Improving the algorithms for performance evaluation in FASE

speaker: Massimo Callisto De Donato

{massimo.callisto}@unicam.it

Scuola di Scienze e Tecnologie - Università di Camerino
Via Madonna delle Carceri, 9 - 62032 Camerino
http://www.cs.unicam.it

PaCo Meeting - L'Aquila
March, 3rd 2010

FASE performance improvement

$$\exists\ a \in \mathbb{R}\ s.t.\ rp_P(n)\ = an + \Theta(1)$$

- FASE implements the quantitative characterization of the performance based on the response processes as stated before. In particular FASE calculates:
  1. **catastrophic cycles** – correctness of the process
  2. **n-critical paths** – exact value of $rp_P(n)$ for a given n
  3. **bad cycle(s)** – coefficient "a" of $rp_P(n)$

- Improving the used algorithms would be an important goal to increase FASE *usability* in *current* usage

Catastrophic cycles

- A response process P could contain one (or more) **catastrophic cycle(s)**
- A catastrophic cycles is composed by:
  1. τ's action
  2. time steps

Catastrophic cycles

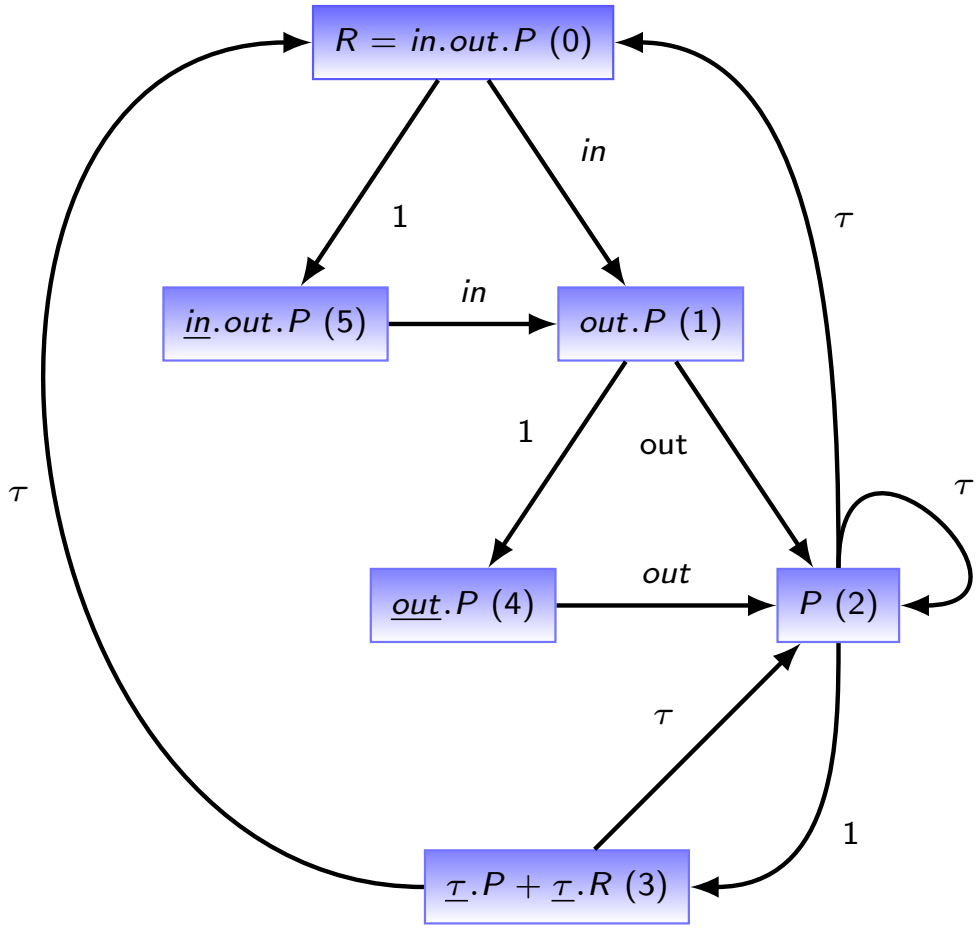
- A response process P could contain one (or more) **catastrophic cycle(s)**
- A catastrophic cycles is composed by:
  1. τ's action
  2. time steps
- When a process enters one of these cycles, it is impossible to know **when** (and **if**) it will come out, thus system performance cannot be calculated

Catastrophic cycles

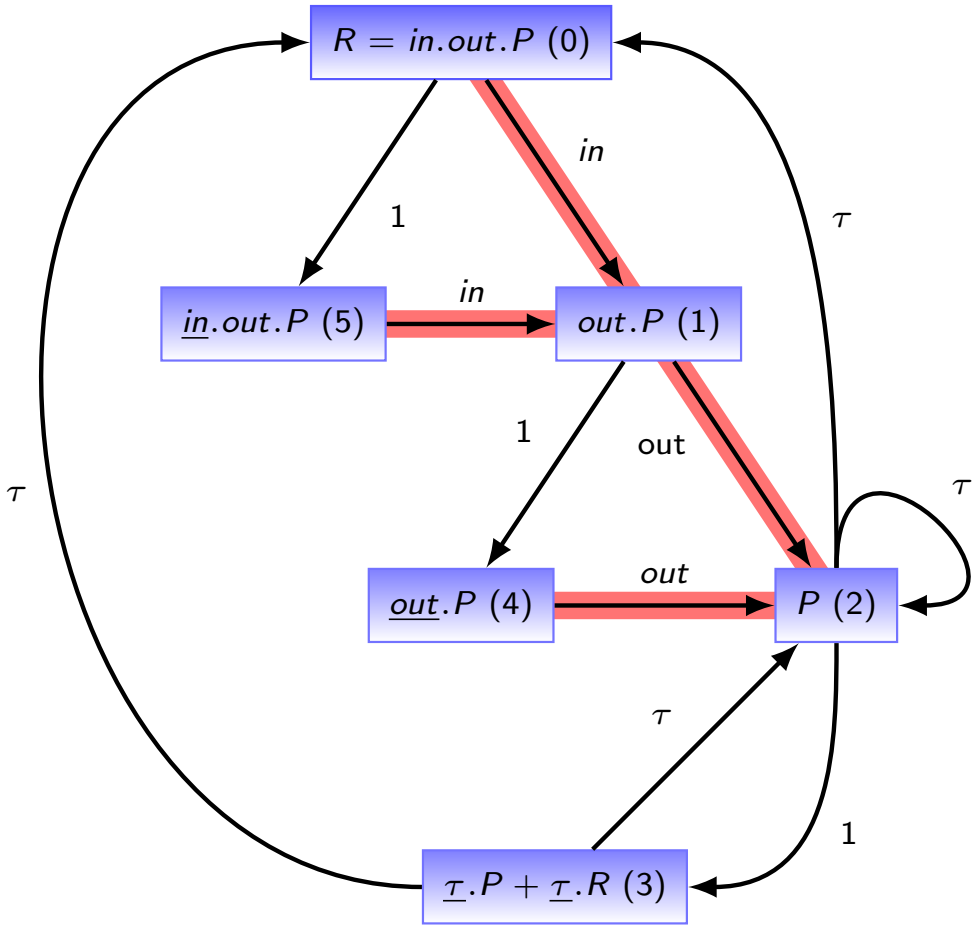
- A response process P could contain one (or more) **catastrophic cycle(s)**
- A catastrophic cycles is composed by:
  1. τ's action
  2. time steps
- When a process enters one of these cycles, it is impossible to know **when** (and **if**) it will come out, thus system performance cannot be calculated
- It is crucial to ensure that the process is free from these cycles

Catastrophic cycles - Original algorithm

$R = in.out.P;$
$P = \tau.P + \tau.R;$



Catastrophic cycles - Original algorithm



$R = in.out.P;$

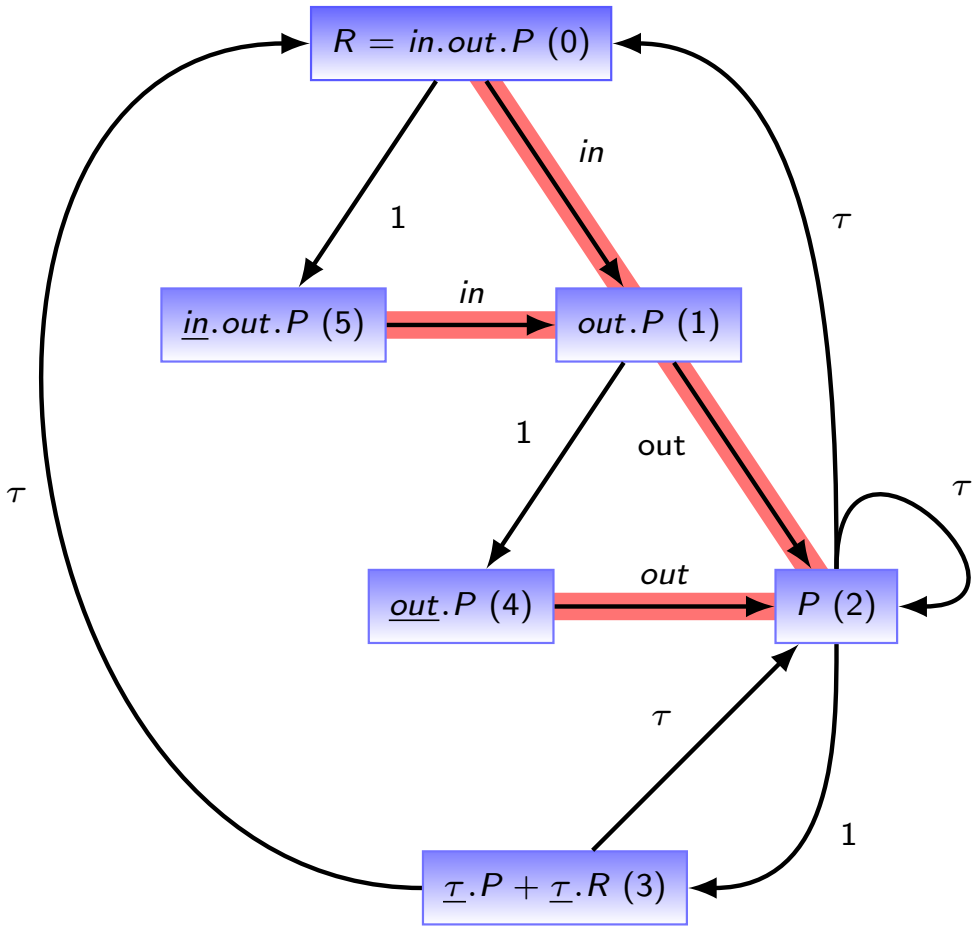
$P = \tau.P + \tau.R;$

Algorithm Steps:

1. *in* and *out* edges removal

Catastrophic cycles - Original algorithm

$R = in.out.P;$
$P = \tau.P + \tau.R;$



Algorithm Steps:

1. *in* and *out* edges removal
2. Floyd-Warshall shortest paths matrix calculation
   -1 for time step
   0 for τ's

	0	1	2	3	4	5
0	∞	∞	∞	∞	∞	-1
1	∞	∞	∞	∞	-1	∞
2	-1	∞	-1	-1	∞	-2
3	0	∞	0	-1	∞	-1
4	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞

Improving Catastrophic cycles calculation

- The original method that uses the Floyd-Warshall algorithm has a complexity of $\mathbf{\Theta(N^3)}$ (with N number of nodes)

Improving Catastrophic cycles calculation

- The original method that uses the Floyd-Warshall algorithm has a complexity of $\mathbf{\Theta(N^3)}$ (with N number of nodes)
- The new algorithm is based on the well-known algorithm for finding the "*Strongly Connected Components*" in a graph, that has a complexity of $\mathbf{\Theta(N + E)}$ (N number of nodes, E number of edges)

Improving Catastrophic cycles calculation

- The original method that uses the Floyd-Warshall algorithm has a complexity of $\mathbf{\Theta(N^3)}$ (with N number of nodes)
- The new algorithm is based on the well-known algorithm for finding the "*Strongly Connected Components*" in a graph, that has a complexity of $\mathbf{\Theta(N + E)}$ (N number of nodes, E number of edges)
- An SCC of $G = (N, E)$ is a subset $K \subseteq N$ s.t. $\forall u, v \in K$, $\exists$ a path $u \rightsquigarrow v$ and vice versa

Improving Catastrophic cycles calculation

- The original method that uses the Floyd-Warshall algorithm has a complexity of $\mathbf{\Theta(N^3)}$ (with N number of nodes)
- The new algorithm is based on the well-known algorithm for finding the "*Strongly Connected Components*" in a graph, that has a complexity of $\mathbf{\Theta(N + E)}$ (N number of nodes, E number of edges)
- An SCC of $G = (N, E)$ is a subset $K \subseteq N$ s.t. $\forall u, v \in K$, $\exists$ a path $u \rightsquigarrow v$ and vice versa

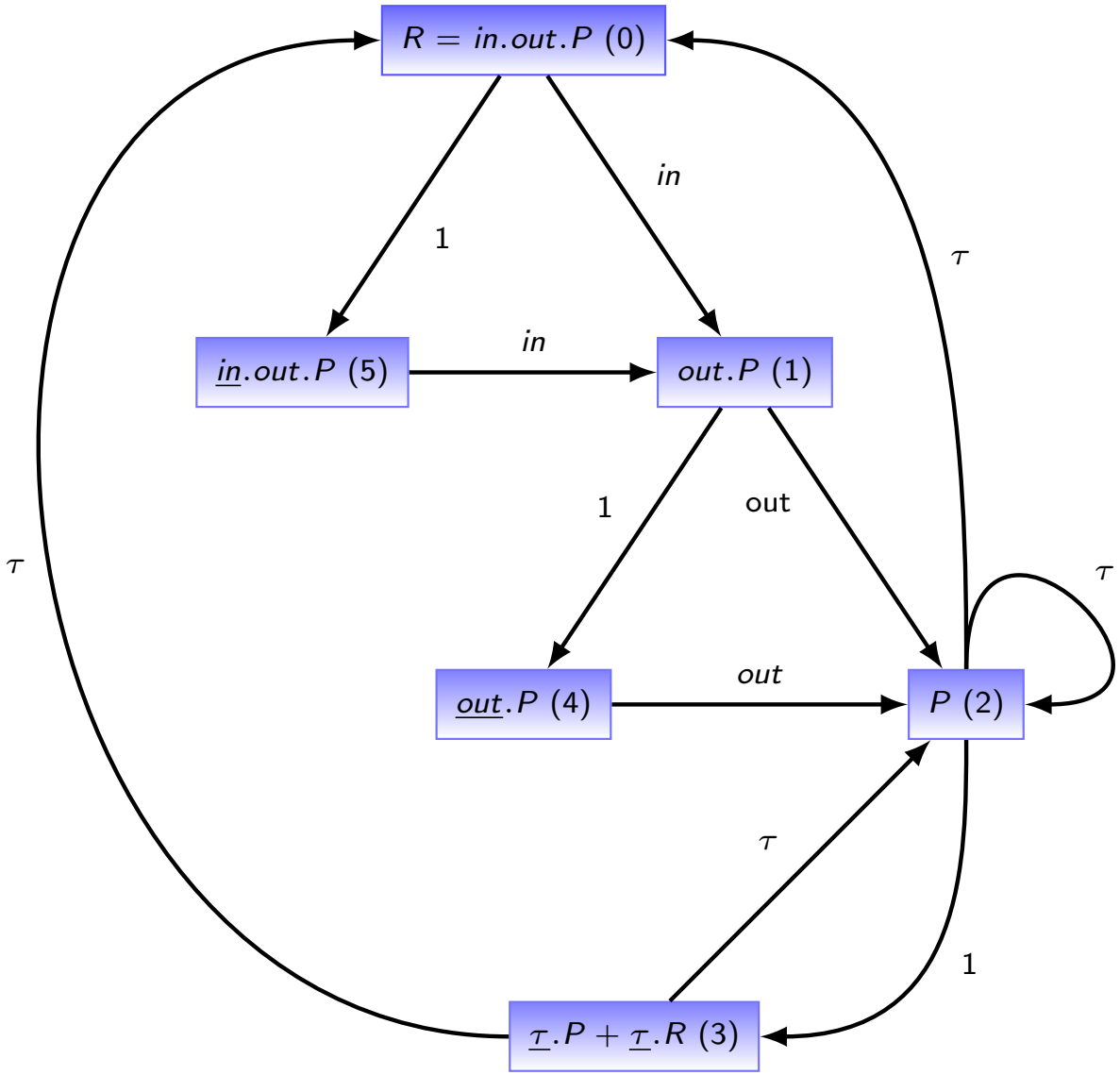
1. We consider a graph G obtained from $rRTS(P)$ with all *in* and *out* actions removed

