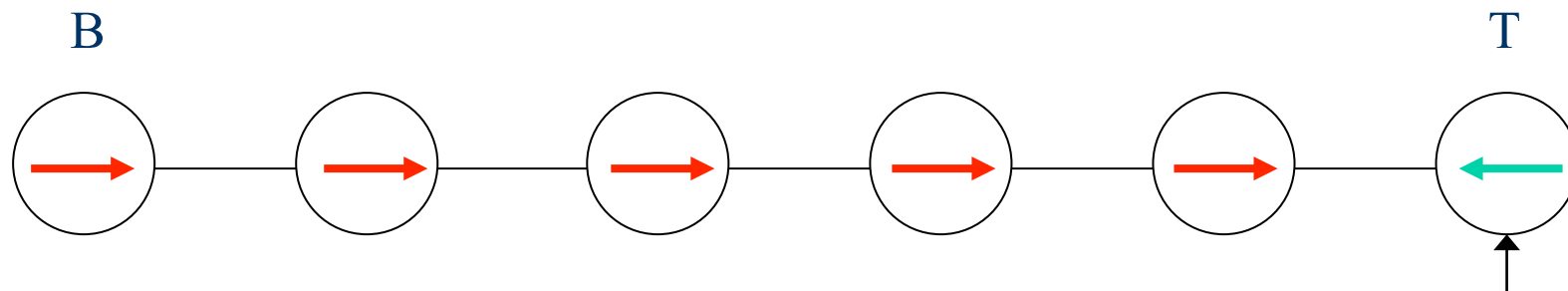
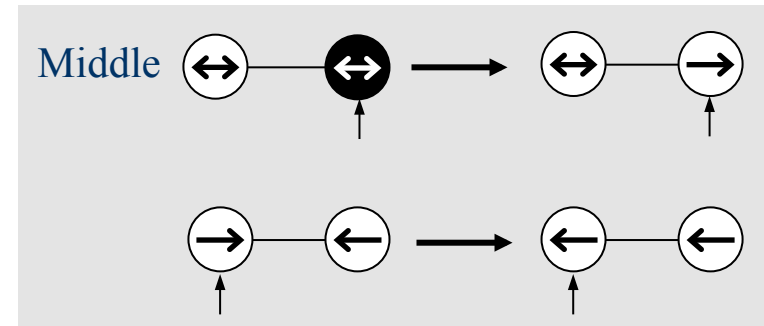
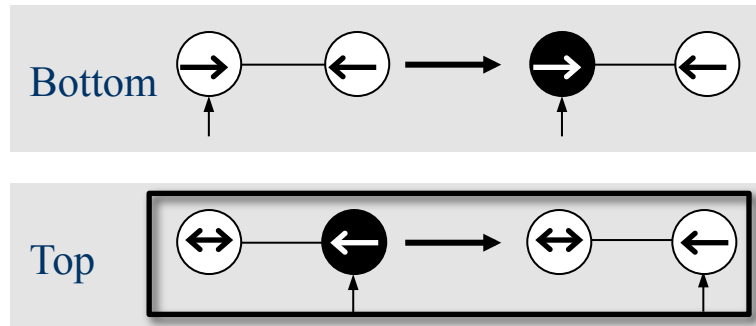


 [Dijkstra 74]



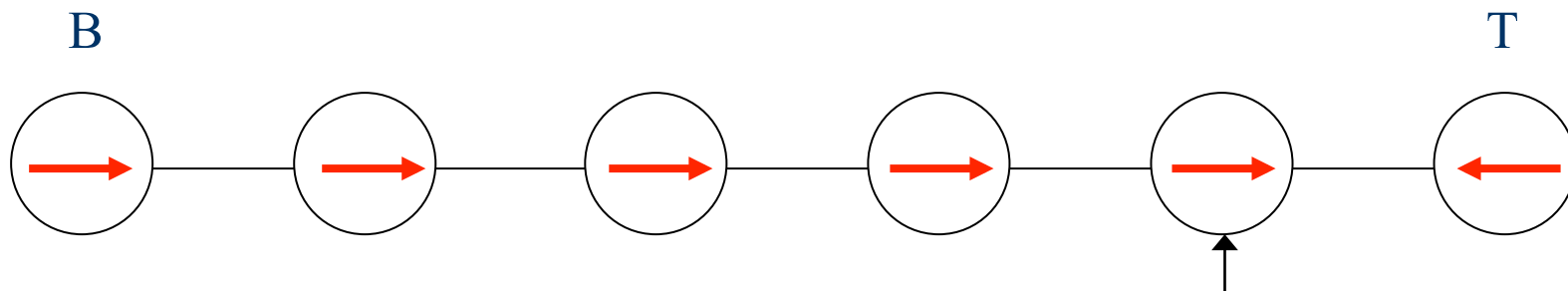
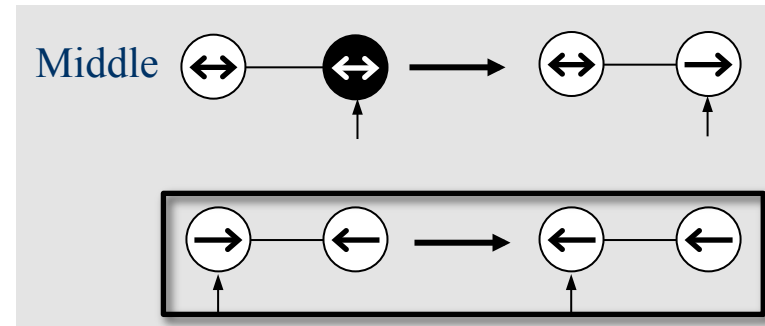
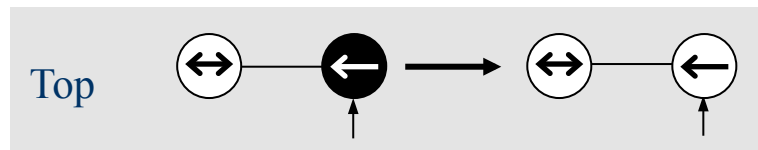
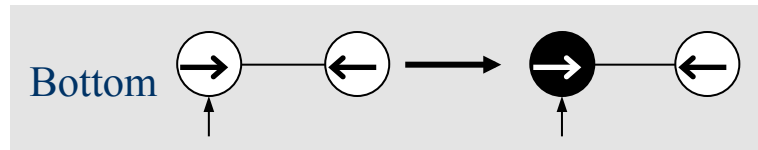


 [Dijkstra 74]



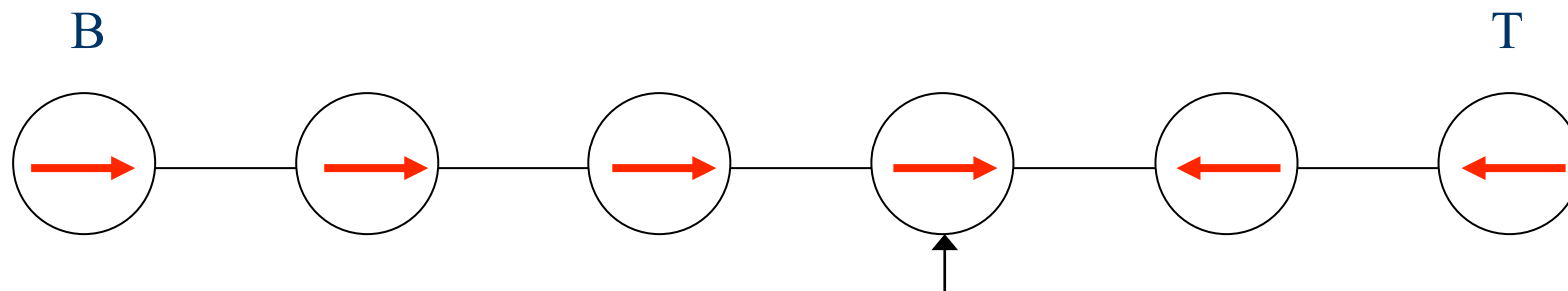
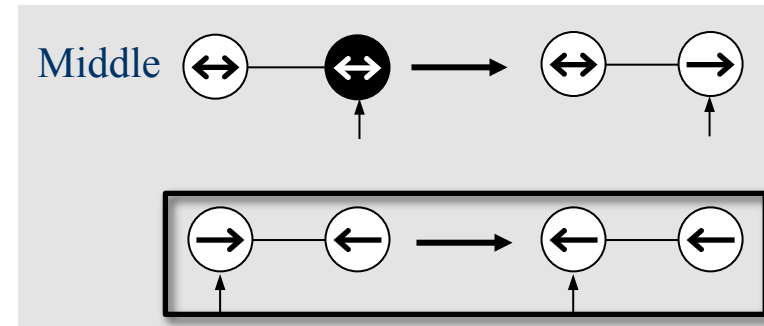
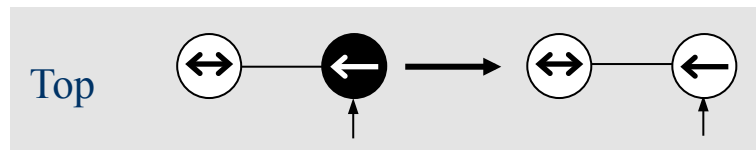
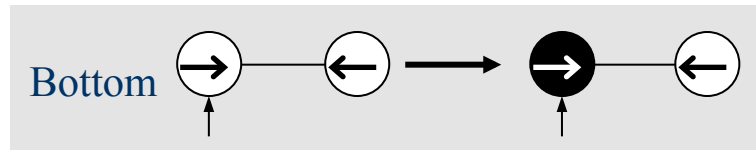