Improving Catastrophic cycles calculation

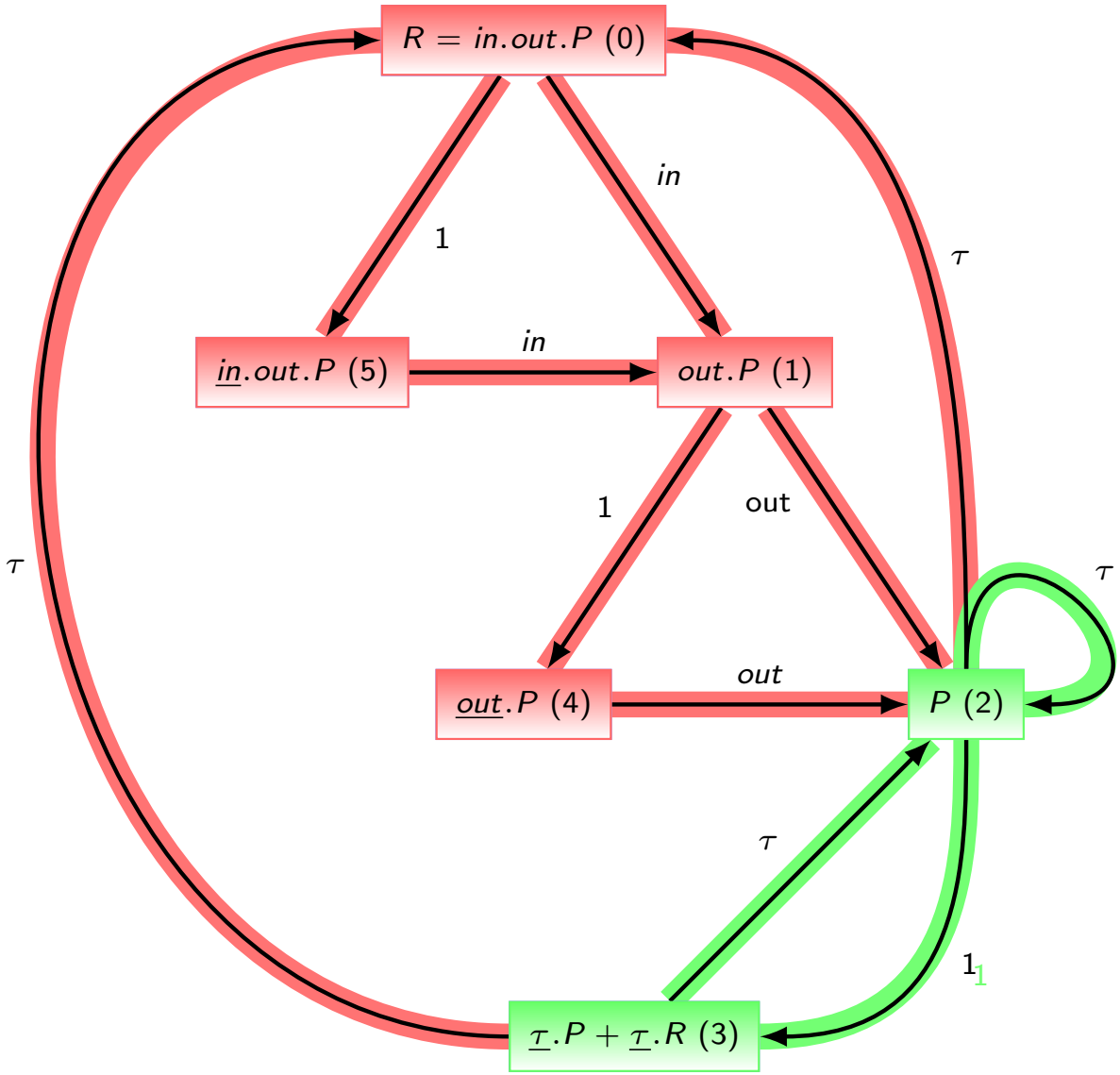
- The original method that uses the Floyd-Warshall algorithm has a complexity of $\mathbf{\Theta(N^3)}$ (with N number of nodes)
- The new algorithm is based on the well-known algorithm for finding the "*Strongly Connected Components*" in a graph, that has a complexity of $\mathbf{\Theta(N + E)}$ (N number of nodes, E number of edges)
- An SCC of $G = (N, E)$ is a subset $K \subseteq N$ s.t. $\forall u, v \in K$, $\exists$ a path $u \rightsquigarrow v$ and vice versa

1. We consider a graph G obtained from $rRTS(P)$ with all *in* and *out* actions removed
2. We compute all the SCCs in G and if one of them contains some time steps then there is a catastrophic cycle in $rRTS(P)$

SCCs in the example process



SCCs in the example process



Catastrophic cycles algorithms performance comparison

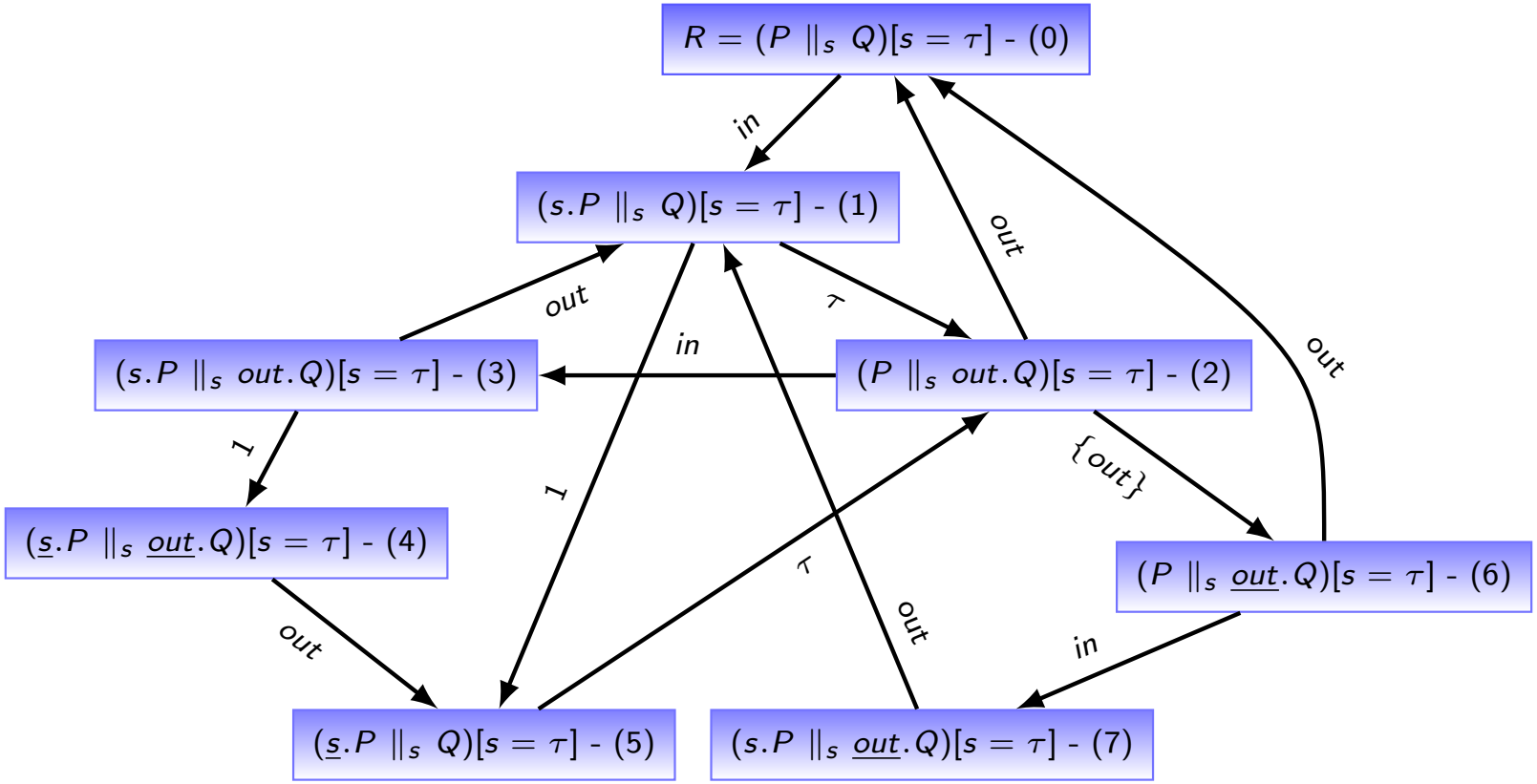
	nodes/edges	*Floyd*	*SCC*	*Gain*
$Fifo_5$	13/28	0	0	–
$Pipe_5$	114/292	62	16	74.1%
$Buff_5$	96/216	16	15	0%
$Fifo_7$	17/38	0	16	0%
$Pipe_7$	648/1958	11484	296	97.4%
$Buff_7$	240/560	390	47	87.9%
$Fifo_9$	21/48	0	0	–
$Pipe_9$	3680/12902	–	9687	100%
$Buff_9$	448/1064	2922	109	96.2%

Table: Catastrophic cycles detection time (expressed in *ms*)

n-critical paths

- Useful to calculate exactly the value rp of P for a given n
- We have not found a suitable algorithm that works in polynomial time to solve this problem
- Current algorithm calculates all n-critical path starting from the root node of process P to return the worst one (i.e. the longest in term of time steps)
- Clearly this approach has an **exponential complexity**

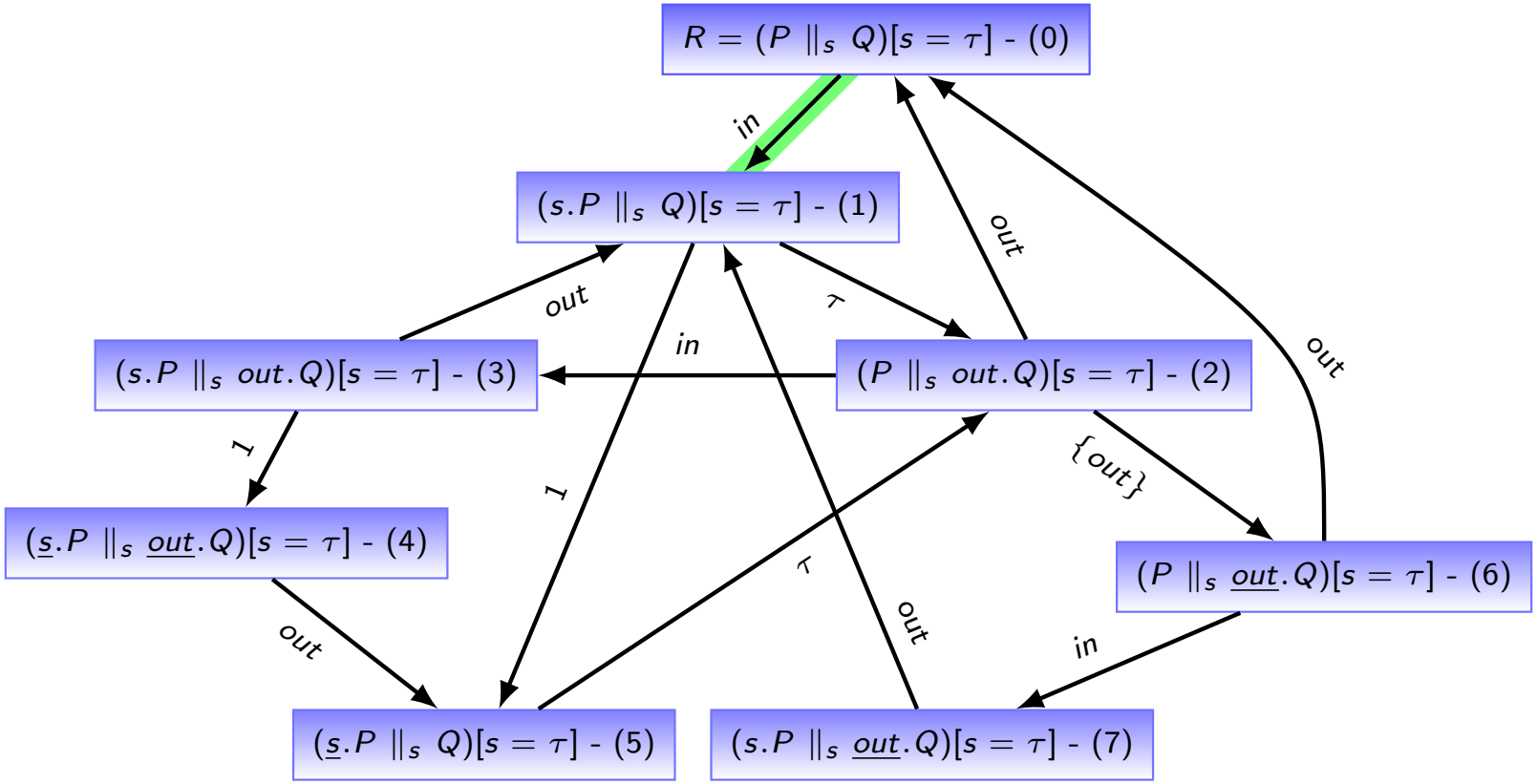
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time:

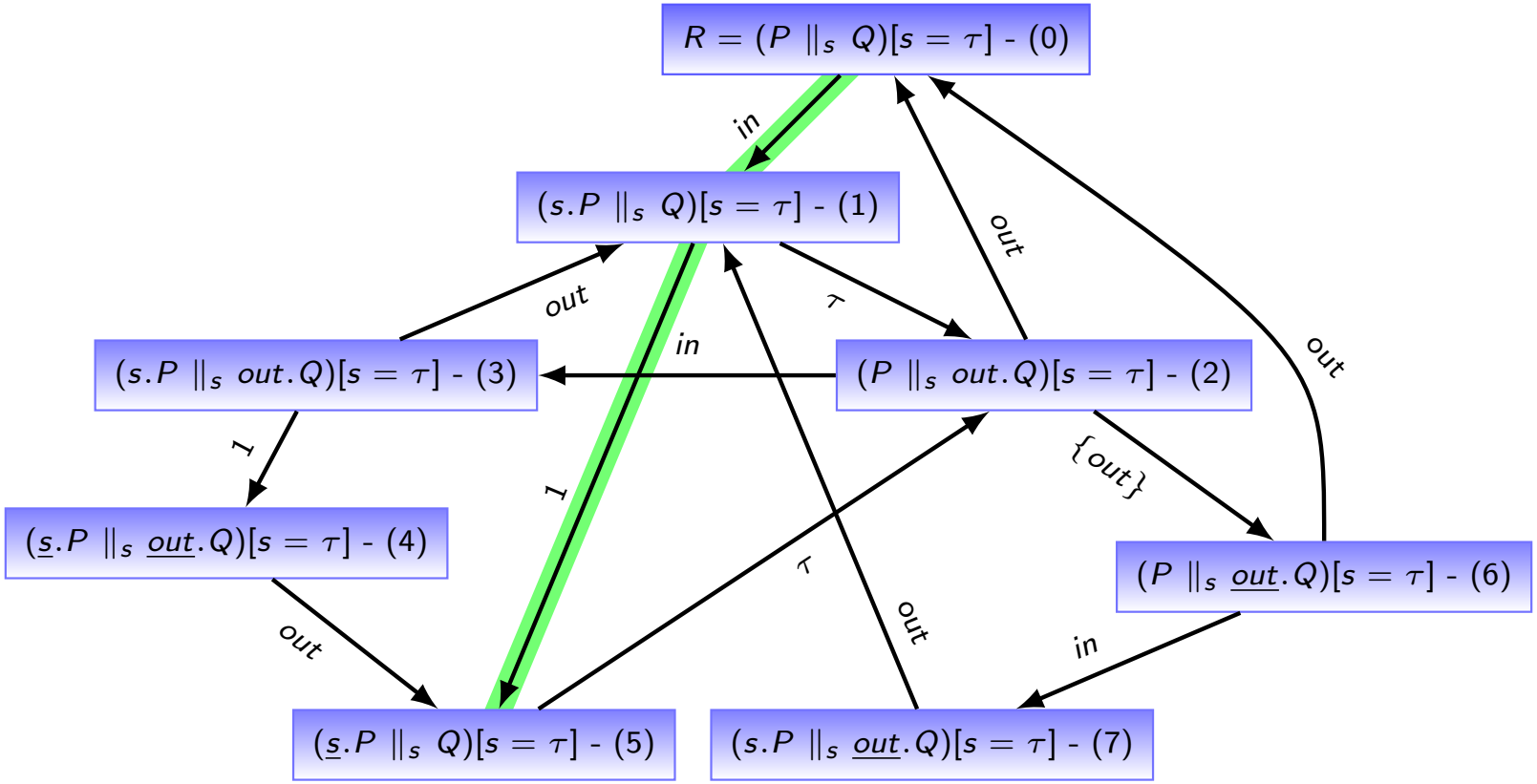
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = in.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time:

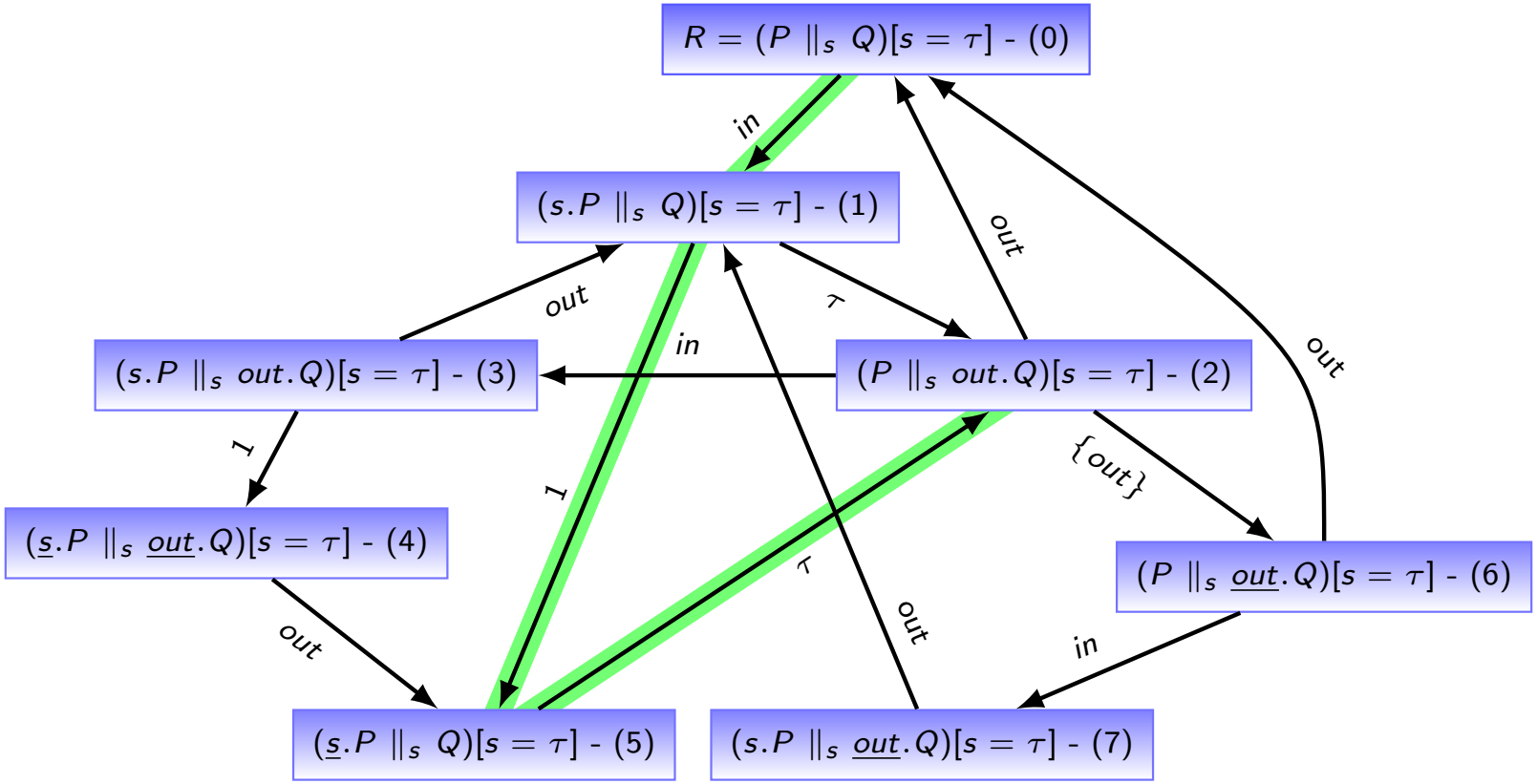
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1