 [Dijkstra 74]



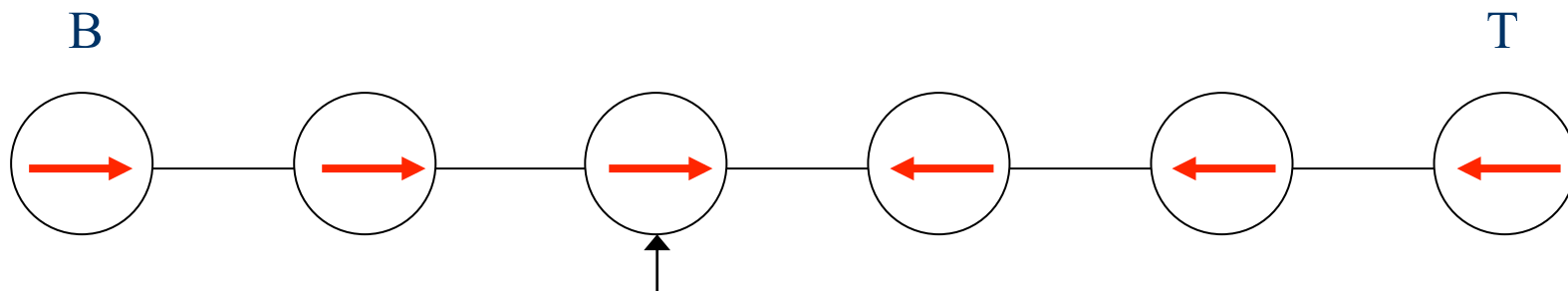
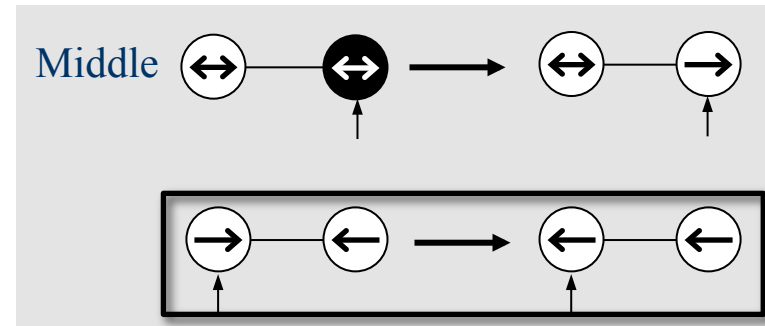
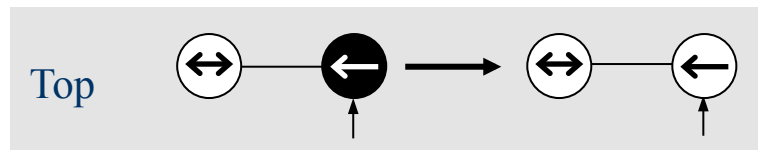
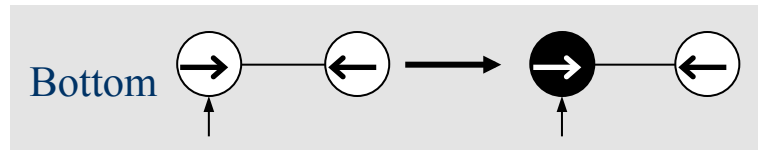


 [Dijkstra 74]



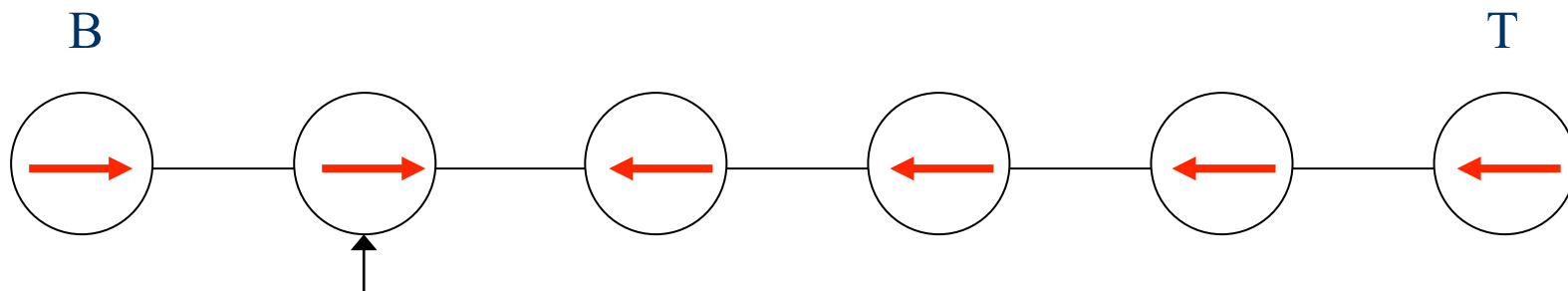
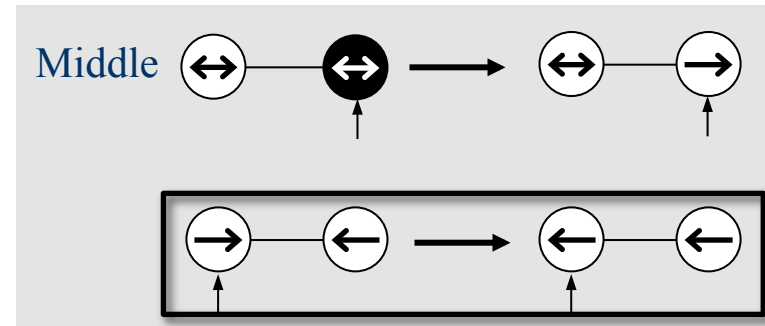
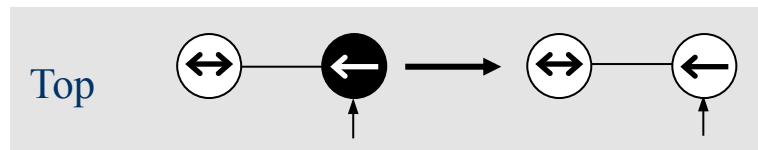
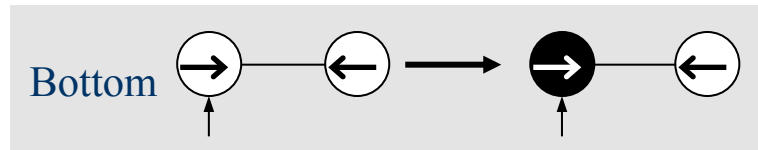


 [Dijkstra 74]



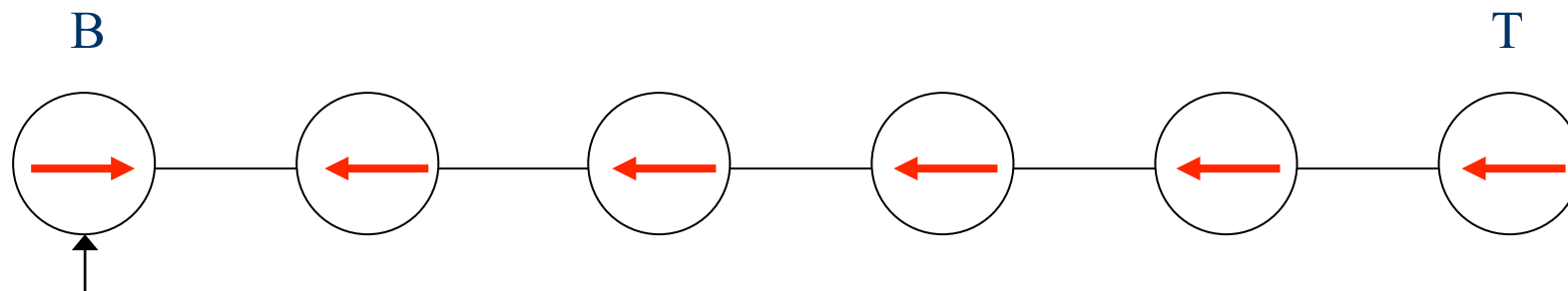
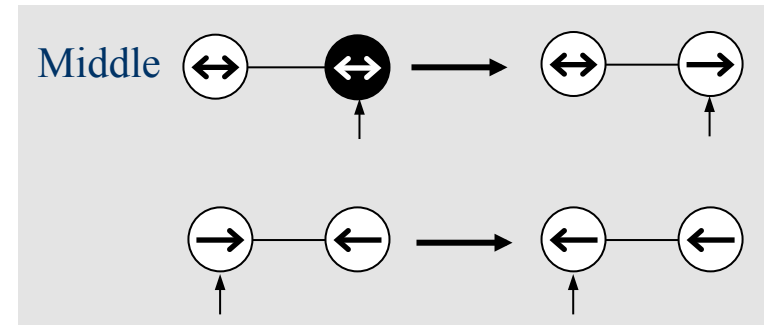
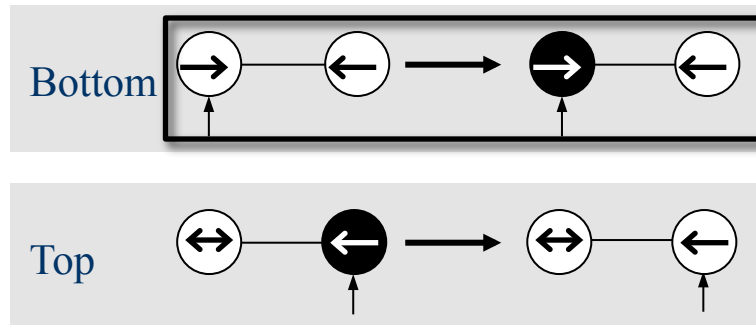


 [Dijkstra 74]



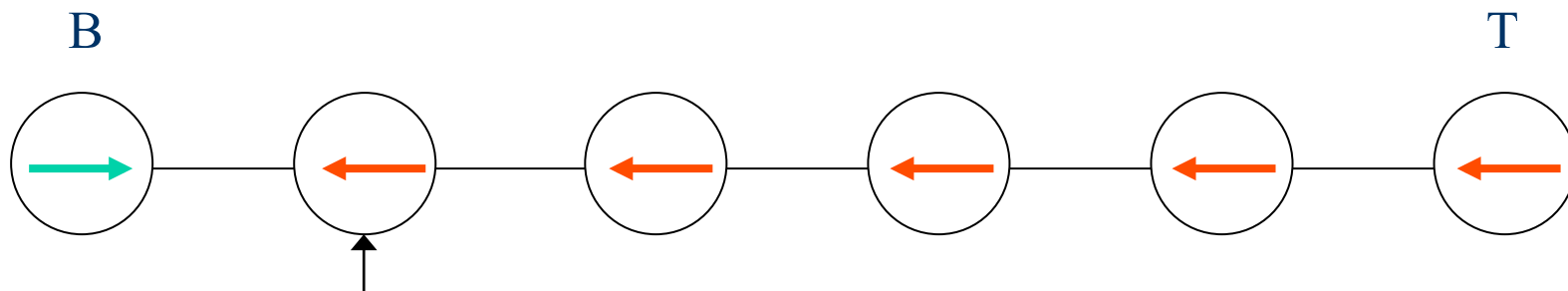
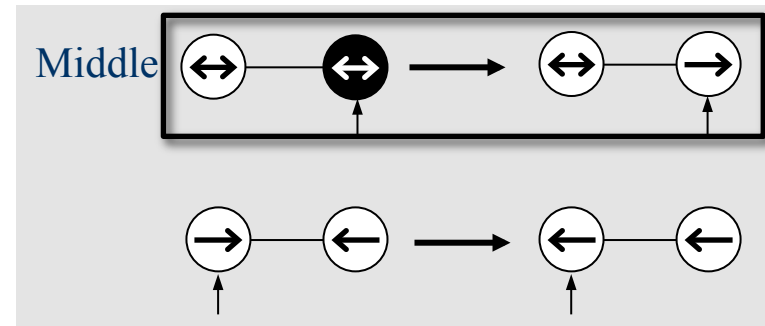
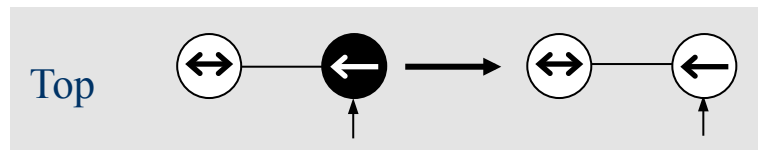
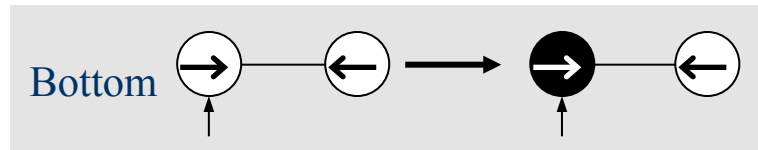


 [Dijkstra 74]



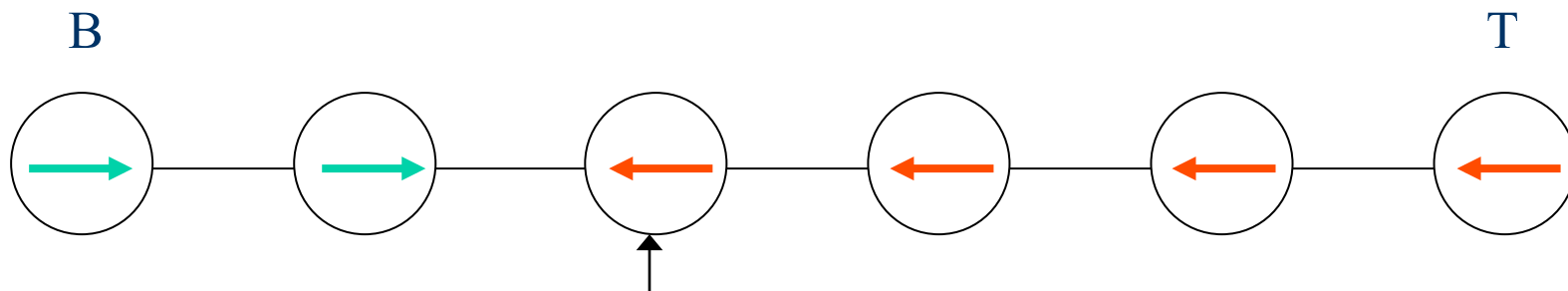
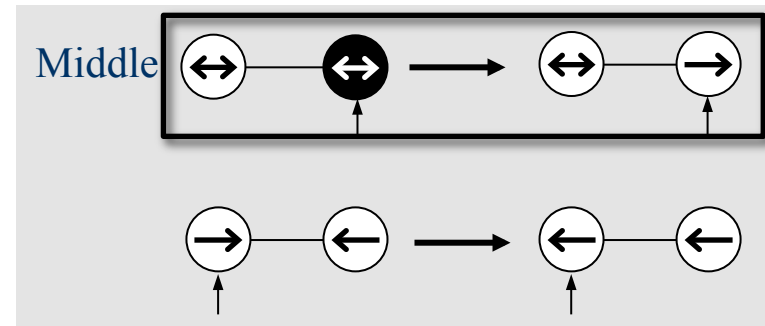
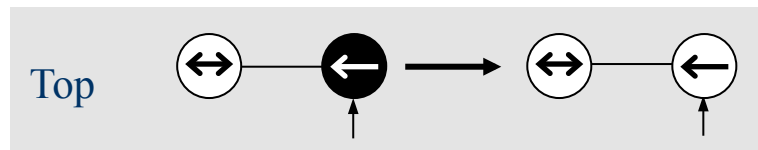
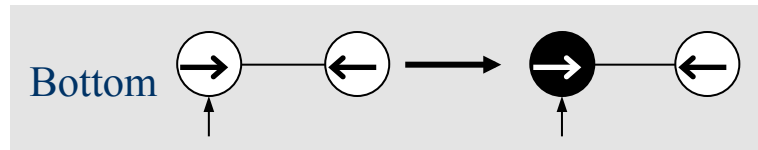