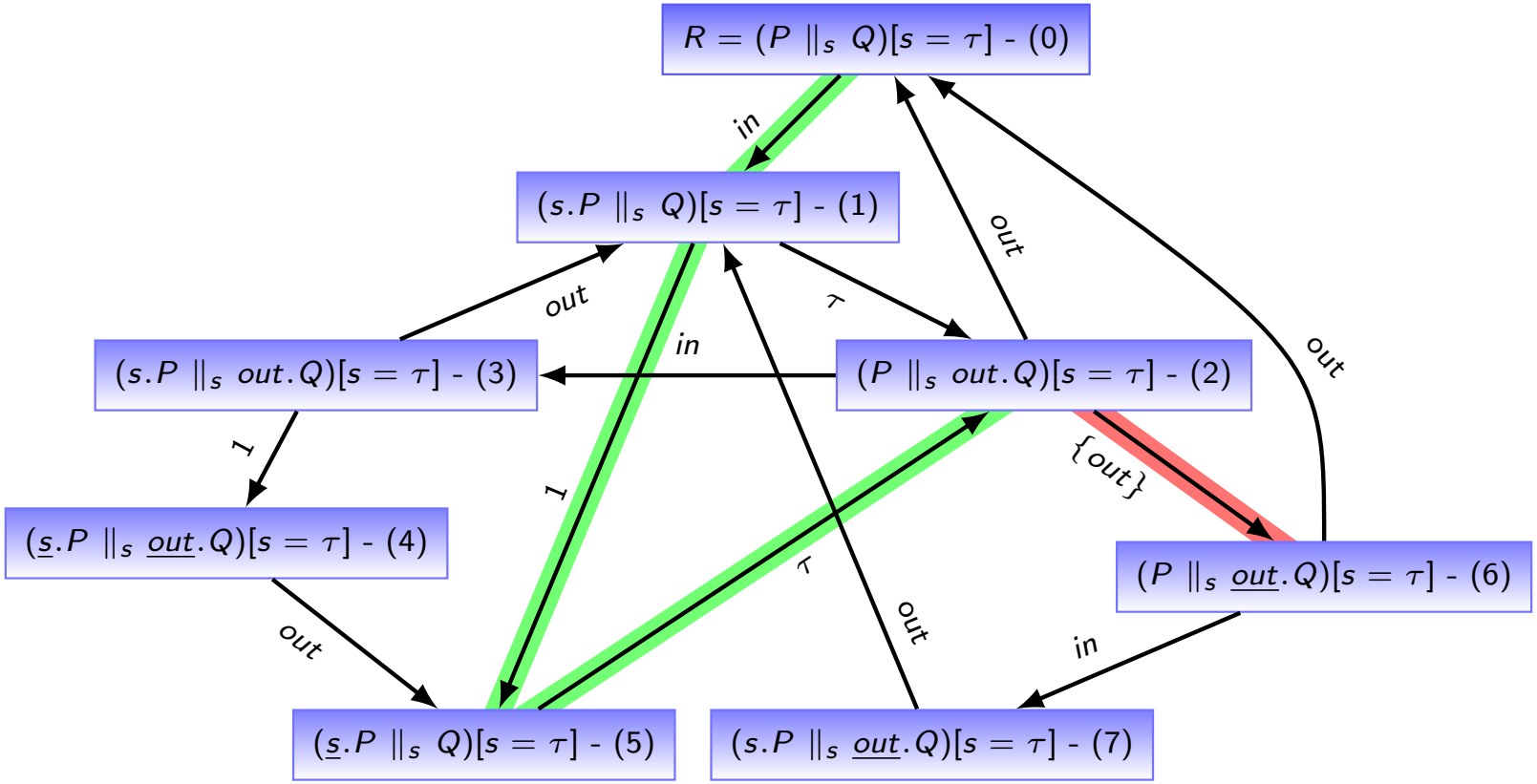
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1

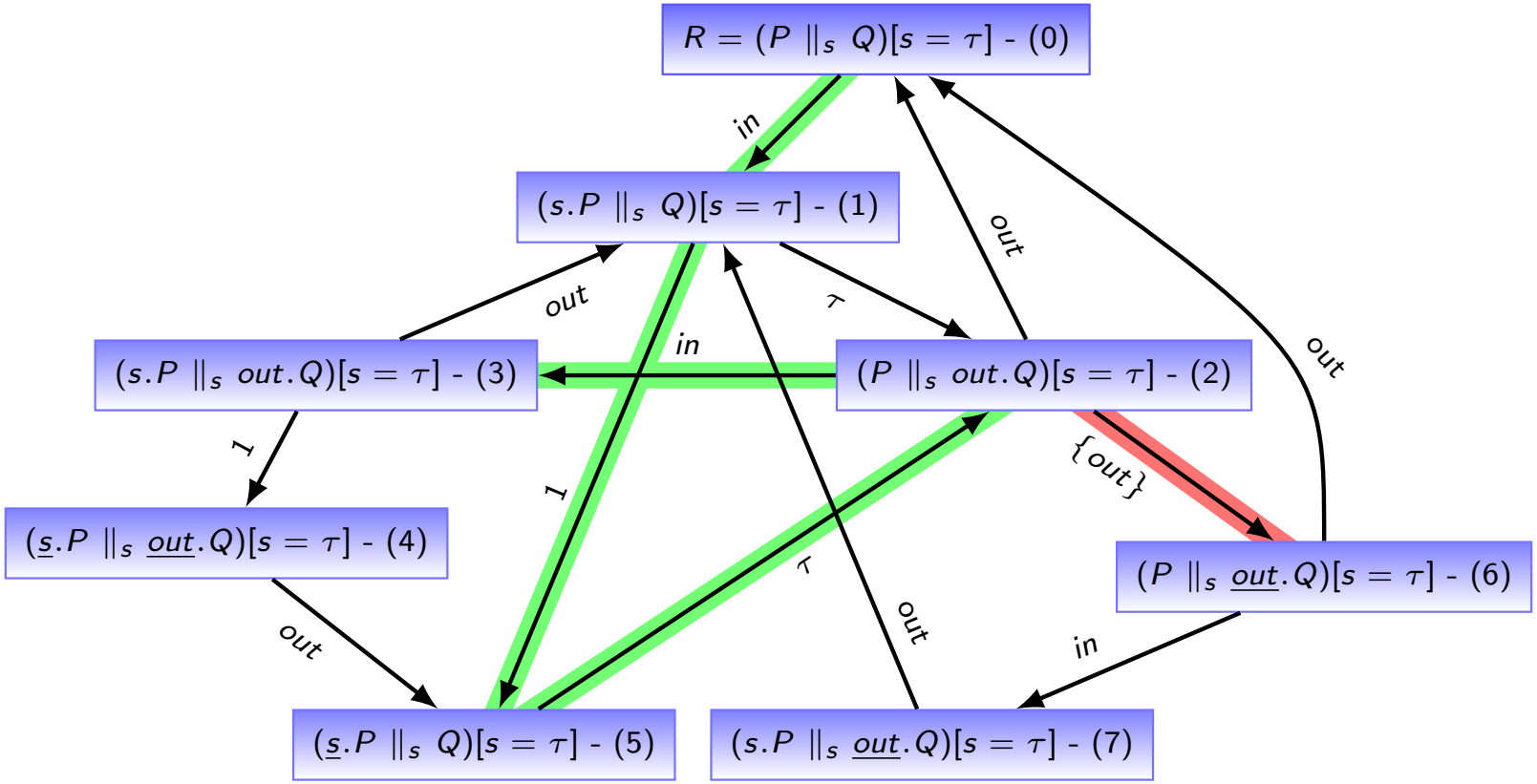
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1

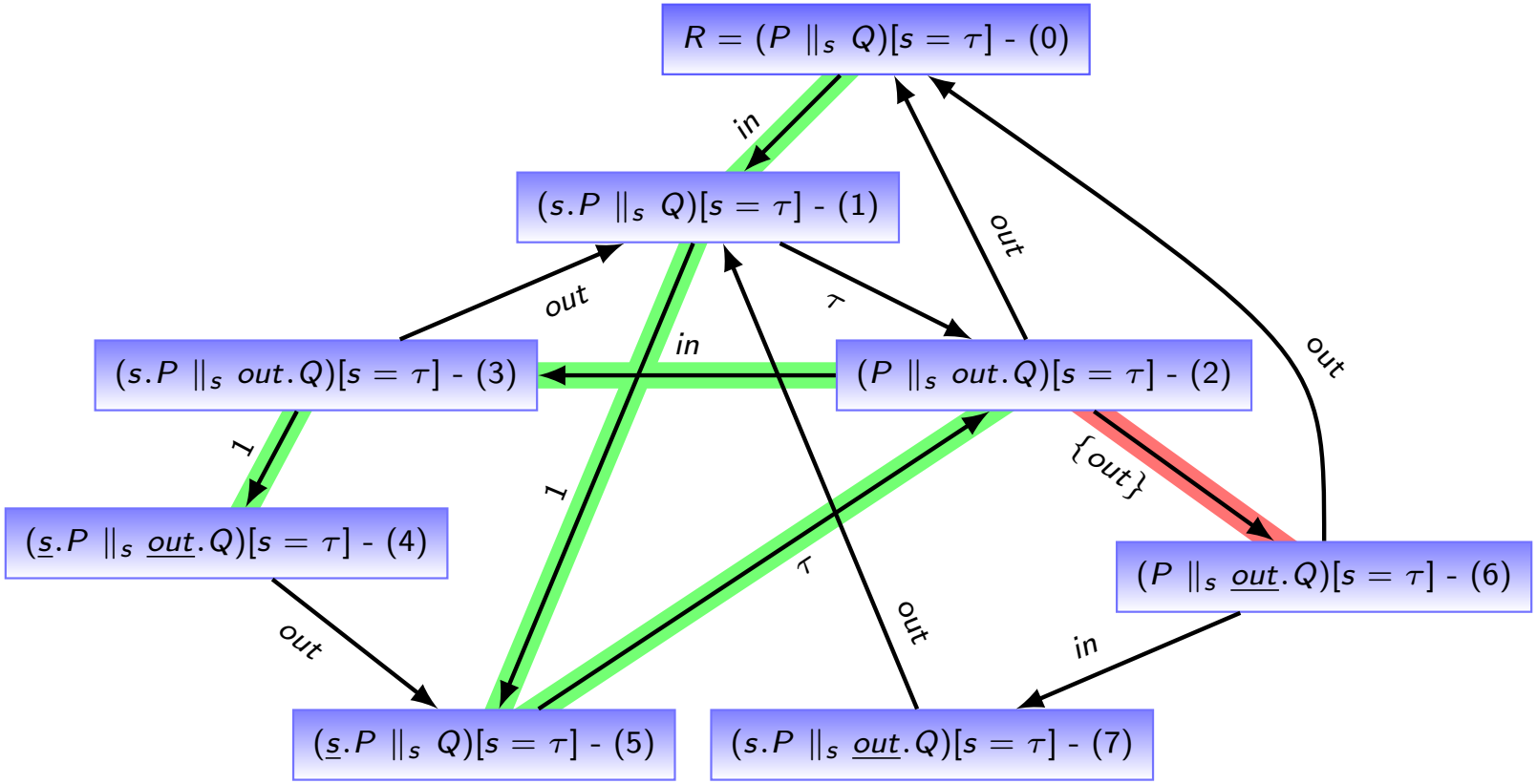
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1

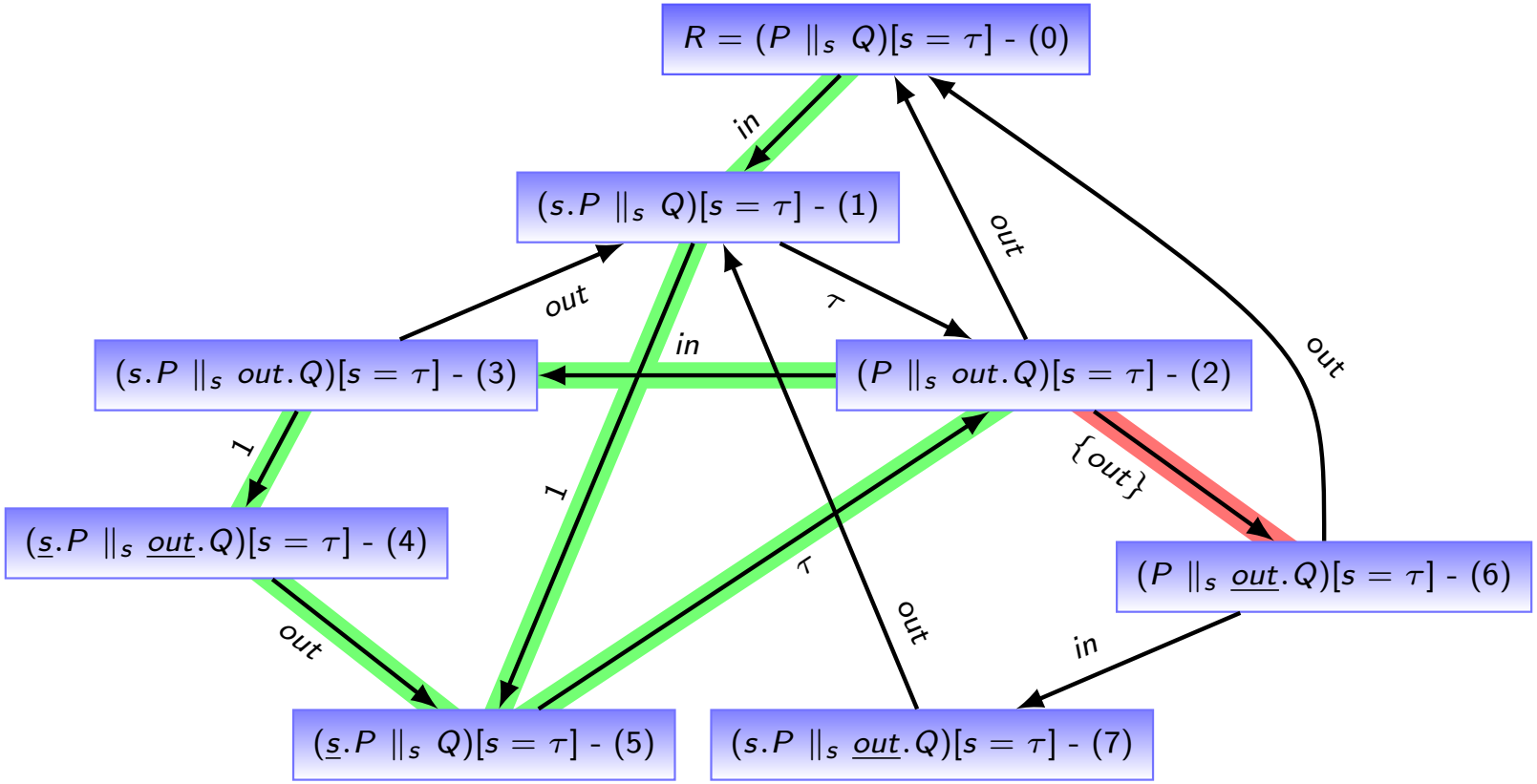
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1 1