 [Dijkstra 74]



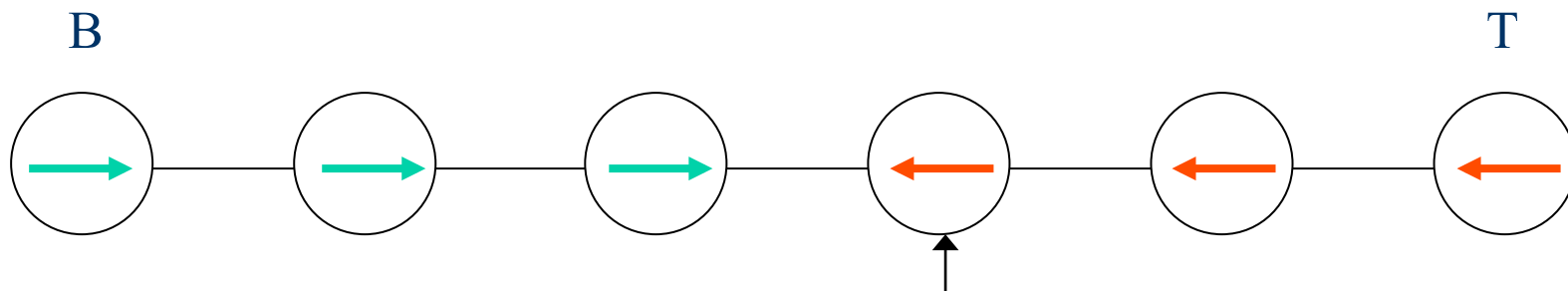
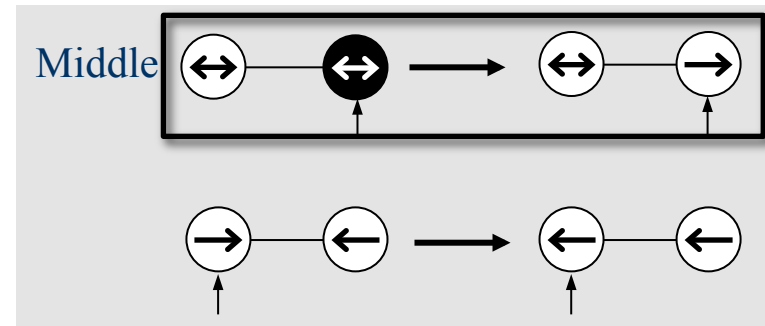
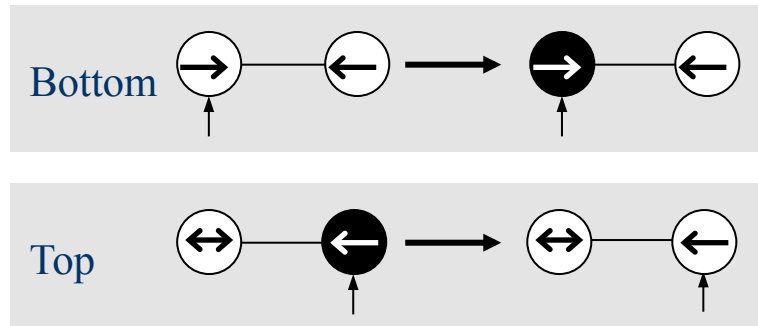


 [Dijkstra 74]



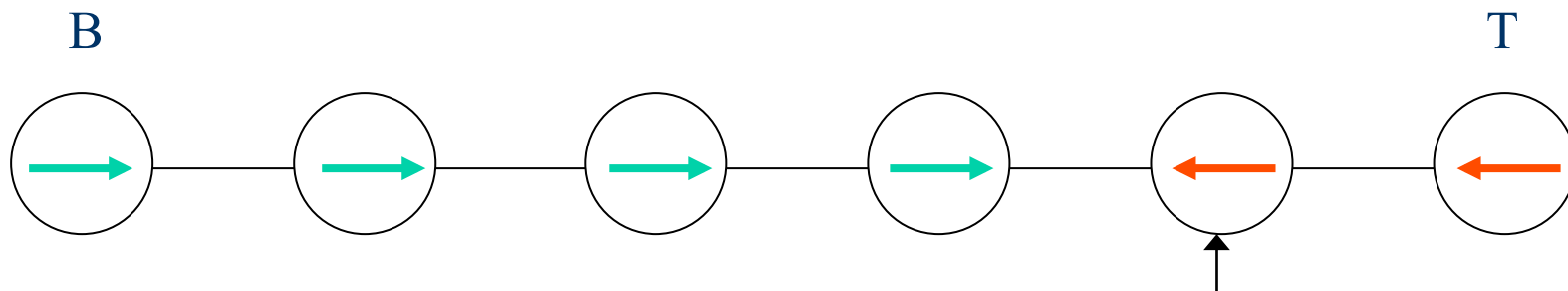
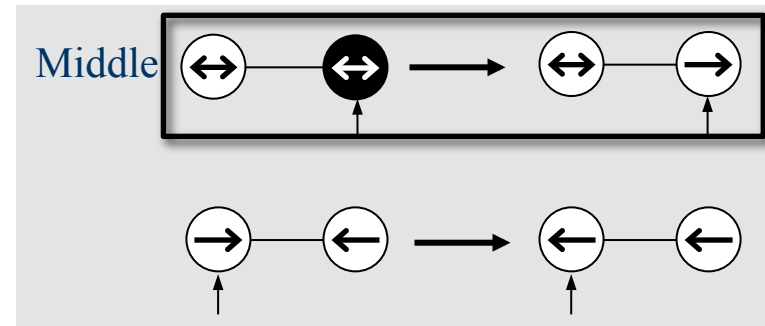
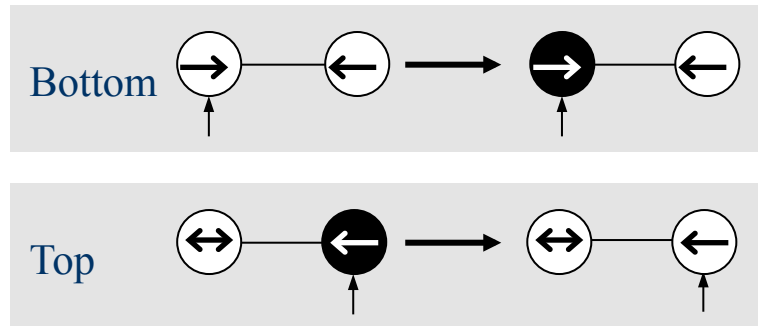


 [Dijkstra 74]



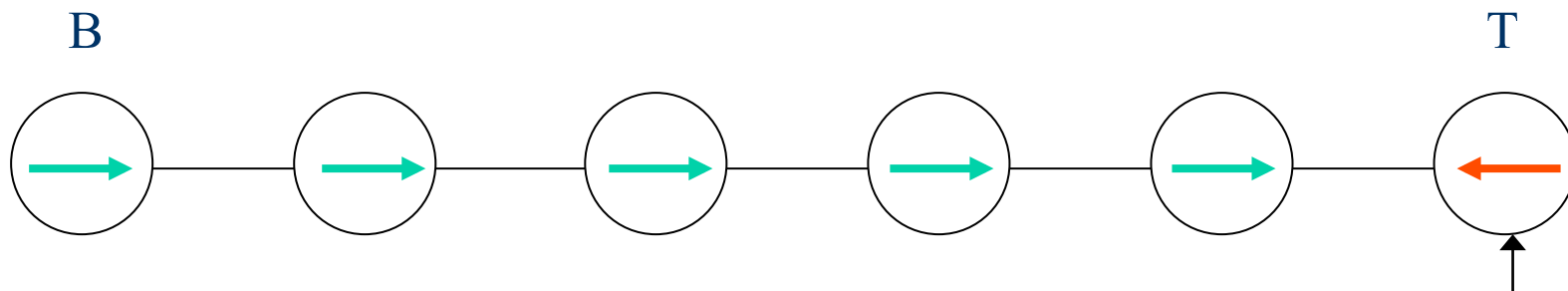
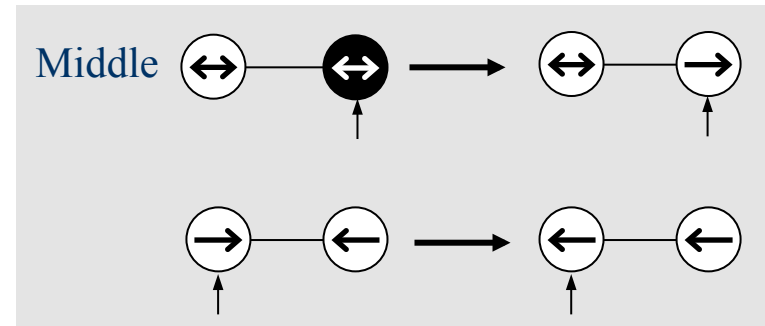
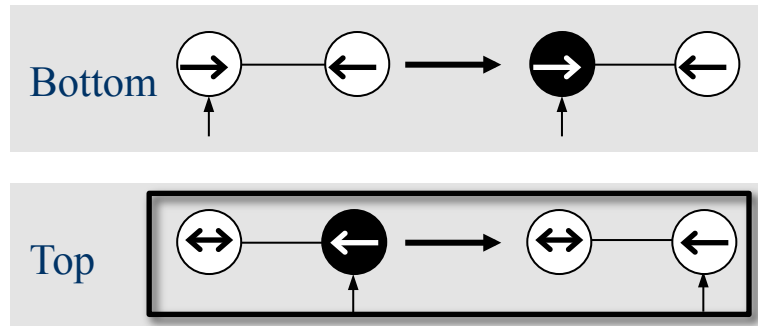