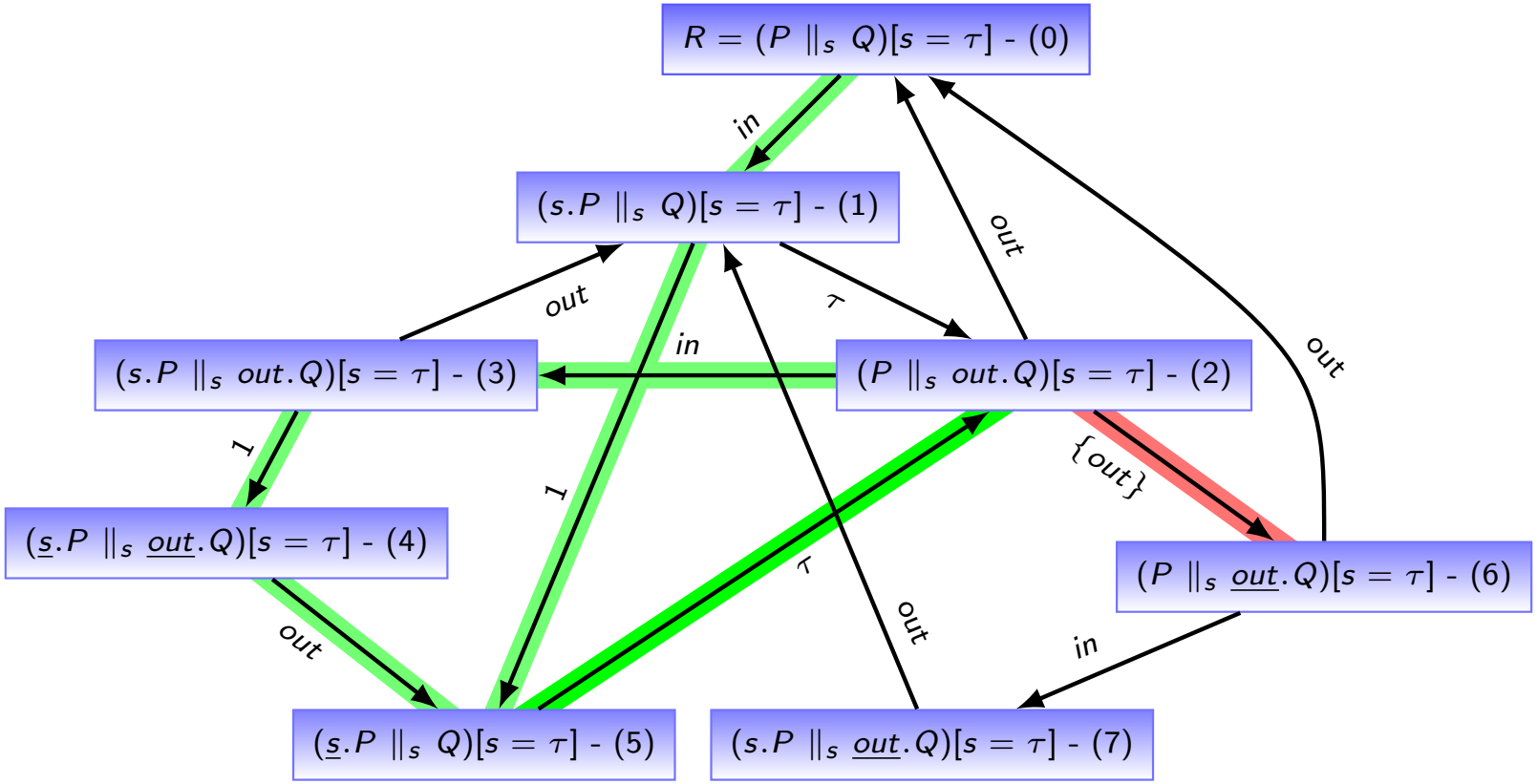
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1 1

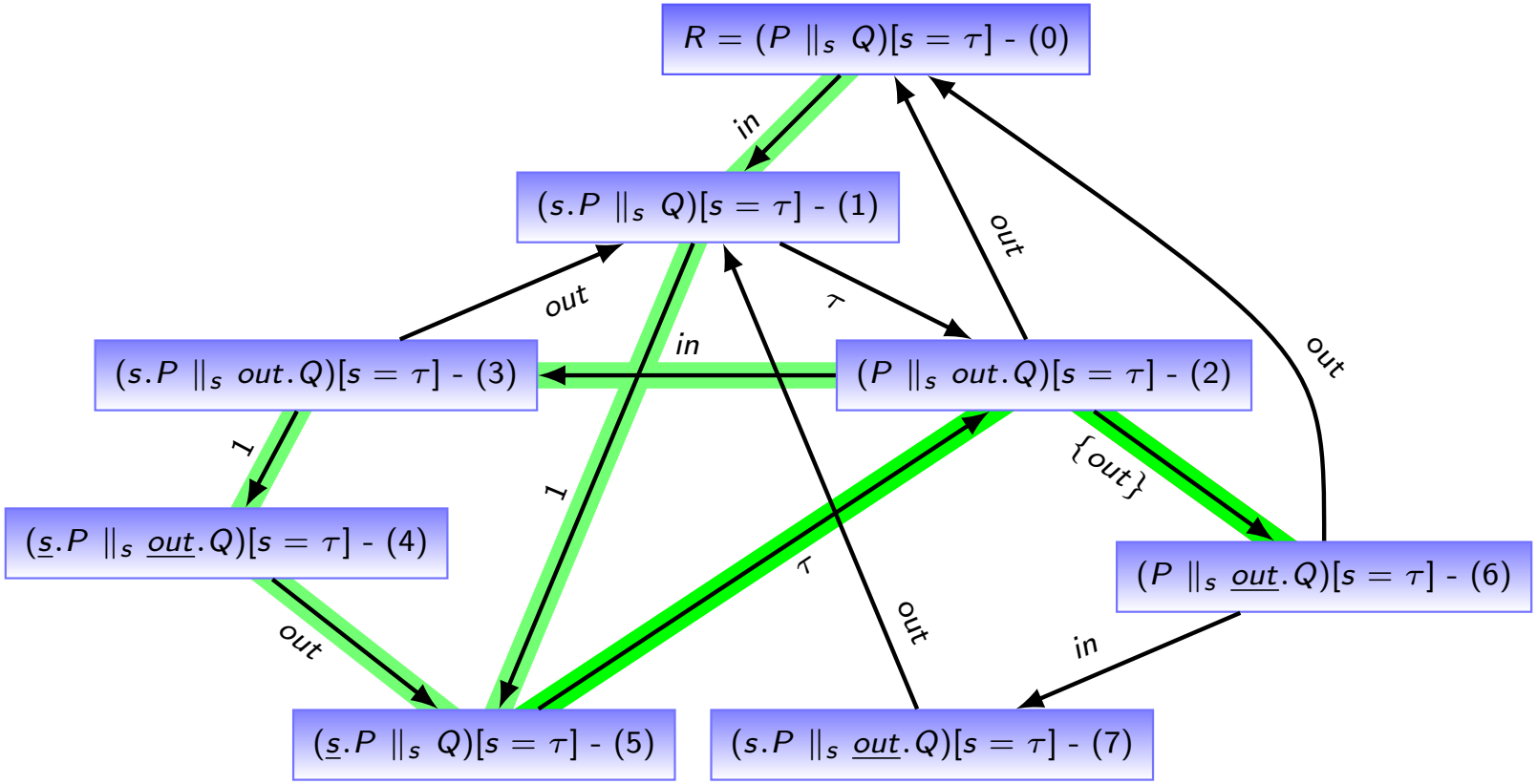
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1 1

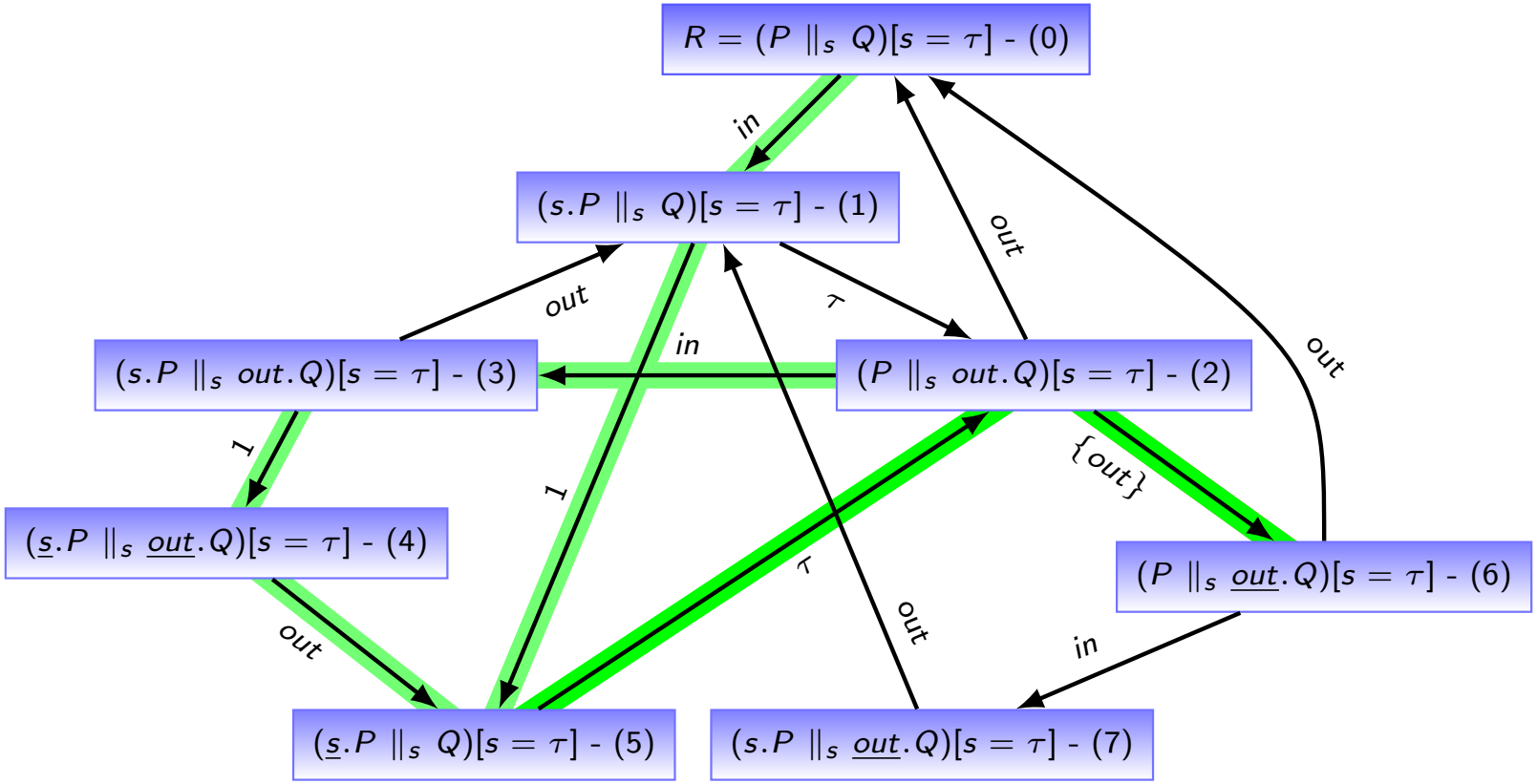
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1 1 1

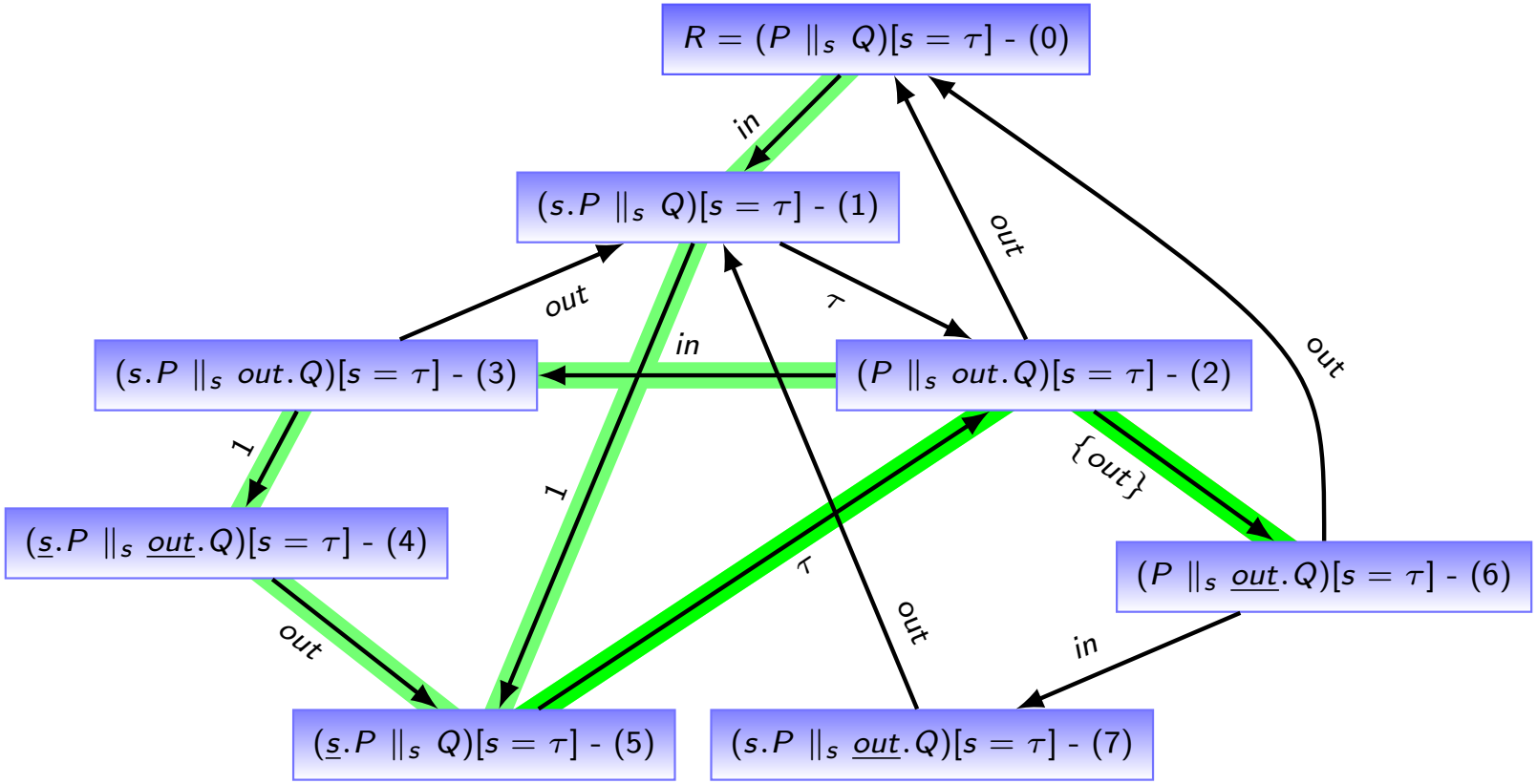
PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = \underline{in}.\ \underline{out}.\ \underline{\omega} \|_\omega \underline{in}.\ \underline{out}.\ \underline{\omega}$
Time: 1 1 1
$rp_{Pipe}(2) = 3$

PIPE - 2-critical example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$U_2 = in.\ out.\ \underline{\omega} \|_\omega in.\ \underline{out}.\ \underline{\omega}$
Time: 1 1 1
$rp_{Pipe}(2) = 3$
It has been shown that $rp_{Pipe}(n) = n + 1$

Bad Cycle(s)

- Since we consider only finite response processes, as requests increase w.r.t. the size of the state space, the only way to satisfy them is entering cycle(s) and rounding them as many times as necessary

Bad Cycle(s)

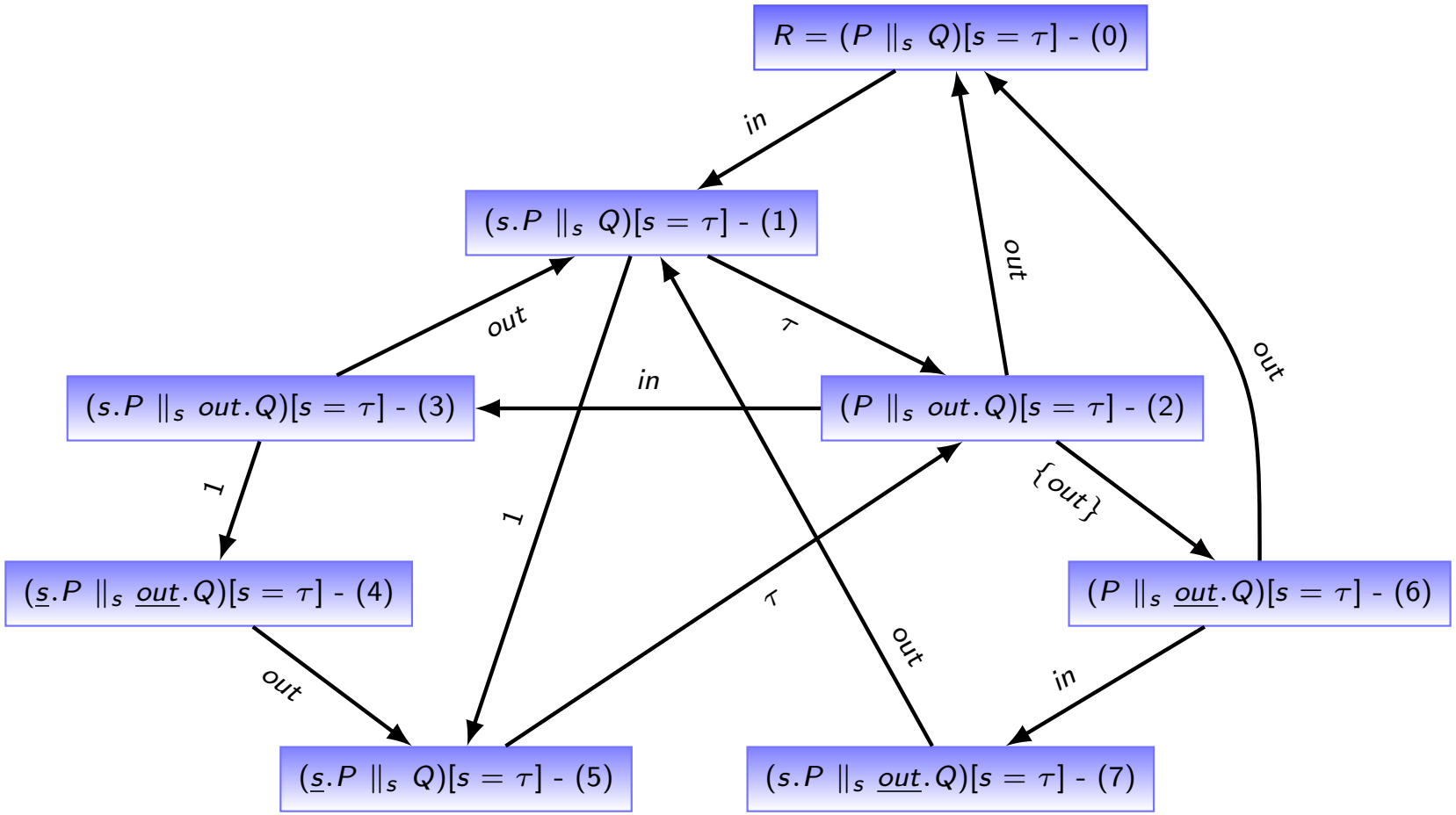
- Since we consider only finite response processes, as requests increase w.r.t. the size of the state space, the only way to satisfy them is entering cycle(s) and rounding them as many times as necessary