 [Dijkstra 74]



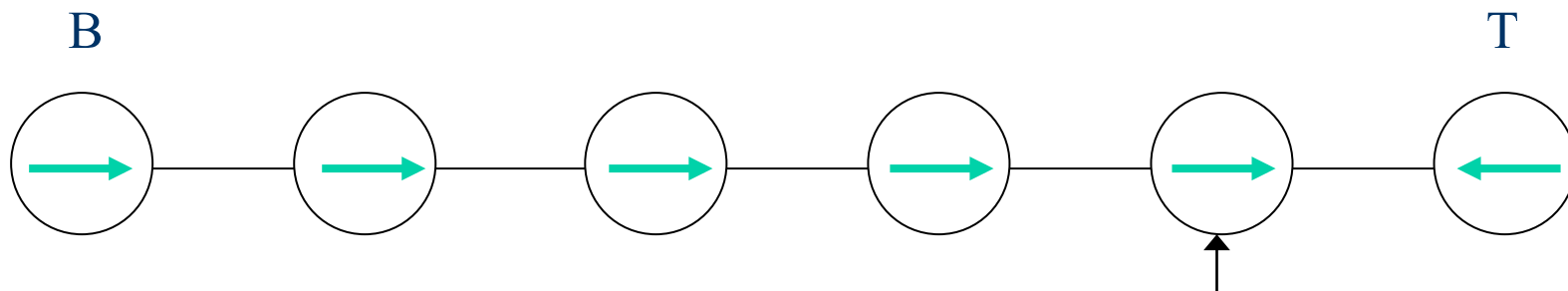
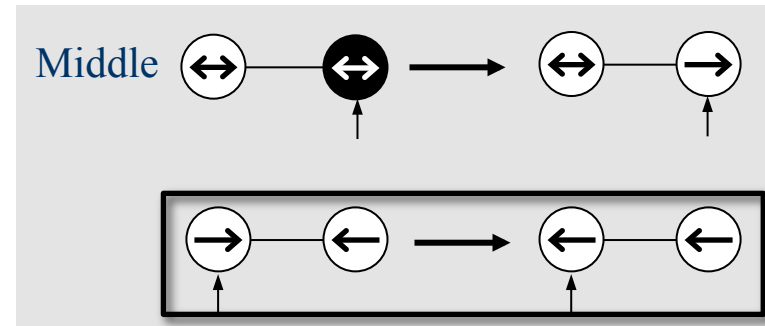
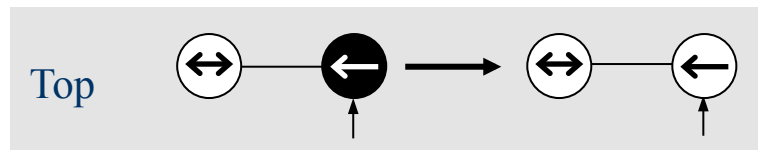
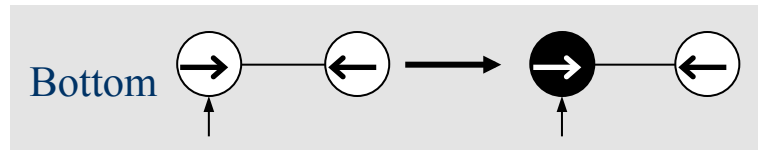


 [Dijkstra 74]



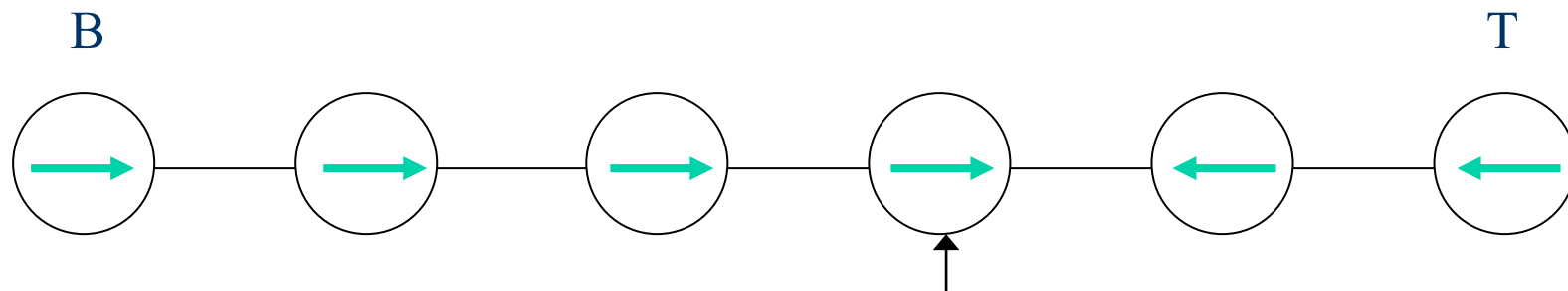
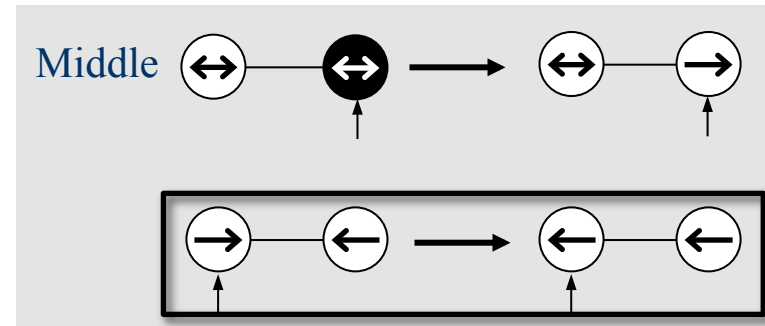
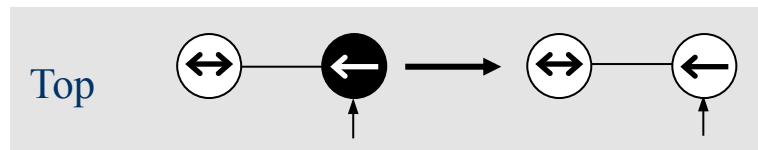
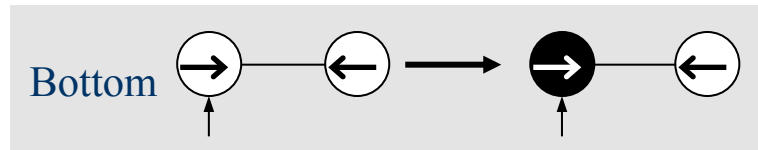


 [Dijkstra 74]