Average performance

$$\frac{|FTS|}{|in|}$$

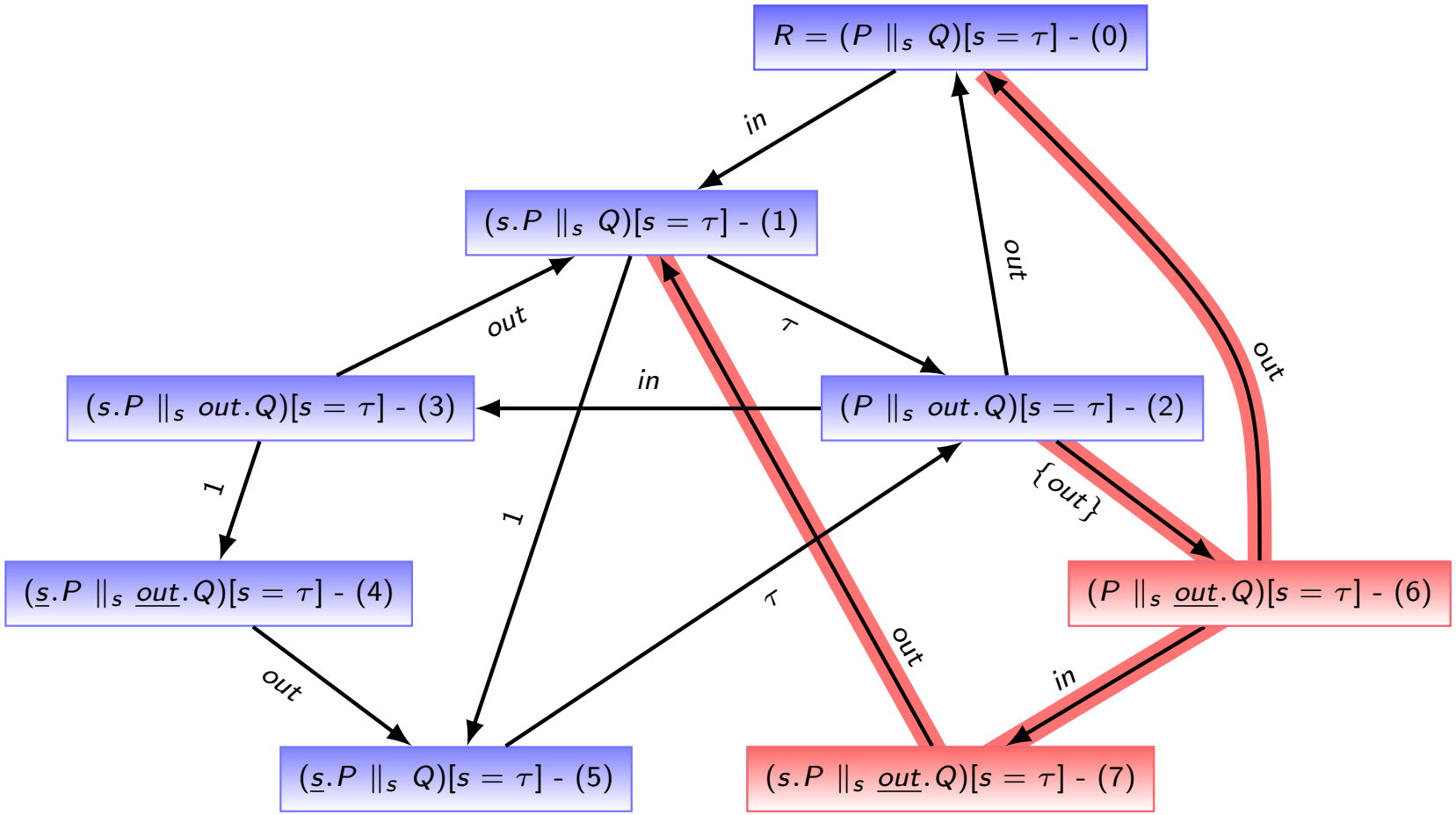
- The cycle(s) with the *worst average cost*, i.e. the cycle(s) with the **maximum ratio** between the number of its (their) full time steps and *in* actions, is (are) called **bad cycle(s)**
- The average cost of the bad cycle(s) corresponds to the *asymptotic factor* of the response performance

PIPE - Bad cycle(s) old algorithm



$$R = (P|s|Q)[s = \tau];$$
$$P = \underline{in}.s.P;$$
$$Q = \underline{s}.out.Q;$$

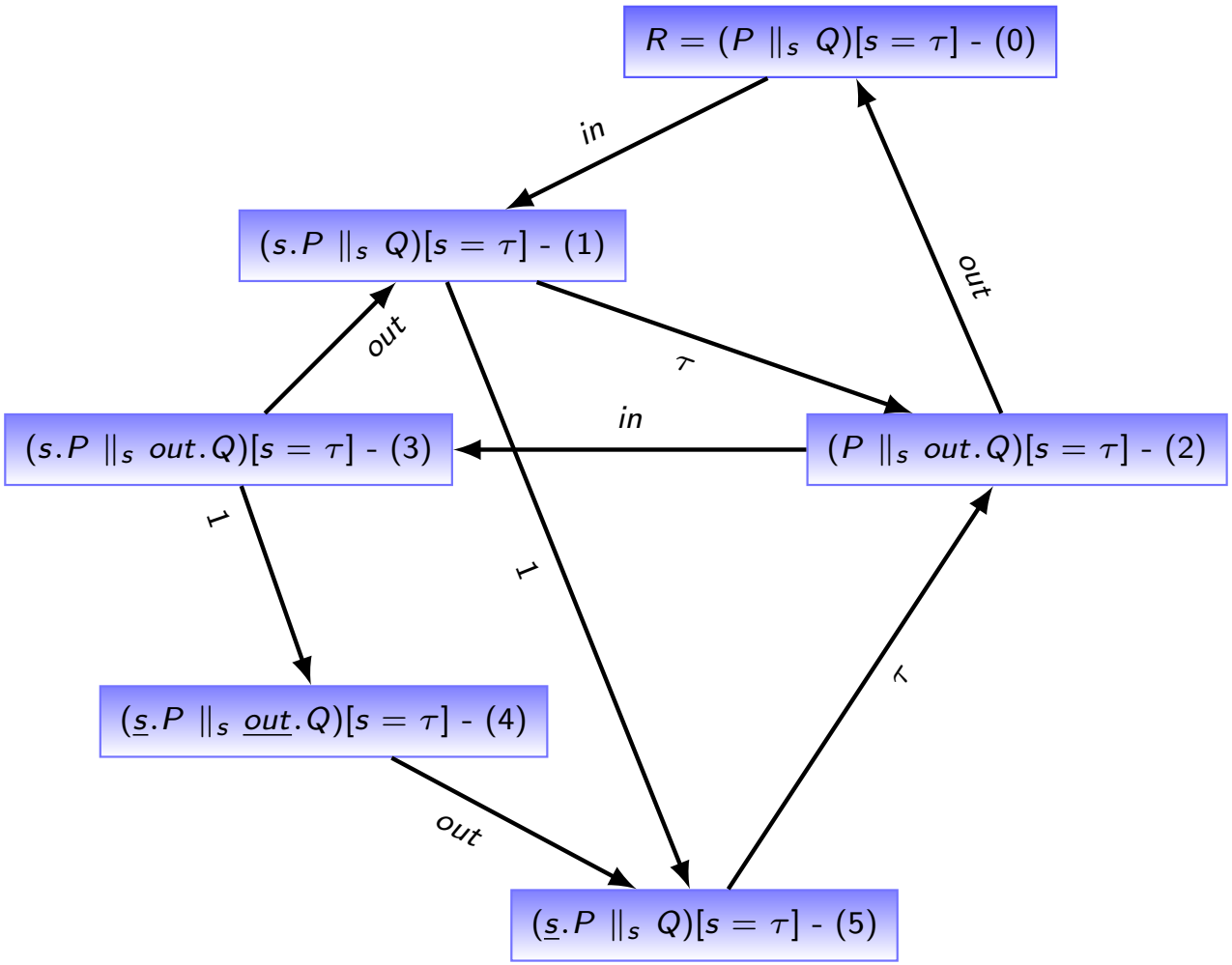
PIPE - Bad cycle(s) old algorithm



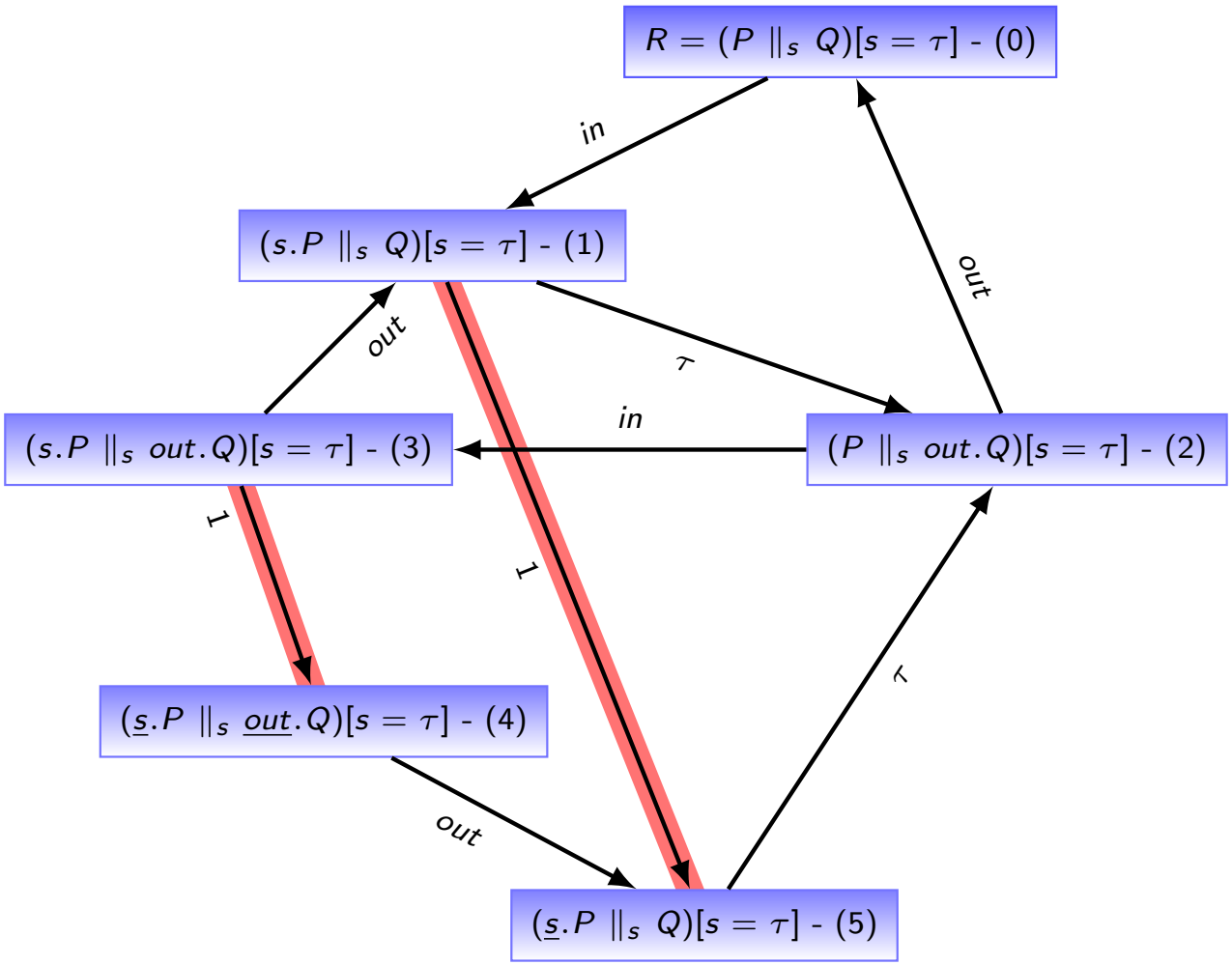
$$R = (P|s|Q)[s = \tau];$$
$$P = \underline{in}.s.P;$$
$$Q = \underline{s}.out.Q;$$

PIPE - Bad cycle(s) old algorithm

$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$



PIPE - Bad cycle(s) old algorithm



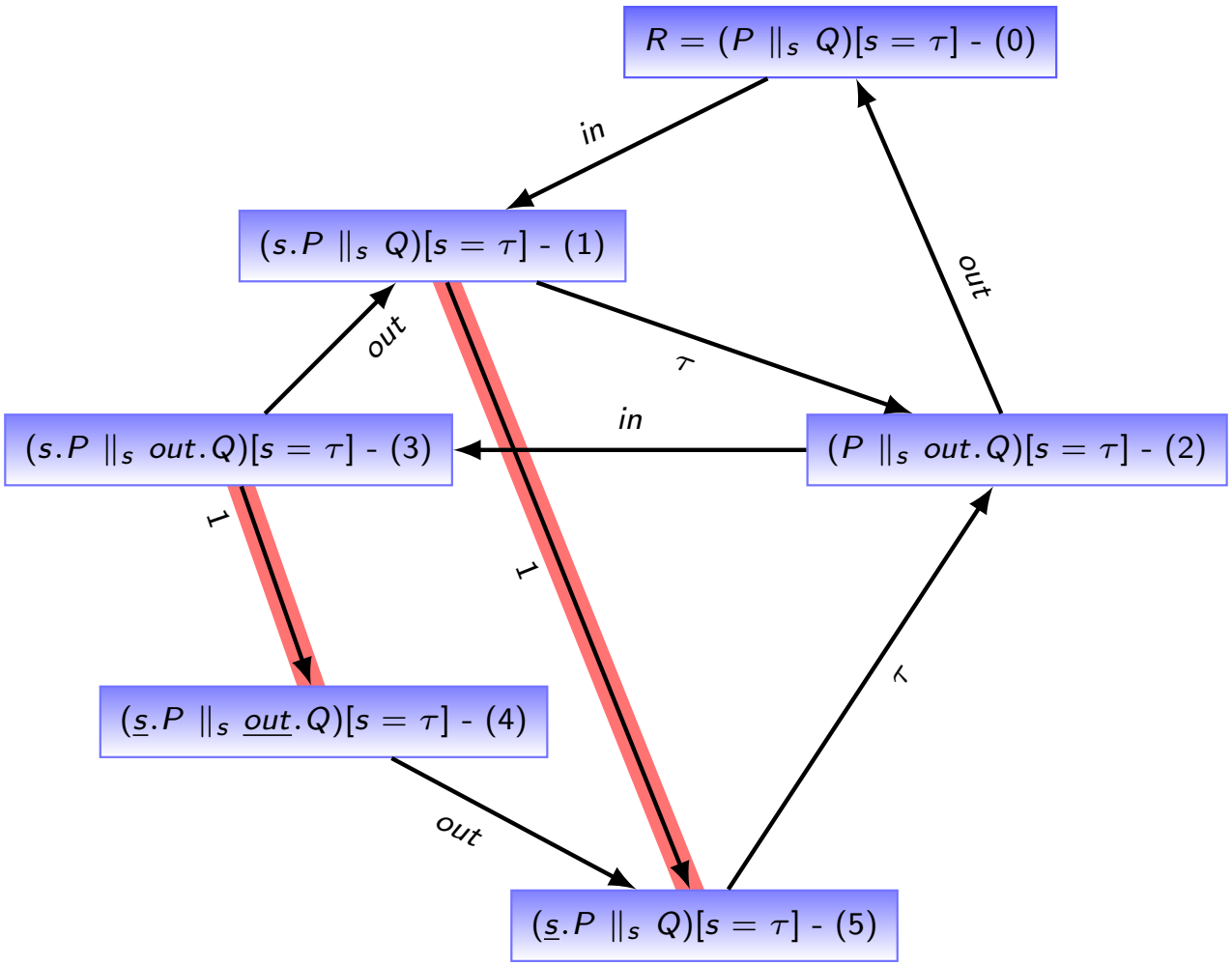
$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

Algorithm steps:

1. Temporal edge removal
2. Calculation of Floyd-Warshall shortest paths matrix
   1 - *in*'s
   0 - otherwise

	0	1	2	3	4	5
0	1	1	1	2	∞	∞
1	0	1	0	1	∞	∞
2	0	1	1	1	∞	∞
3	0	0	0	1	∞	∞
4	0	1	0	1	∞	0
5	0	1	0	1	∞	∞