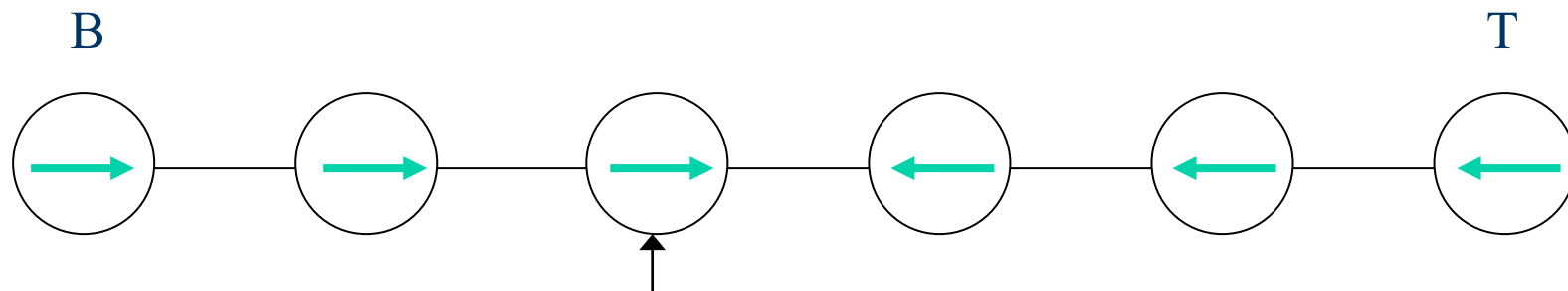
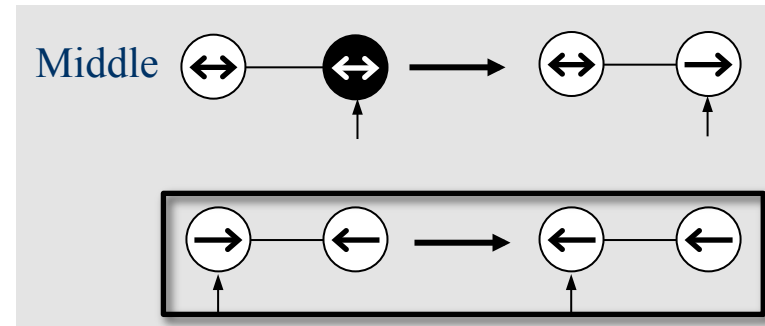
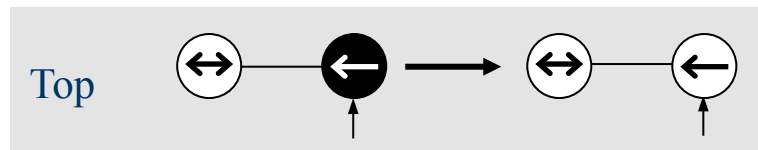
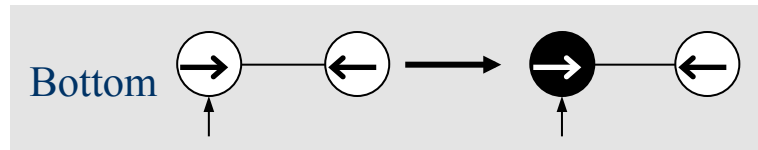


 [Dijkstra 74]



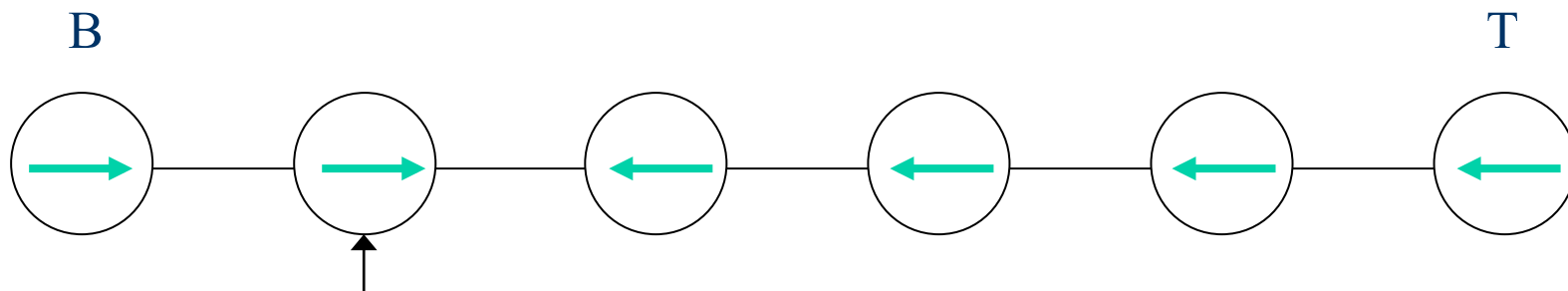
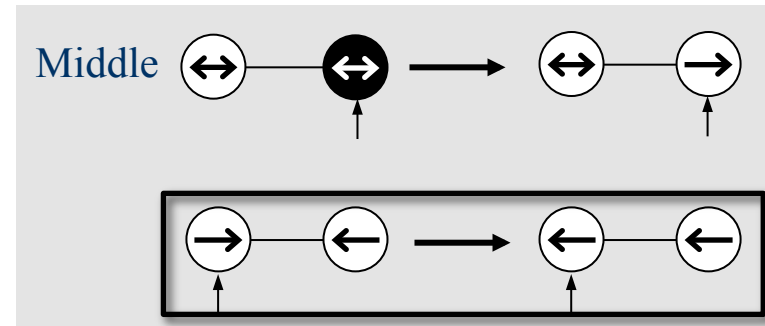
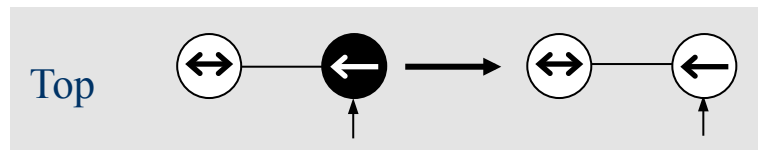
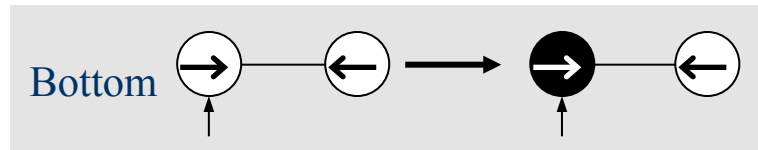


 [Dijkstra 74]



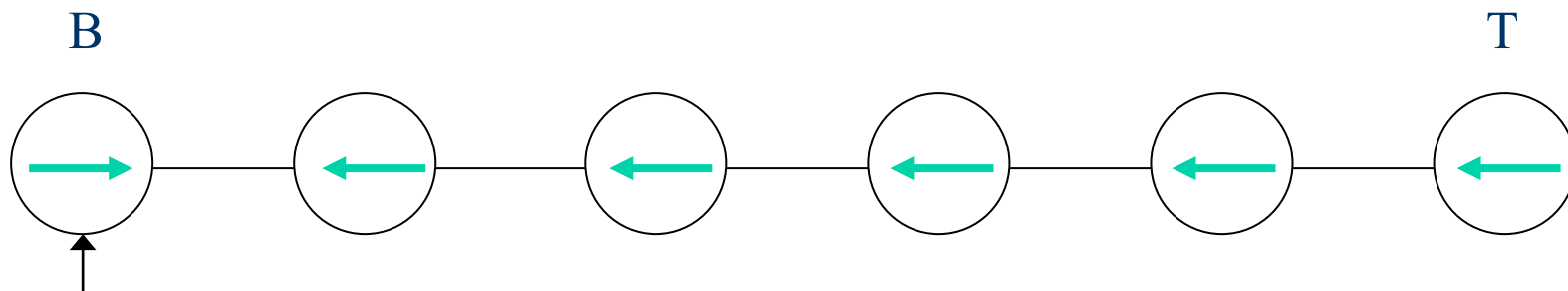
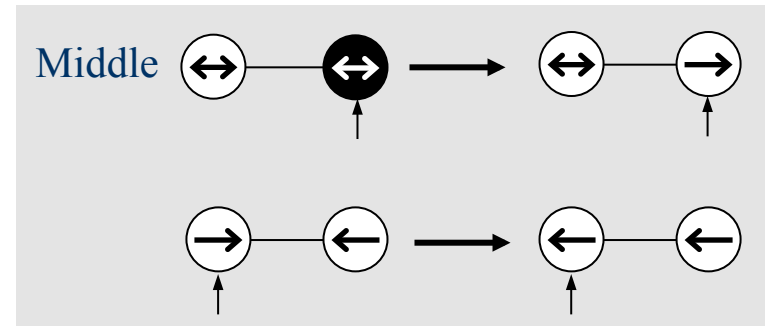
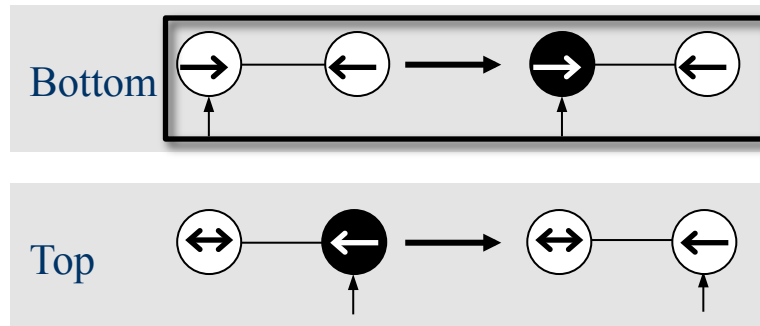