PIPE - Bad cycle(s) old algorithm



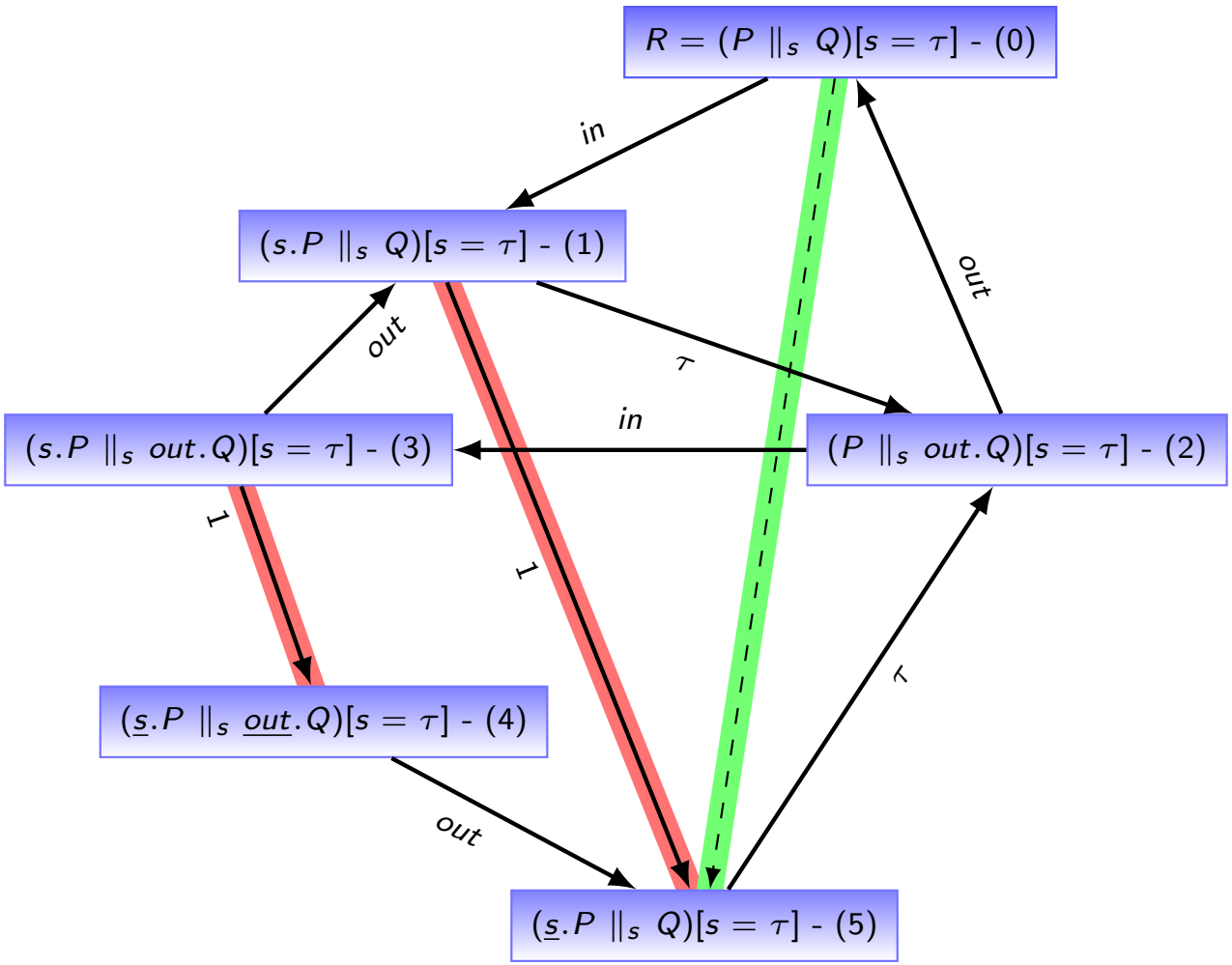
We use the created paths matrix to generate a new graph

For every finite distance:

- We add an edge between the source and the target of a following FTS (if available)
- The edge weight is equal to the corresponding matrix path value

	0	1	2	3	4	5
0	1	1	1	2	∞	∞
1	0	1	0	1	∞	∞
2	0	1	1	1	∞	∞
3	0	0	0	1	∞	∞
4	0	1	0	1	∞	0
5	0	1	0	1	∞	∞

PIPE - Bad cycle(s) old algorithm



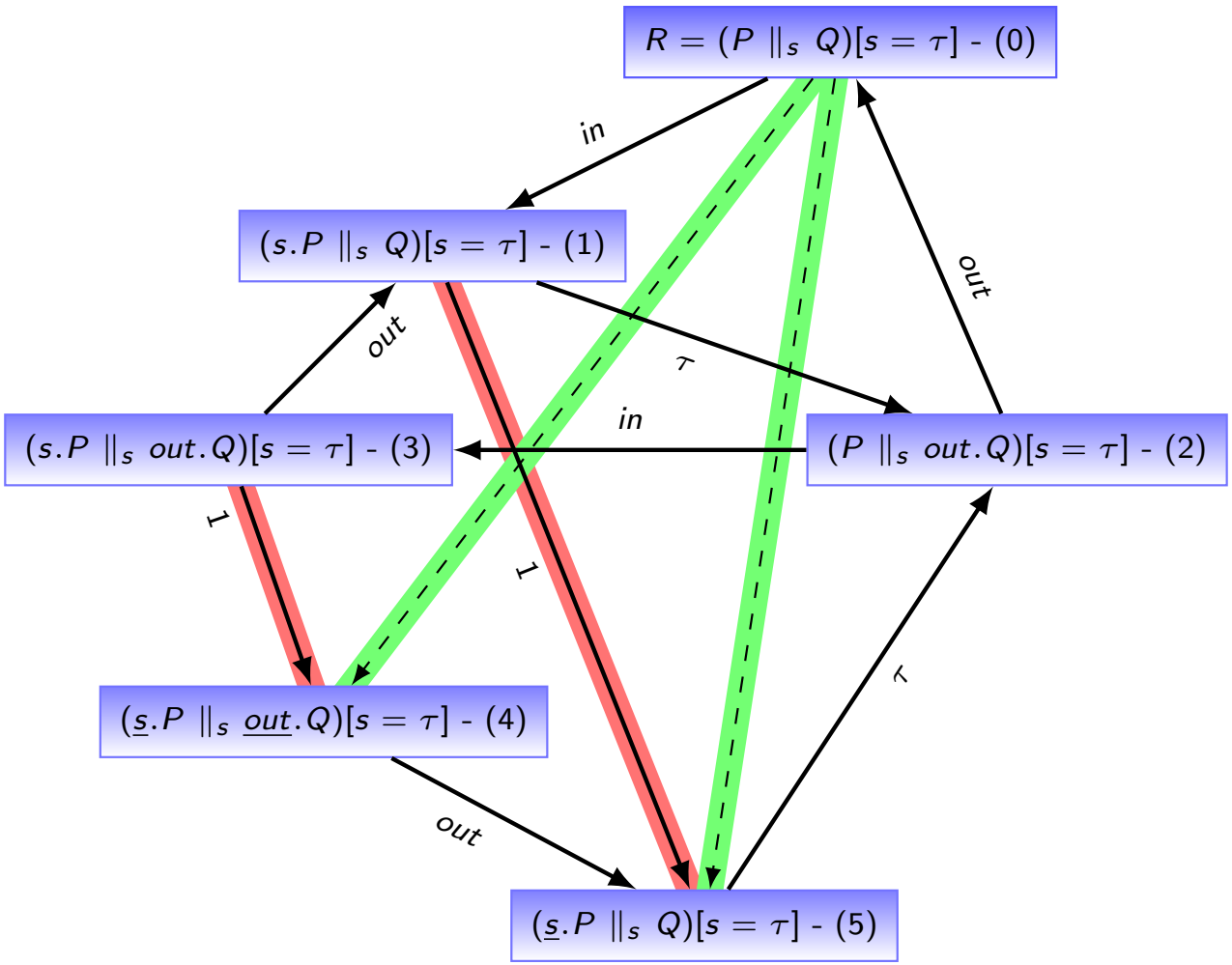
We use the created paths matrix to generate a new graph

For every finite distance:

- We add an edge between the source and the target of a following FTS (if available)
- The edge weight is equal to the corresponding matrix path value

	0	1	2	3	4	5
0	1	1	1	2	∞	∞
1	0	1	0	1	∞	∞
2	0	1	1	1	∞	∞
3	0	0	0	1	∞	∞
4	0	1	0	1	∞	0
5	0	1	0	1	∞	∞

PIPE - Bad cycle(s) old algorithm



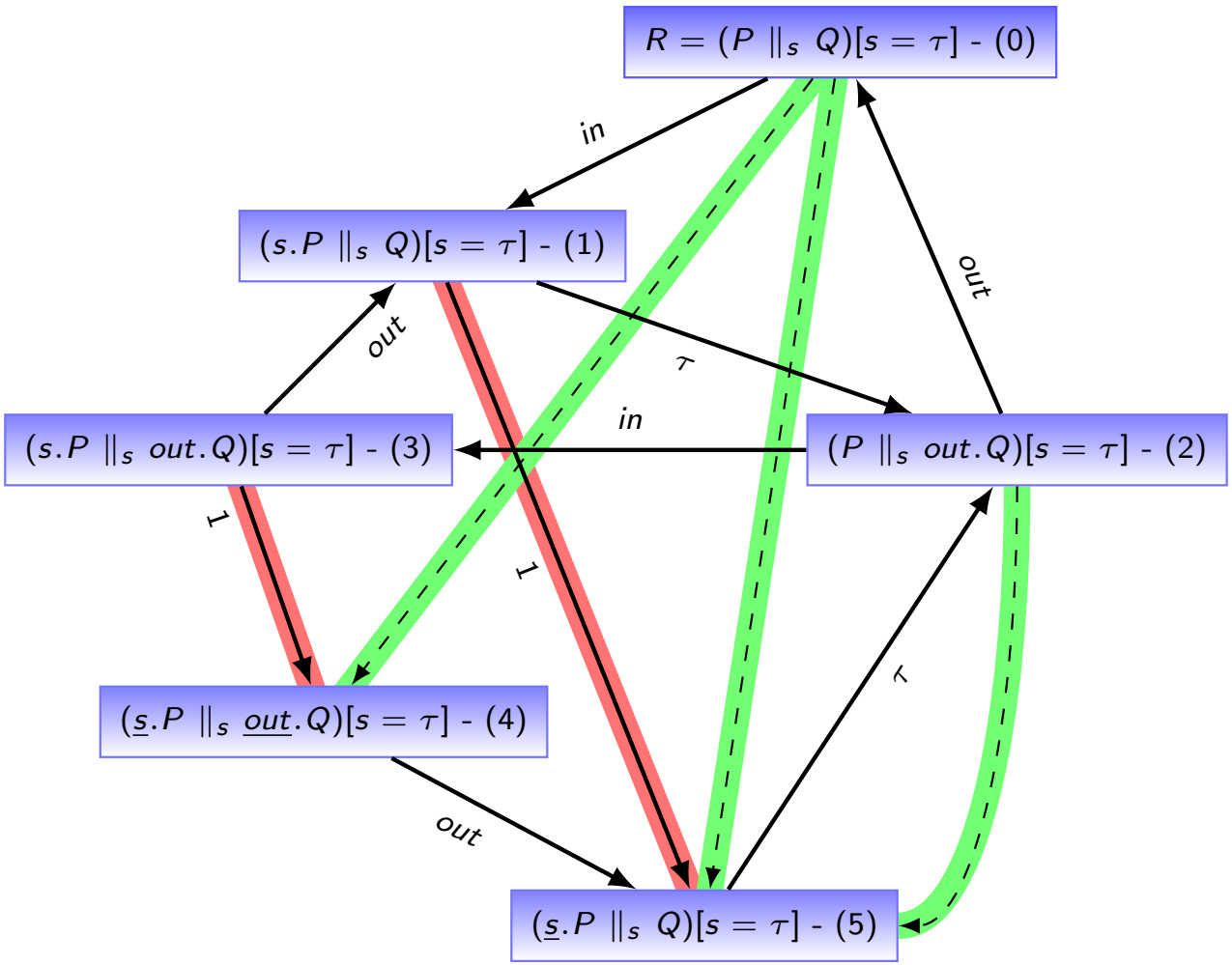
We use the created paths matrix to generate a new graph

For every finite distance:

- We add an edge between the source and the target of a following FTS (if available)
- The edge weight is equal to the corresponding matrix path value

	0	1	2	3	4	5
0	1	1	1	2	∞	∞
1	0	1	0	1	∞	∞
2	0	1	1	1	∞	∞
3	0	0	0	1	∞	∞
4	0	1	0	1	∞	0
5	0	1	0	1	∞	∞

PIPE - Bad cycle(s) old algorithm



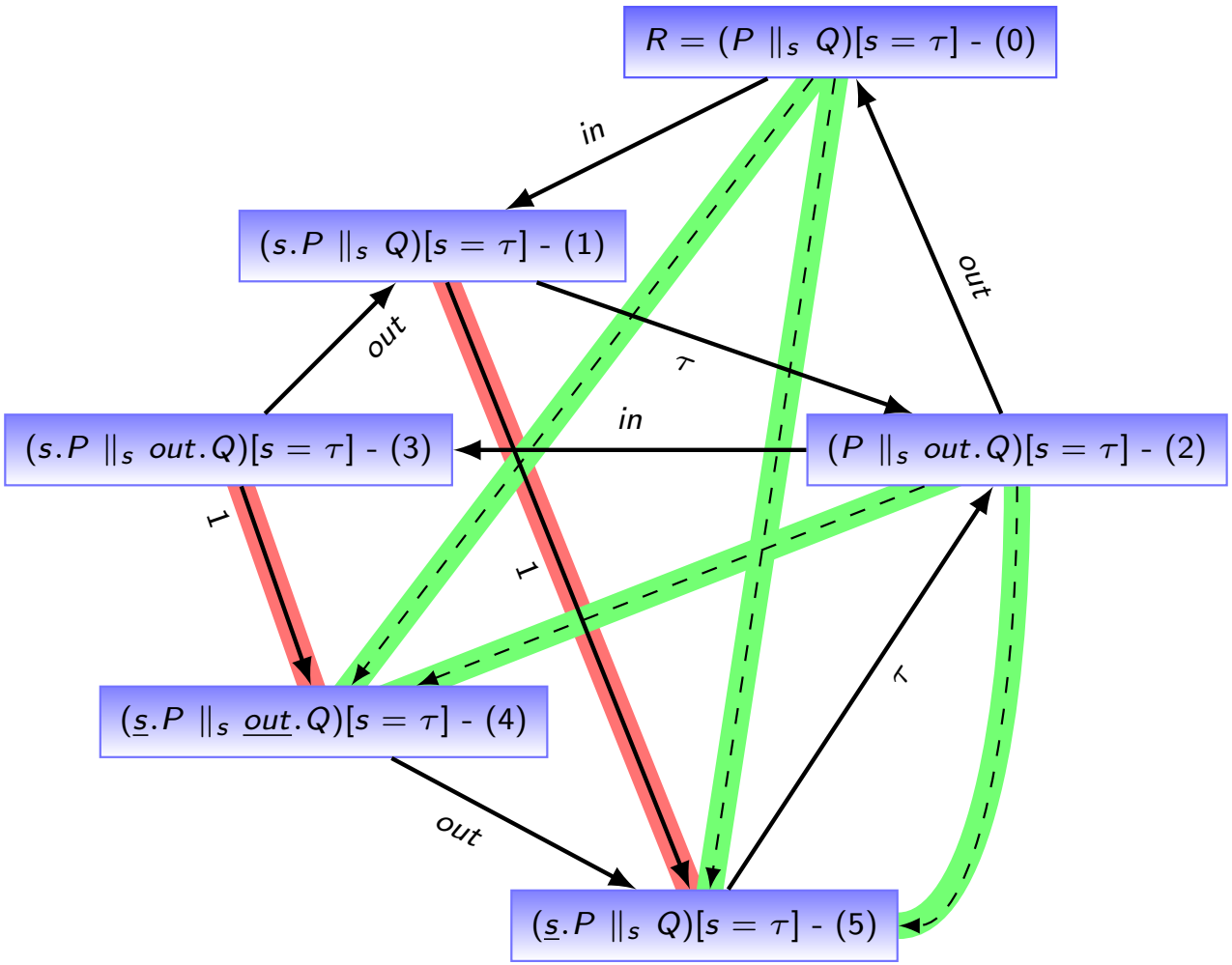
We use the created paths matrix to generate a new graph

For every finite distance:

- We add an edge between the source and the target of a following FTS (if available)
- The edge weight is equal to the corresponding matrix path value

	0	1	2	3	4	5
0	1	1	1	2	∞	∞
1	0	1	0	1	∞	∞
2	0	1	1	1	∞	∞
3	0	0	0	1	∞	∞
4	0	1	0	1	∞	0
5	0	1	0	1	∞	∞