 [Dijkstra 74]





 [Dijkstra 74]

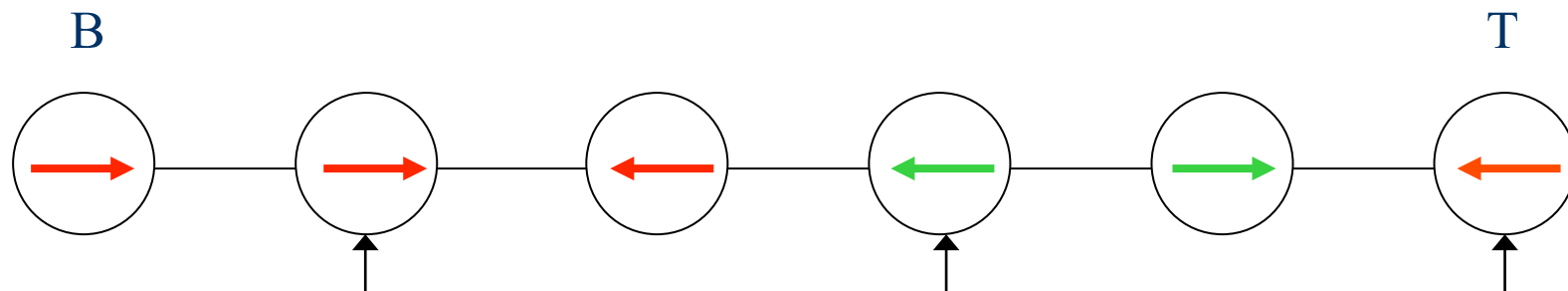
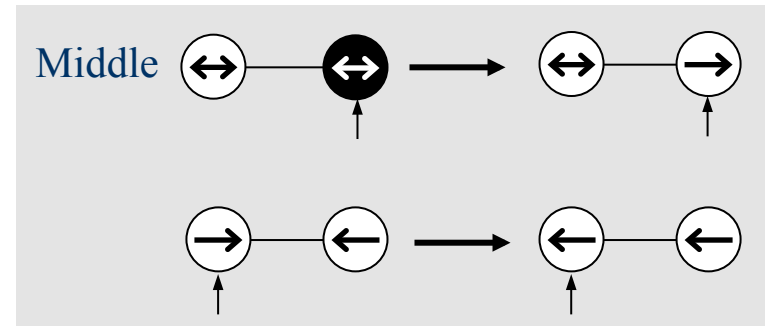
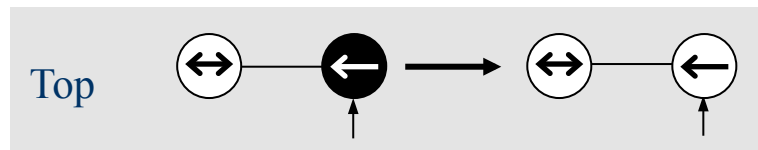
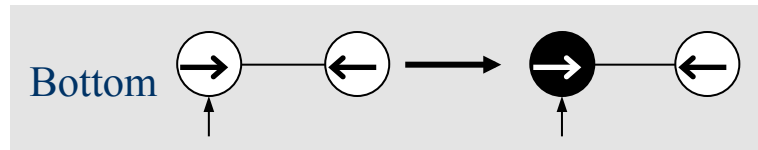




Auto-stabilisation ?

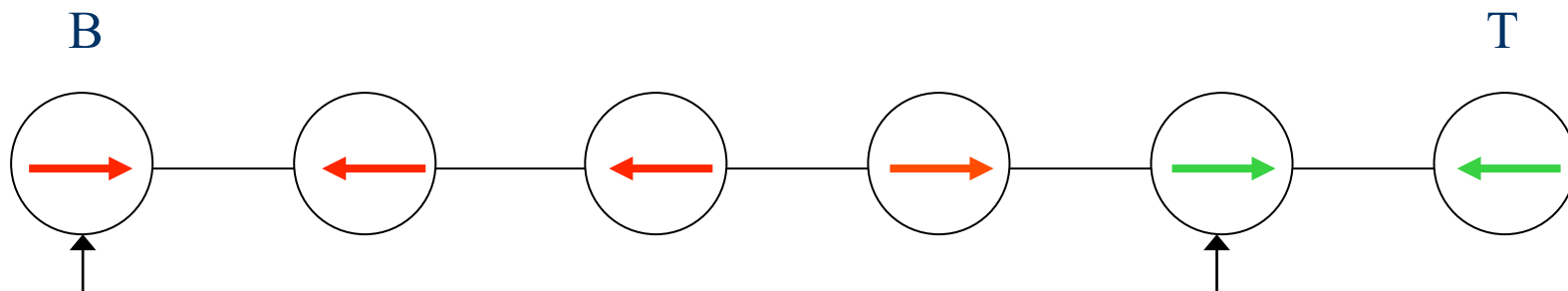
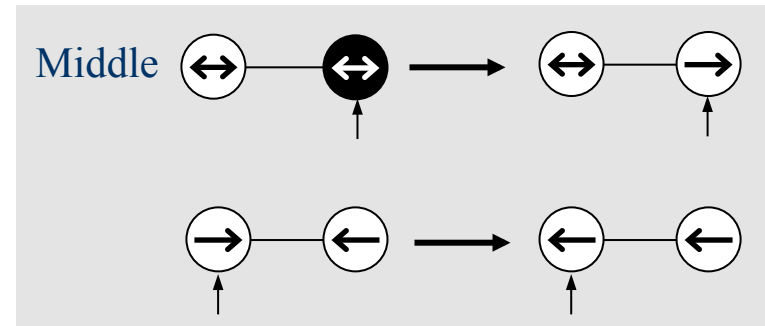
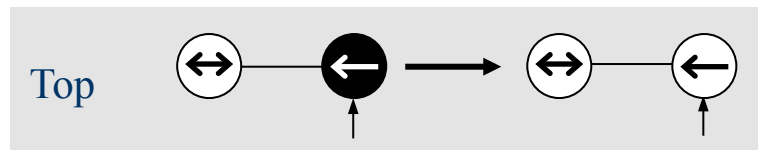
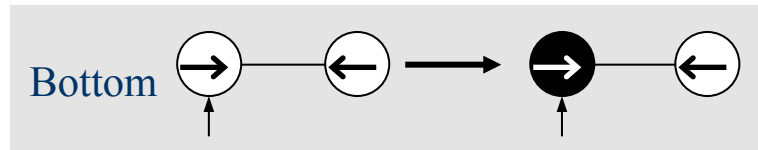


 [Dijkstra 74]