PIPE - Bad cycle(s) old algorithm



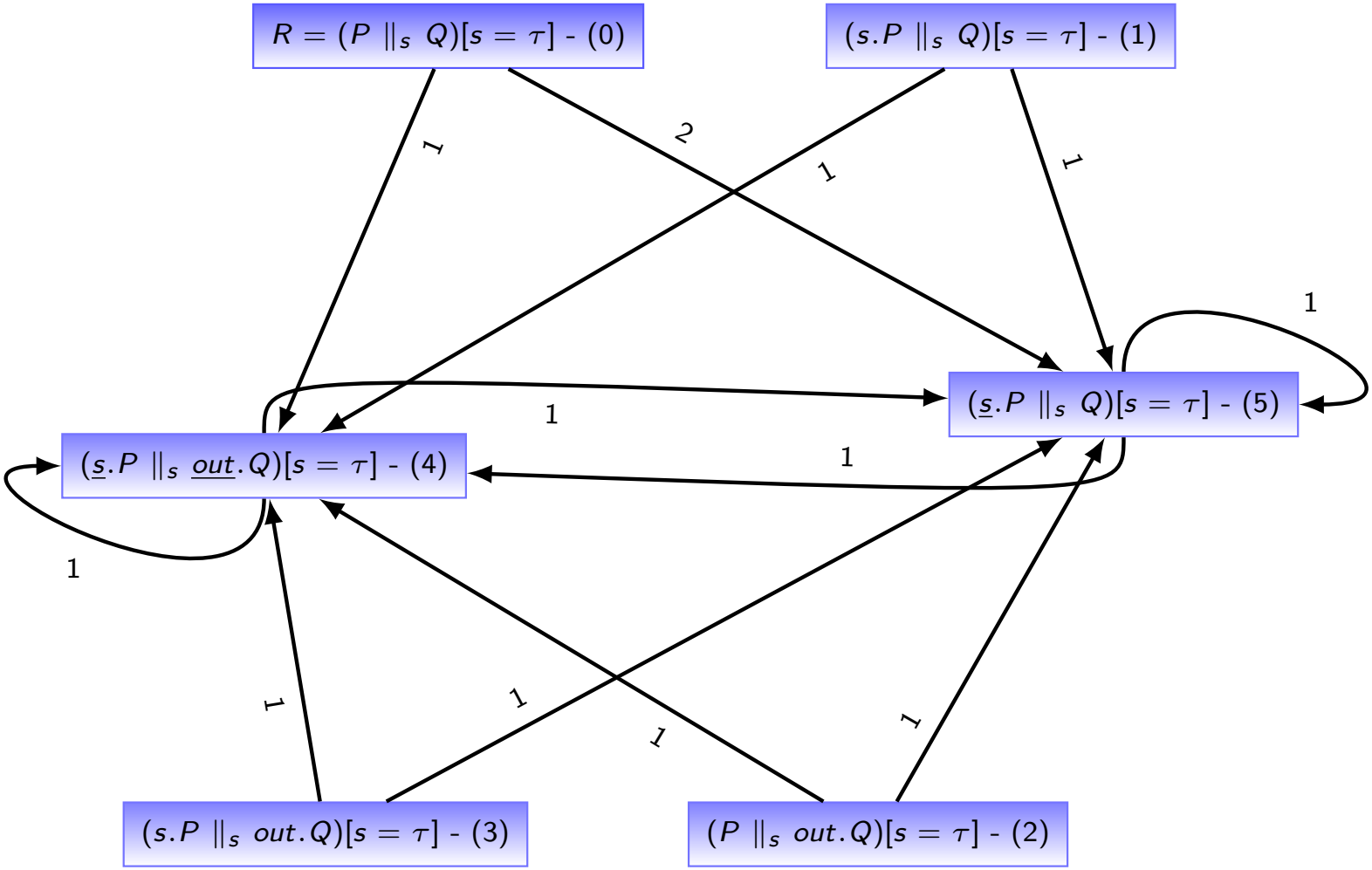
We use the created paths matrix to generate a new graph

For every finite distance:

- We add an edge between the source and the target of a following FTS (if available)
- The edge weight is equal to the corresponding matrix path value

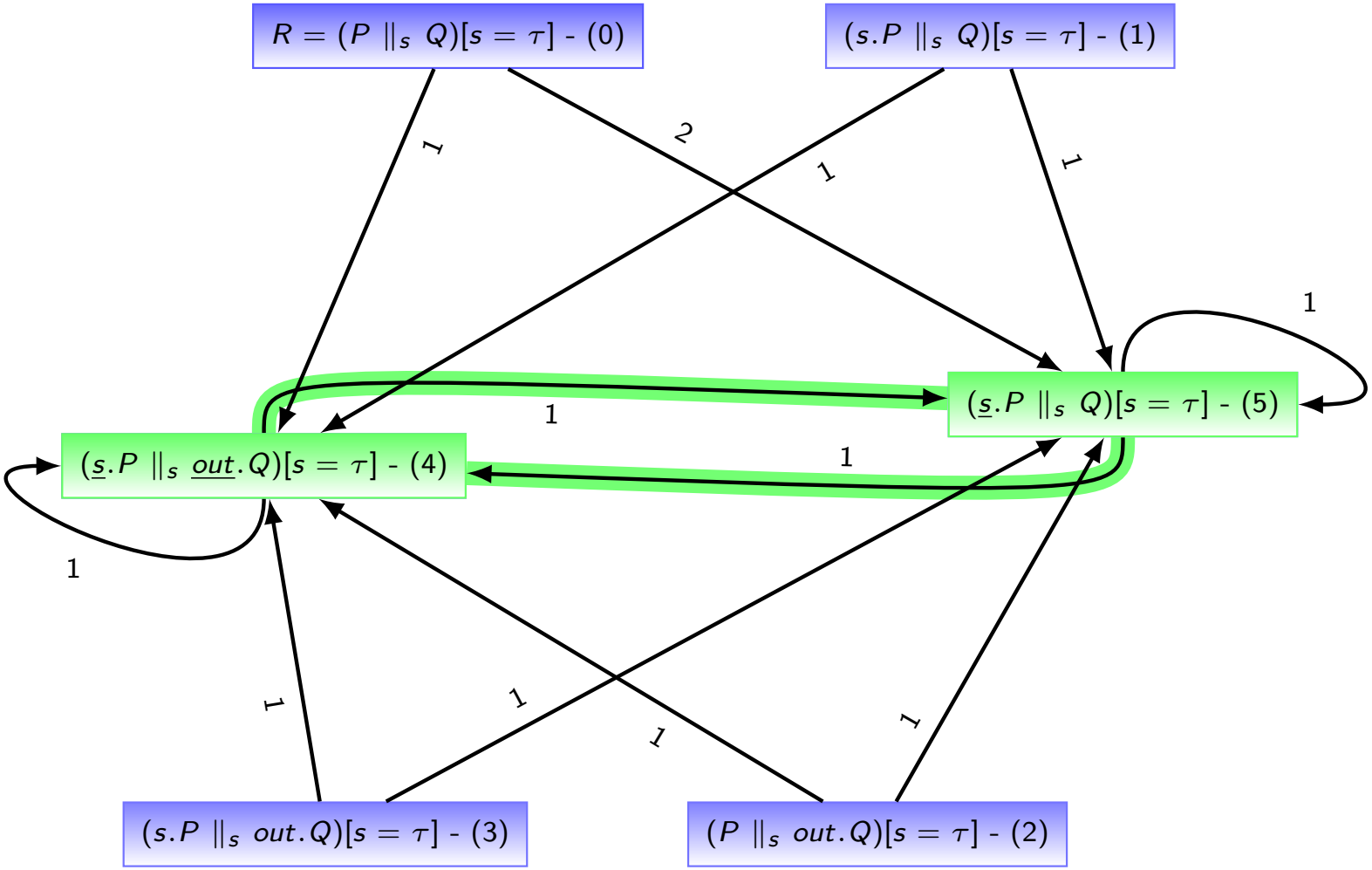
	0	1	2	3	4	5
0	1	1	1	2	∞	∞
1	0	1	0	1	∞	∞
2	0	1	1	1	∞	∞
3	0	0	0	1	∞	∞
4	0	1	0	1	∞	0
5	0	1	0	1	∞	∞

PIPE - Bad cycle(s) calculation graph



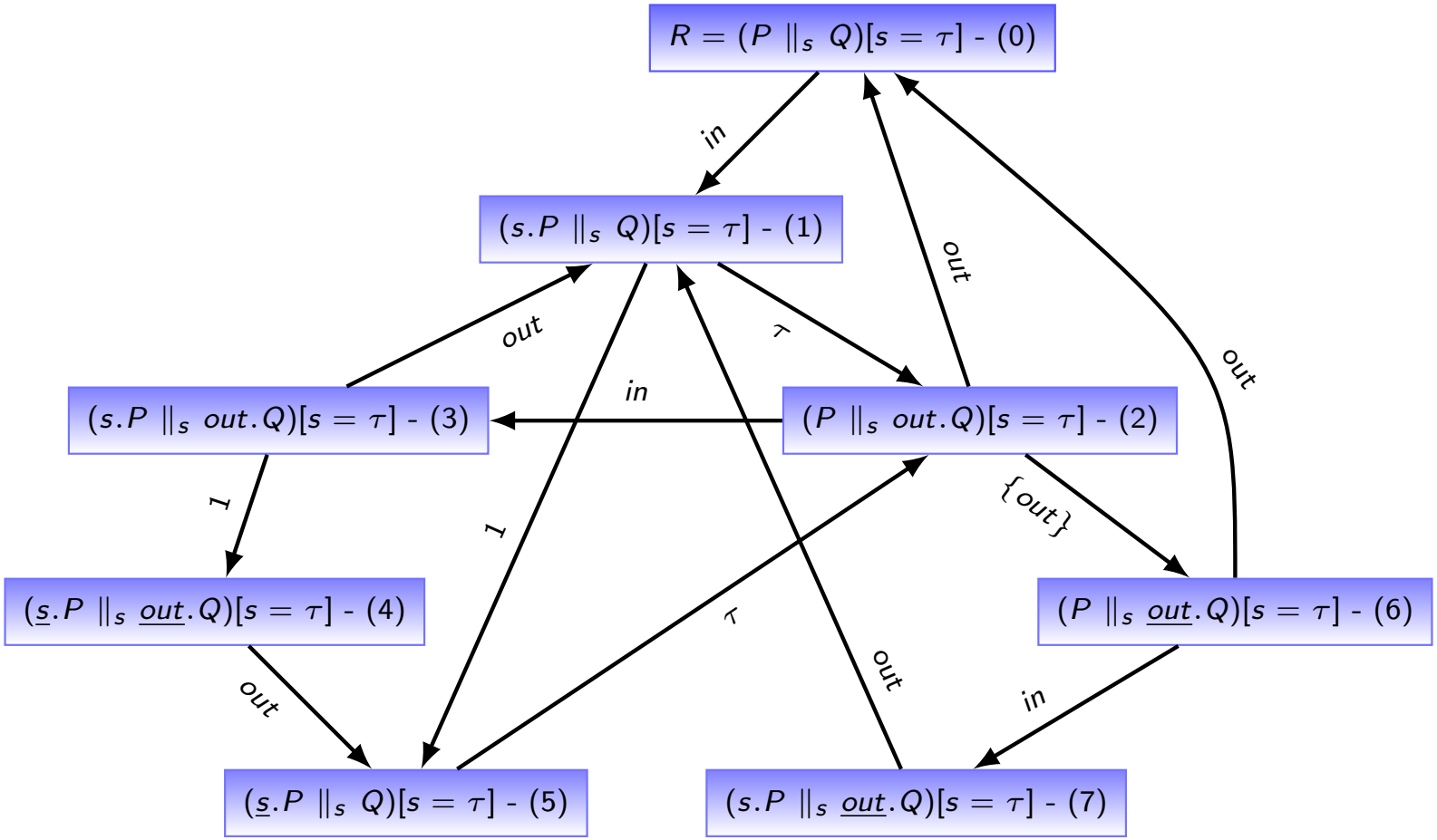
On the obtained graph we apply the **Karp algorithm** for finding the minimum mean cycle

PIPE - Bad cycle(s) calculation graph



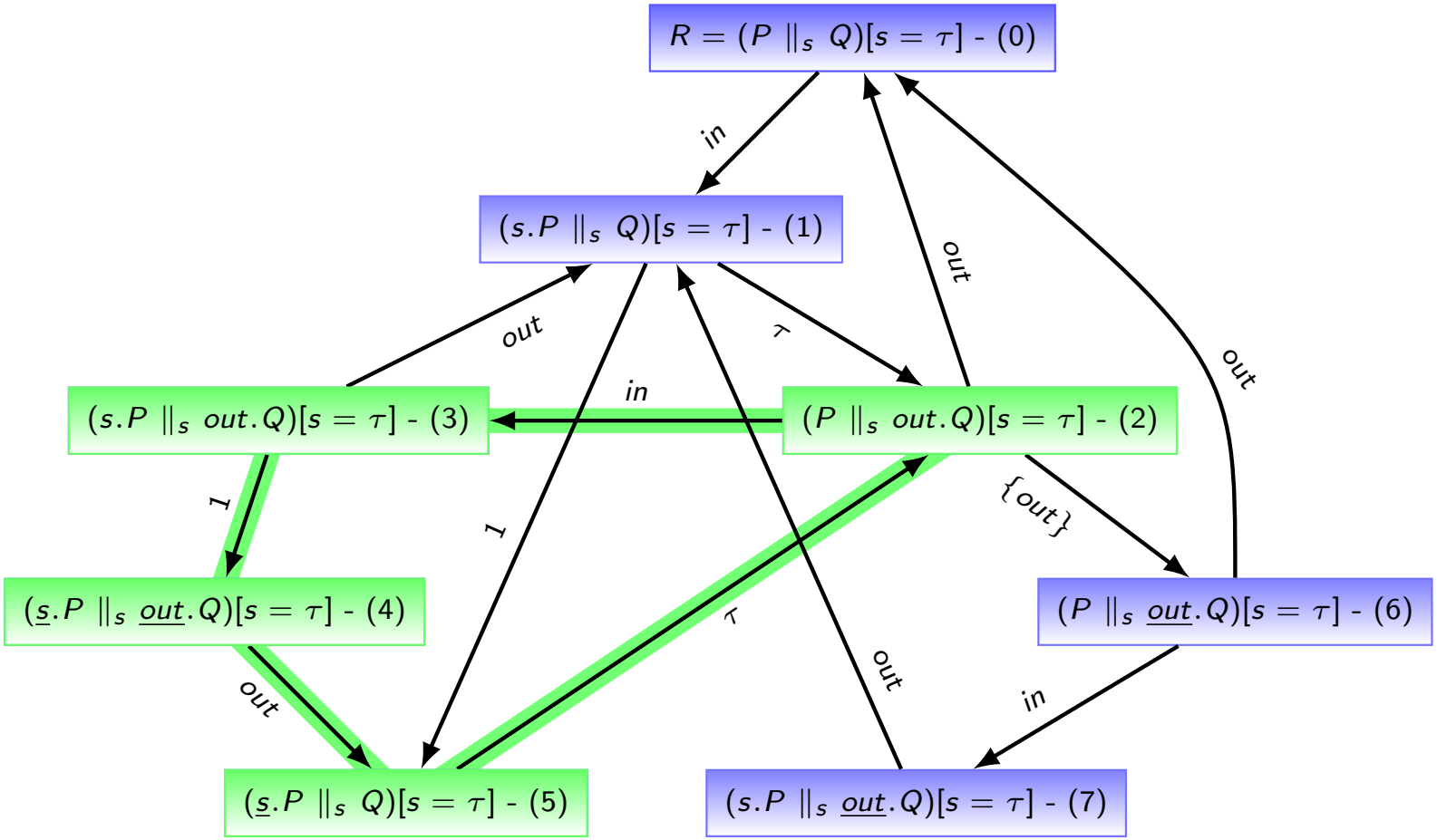
On the obtained graph we apply the **Karp algorithm** for finding the minimum mean cycle

PIPE - Bad cycle(s) example - final result



$$R = (P|s|Q)[s = \tau];$$
$$P = \underline{in}.s.P;$$
$$Q = \underline{s}.out.Q;$$

PIPE - Bad cycle(s) example - final result



$$R = (P|s|Q)[s = \tau];$$
$$P = \underline{in}.s.P;$$
$$Q = \underline{s}.out.Q;$$

Bad cycle(s) algorithms running time

- The current algorithm has a complexity of $\mathbf{\Theta(N^3) + \Theta(NE)}$
- Our *new* solution is based on the **Bellman-Ford** single-source shortest paths algorithm

Bad cycle(s) algorithms running time

- The current algorithm has a complexity of $\mathbf{\Theta(N^3) + \Theta(NE)}$
- Our *new* solution is based on the **Bellman-Ford** single-source shortest paths algorithm
- Starting from an SCC of G (G is the TTS obtained from $rRTS(P)$) we have that:

Bad cycle(s) algorithms running time

- The current algorithm has a complexity of $\mathbf{\Theta(N^3) + \Theta(NE)}$
- Our *new* solution is based on the **Bellman-Ford** single-source shortest paths algorithm
- Starting from an SCC of G (G is the TTS obtained from $rRTS(P)$) we have that:

1. $\forall$ time step $(u, v) \in E(G)$, $\exists \rightsquigarrow$ (path) in G s.t. $v \rightsquigarrow u$ with some *in*'s (no CC in P).

   Moreover $|out|$ in $v \rightsquigarrow u$ is equal to $|in|$ of the same path (P is a response process)

Bad cycle(s) algorithms running time

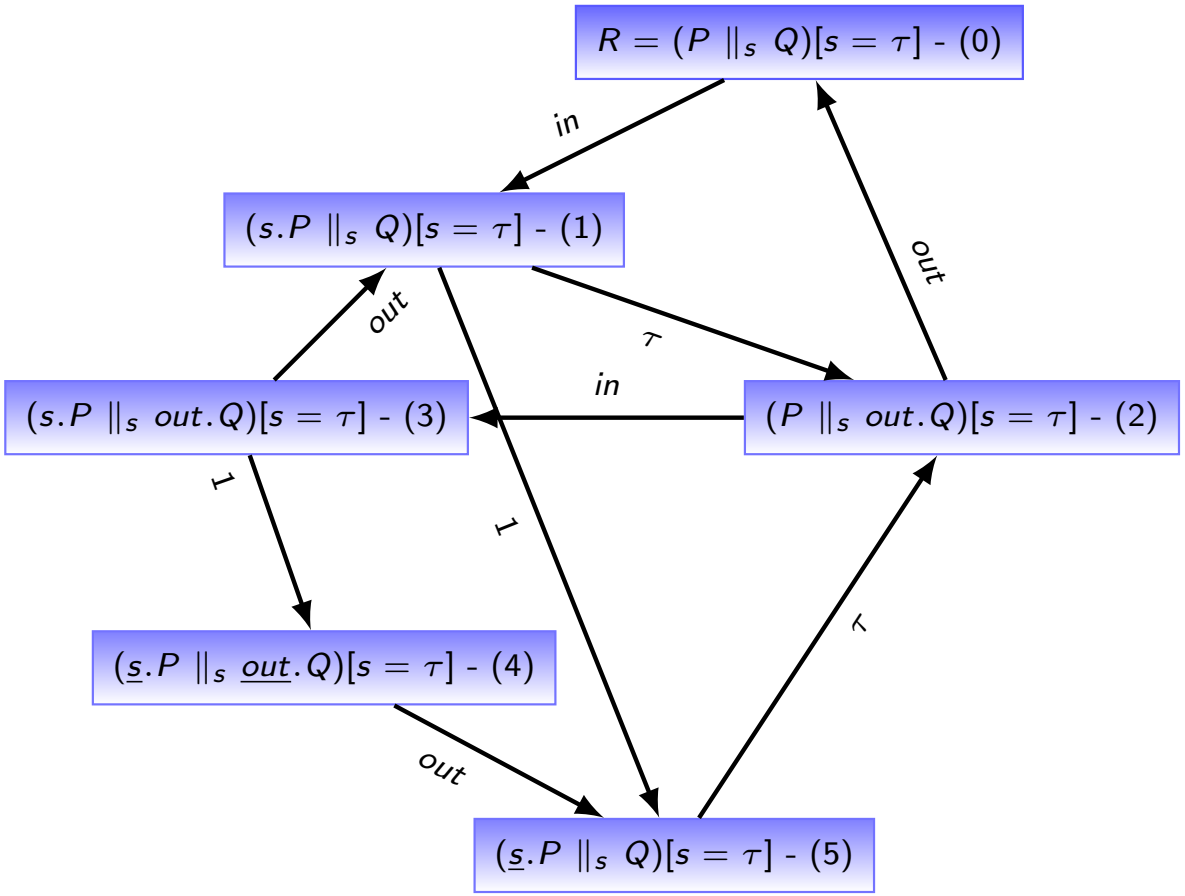
- The current algorithm has a complexity of $\mathbf{\Theta(N^3) + \Theta(NE)}$
- Our *new* solution is based on the **Bellman-Ford** single-source shortest paths algorithm
- Starting from an SCC of G (G is the TTS obtained from $rRTS(P)$) we have that:

1. $\forall$ time step $(u, v) \in E(G)$, $\exists \rightsquigarrow$ (path) in G s.t. $v \rightsquigarrow u$ with some *in*'s (no CC in P).

   Moreover $|out|$ in $v \rightsquigarrow u$ is equal to $|in|$ of the same path (P is a response process)

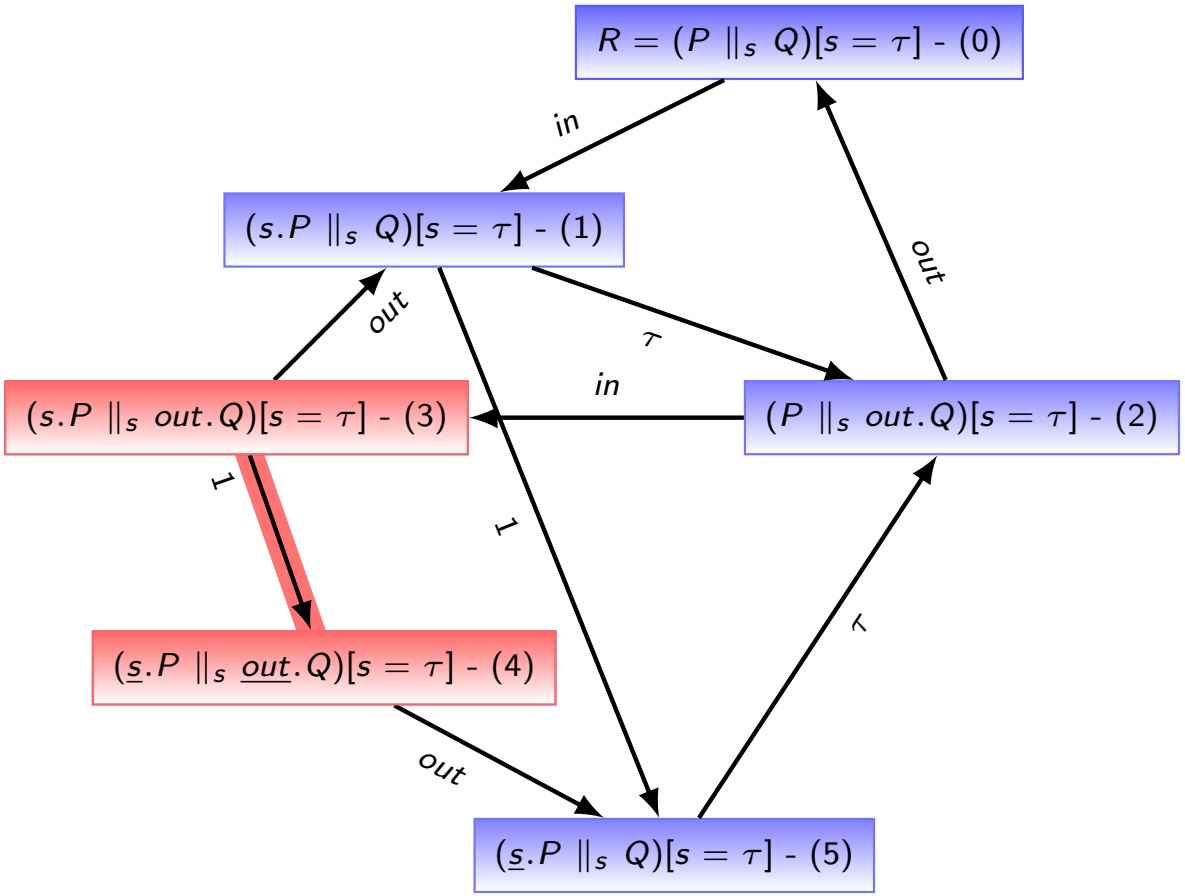
2. We apply *B-F* on each time step (u, v) in G with root v. The weights are:
   - $|N(G)|$ for *in* actions
   - -1 for time steps
   - 0 for *out* and τ actions

PIPE - New Bad cycle(s) example



$$R = (P|s|Q)[s = \tau];$$
$$P = \underline{in}.s.P;$$
$$Q = \underline{s}.out.Q;$$

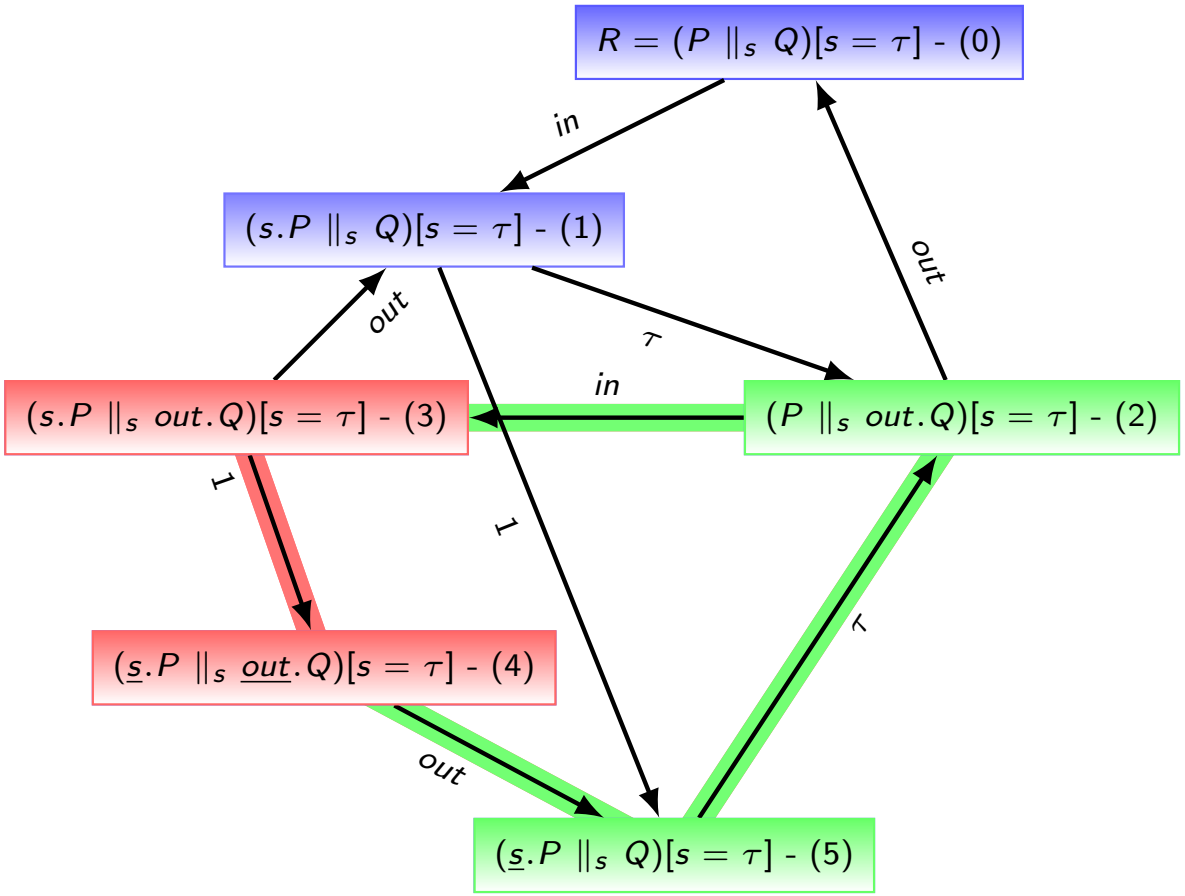
PIPE - New Bad cycle(s) example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$4 \rightsquigarrow 3$

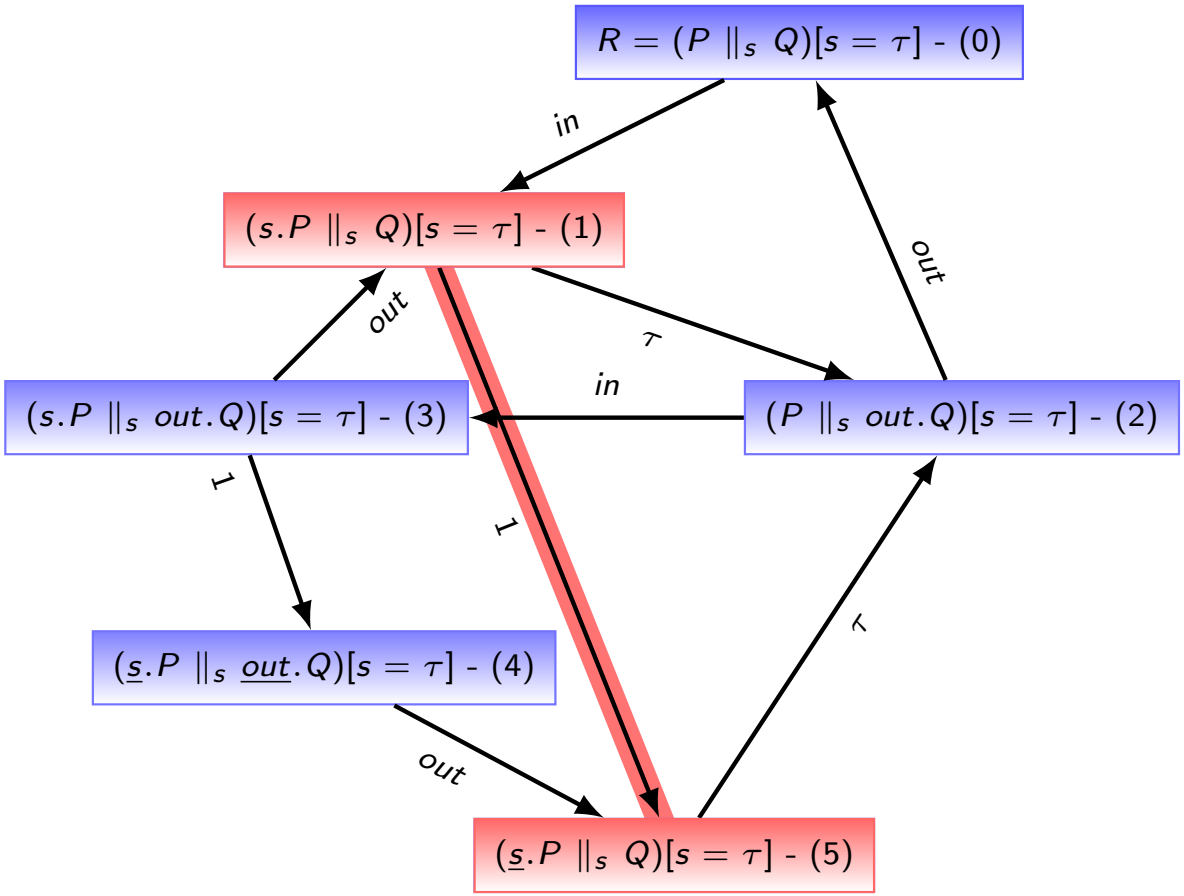
PIPE - New Bad cycle(s) example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$4 \rightsquigarrow 3$
1 *in* - 0 *ts* (+1 due to $(3, 4)$)
average performance $= 1$

PIPE - New Bad cycle(s) example

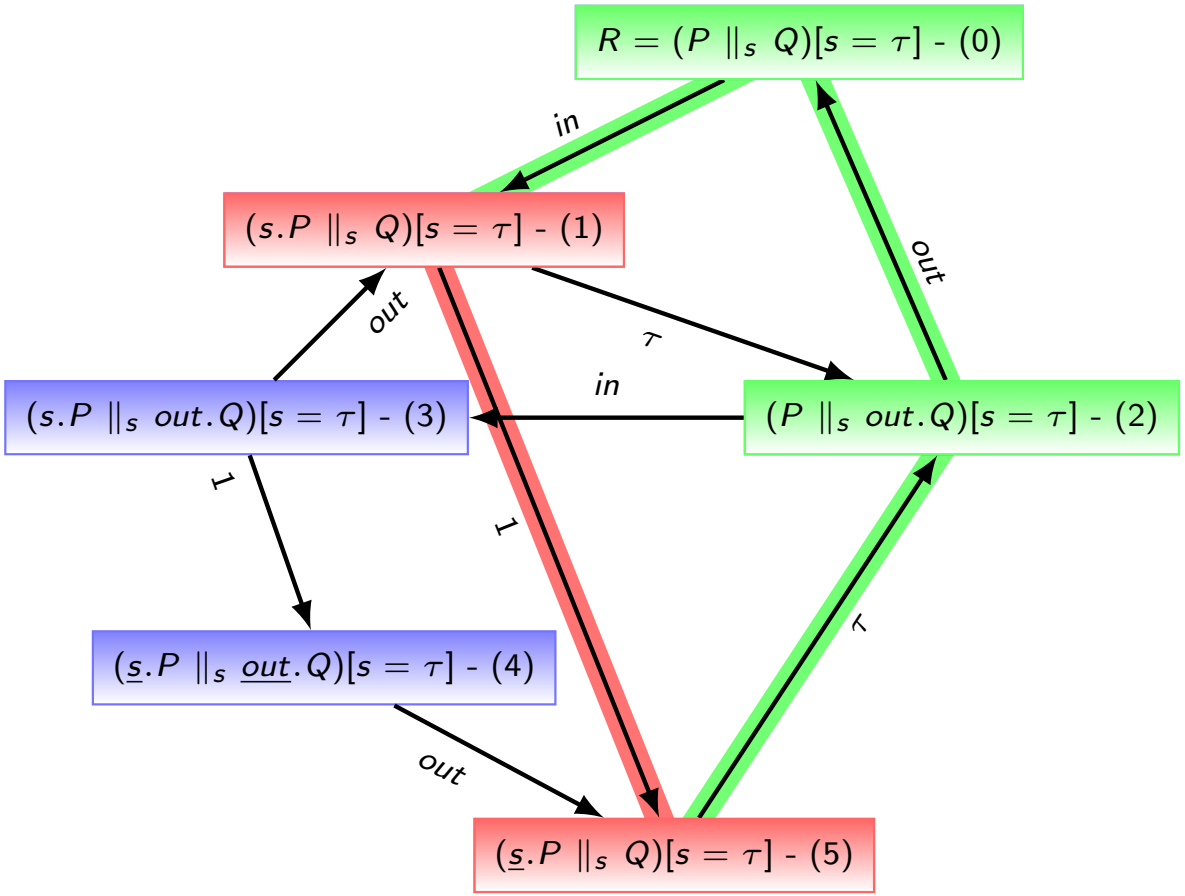


$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$4 \rightsquigarrow 3$
1 *in* - 0 *ts* ($+1$ due to $(3, 4)$)
average performance $= 1$

$5 \rightsquigarrow 1$

PIPE - New Bad cycle(s) example



$R = (P|s|Q)[s = \tau];$
$P = \underline{in}.s.P;$
$Q = \underline{s}.out.Q;$

$4 \rightsquigarrow 3$
1 *in* - 0 *ts* (+1 due to $(3, 4)$)
average performance $= 1$

$5 \rightsquigarrow 1$
1 *in* - 0 *ts* (+1 due to $(1, 5)$)
average performance $= 1$

Improving Bad cycles calculation

- Applying the procedure for each time step (u, v) in each SCC, we retain only the paths that give the maximum ratio between time and *in* actions, i.e. the *bad cycle(s)*
- The complexity of the new algorithm is $\mathbf{O(N^2E)}$:
  1. The complexity of *Bellman-Ford* is $O(NE)$
  2. We could have at most N full time steps in $rRTS(P)$
- Even if this complexity value seems not much better than the previous one, experimental results show a drastic improvement in bad cycle(s) calculation

Bad Cycle(s) algorithms performance comparison

	nodes/edges	*Floyd + Karp*	*Bellman − Ford*	*Gain*
$Fifo_3$	9/18	0	0	0%
$Pipe_3$	20/40	32	15	53.2%
$Buff_3$	17/34	281	47	83.2%
$Fifo_7$	17/38	15	0	100%
$Pipe_7$	648/1896	40375	1063	97.3%
$Buff_7$	240/560	5547	1531	72.3%
$Fifo_9$	21/48	16	0	100%
$Pipe_9$	3680/12648	-	34110	100%
$Buff_9$	448/1064	32594	8312	74.4%

Table: Bad cycles detection time (expressed in *ms*)

And...

Thanks!