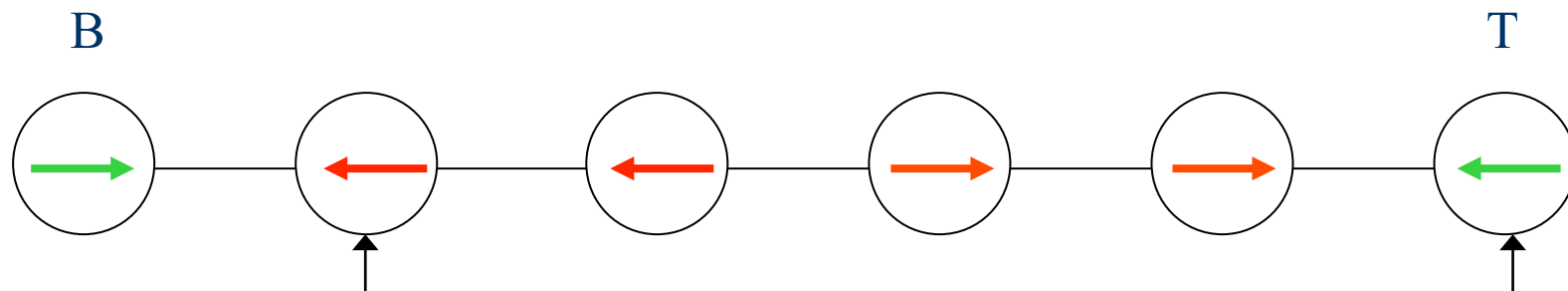
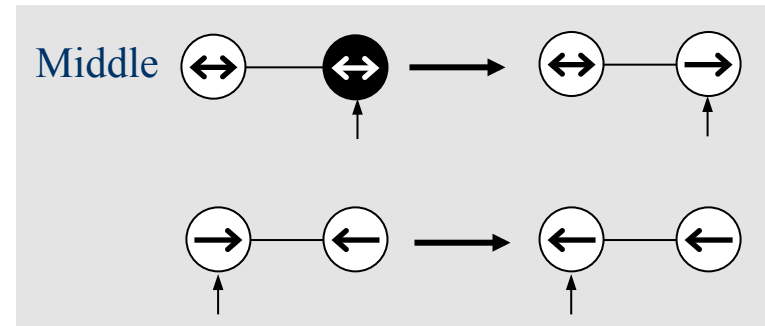
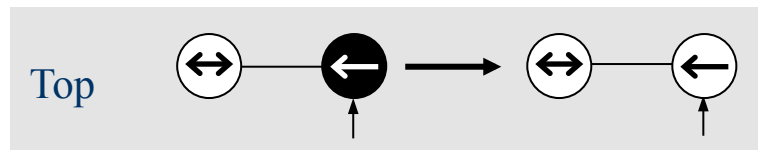
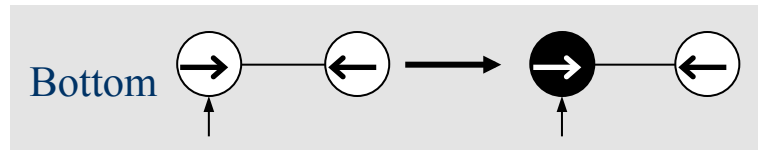


 [Dijkstra 74]



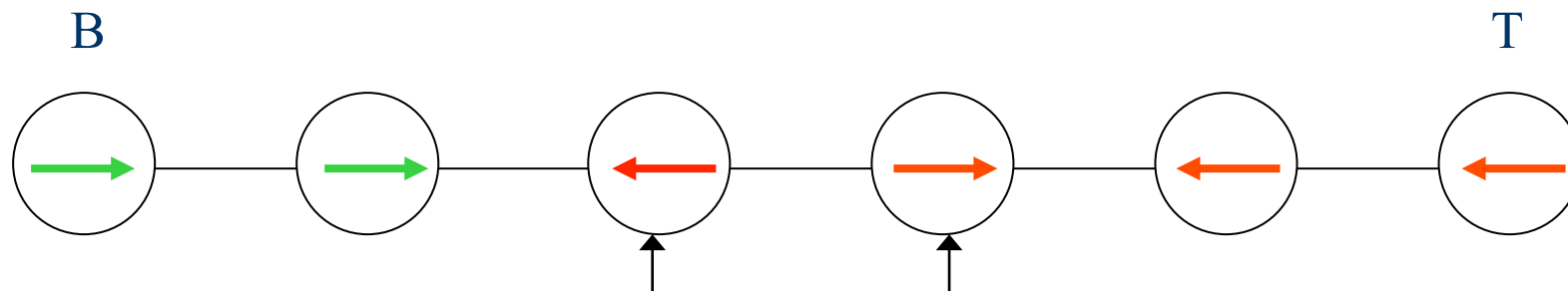
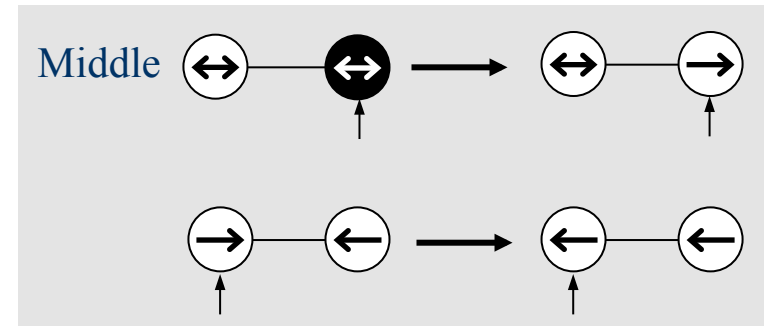
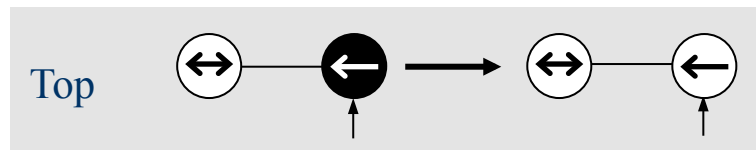
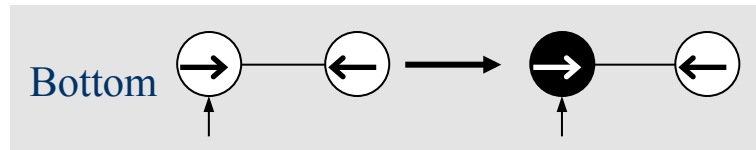


 [Dijkstra 74]



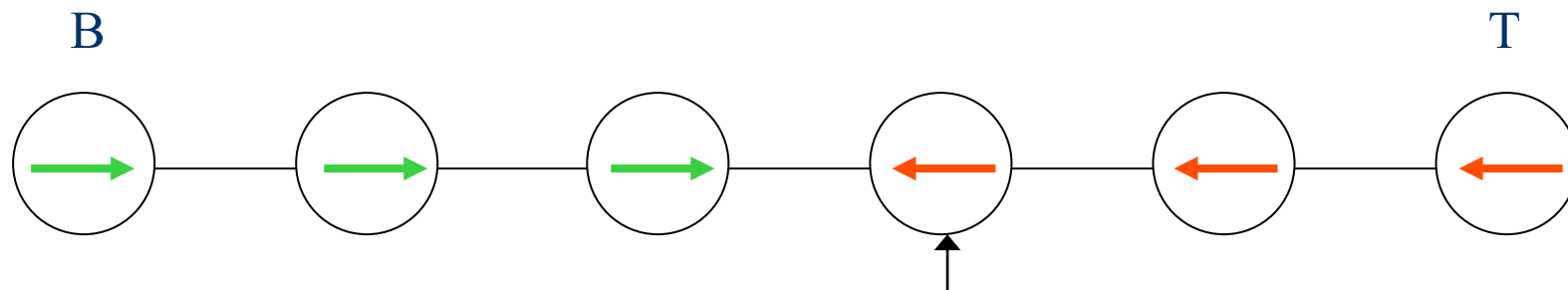
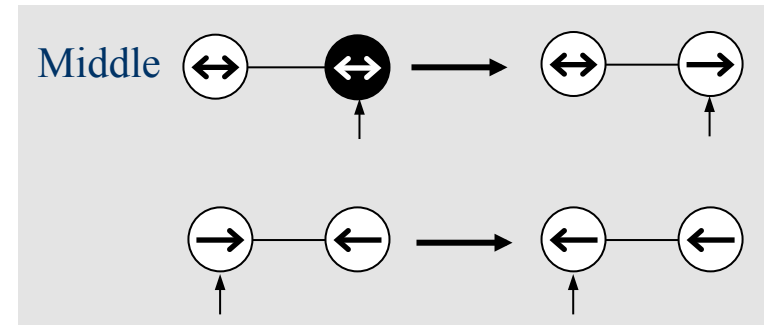
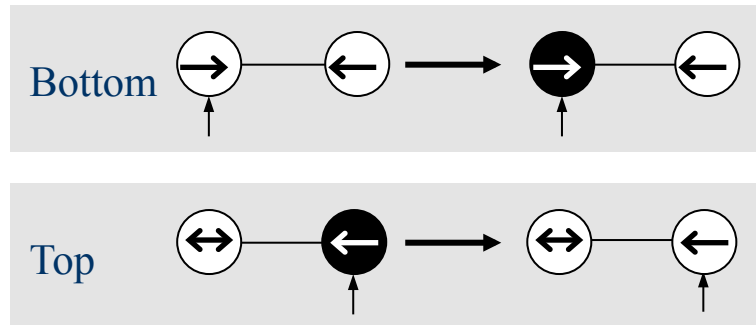


 [Dijkstra 74]





 [Dijkstra 74]



Stabilisé !

