Generalized Timed Büchi Automata (Revisited)

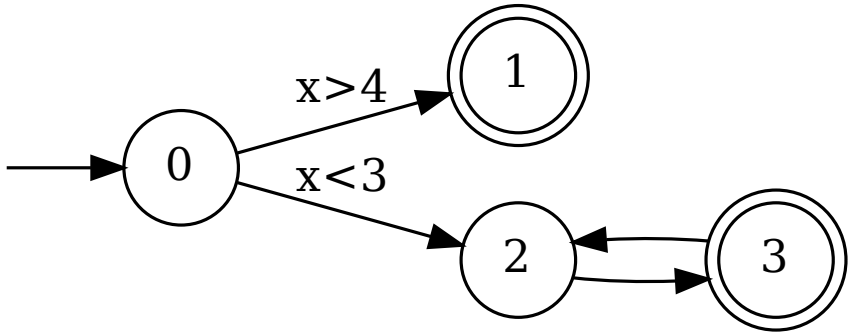
Philipp Schlehuber-Caissier
philipp@epita.lrde.fr
Post-Doc, LRDE, Epita

Timed Büchi Automata

Timed Büchi Automata are, like Timed Automata, defined as a tuple

$$\mathcal{A} = (L, l_0, L_F, X, E, \mathrm{Inv})$$

however a run $r = ((l_0, v_0), (l_1, v_1), (l_2, v_2), ...)$ has to visit L_F infinitely often in order to be accepting.



Rejected : $r_{rej} = ((0, 0), (1, 5))$

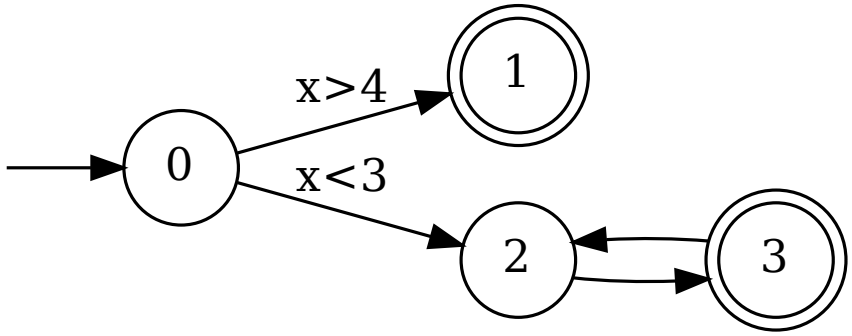
Accepted : $r_{acc} = ((0, 0), ((2, 2), (3, 2))^{\omega})$

Timed Büchi Automata

Timed Büchi Automata are, like Timed Automata, defined as a tuple

$$\mathcal{A} = (L, l_0, L_F, X, E, \mathrm{Inv})$$

however a run $r = ((l_0, v_0), (l_1, v_1), (l_2, v_2), ...)$ has to visit L_F infinitely often in order to be accepting.



Bonus question : Guarantee that time is allowed to diverge

Accepted : $r_{acc} = ((0,0), (2,2), (3,2), (2,3), (3,3), (2,4), (3,4), ...)$

Emptiness check

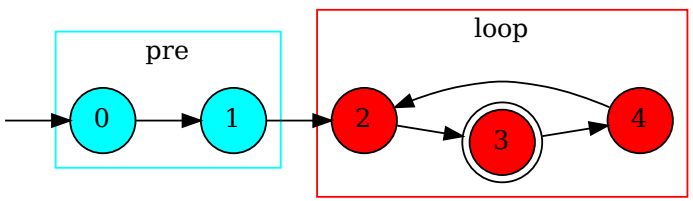
- For reachability/safety properties, we use forward exploration.

Emptiness check

- For reachability/safety properties, we use forward exploration. We therefore finished when a location $l \in L_F$ is attained.

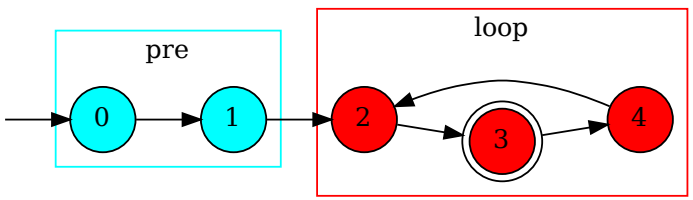
Emptiness check

- For reachability/safety properties, we use forward exploration. We therefore finished when a location $l \in L_F$ is attained.
- For Büchi acceptance this is not sufficient as we need a "Lasso" :
  $r = (r_{pre}, (r_{cyc})^{\omega})$



Emptiness check

- For reachability/safety properties, we use forward exploration. We therefore finished when a location $l \in L_F$ is attained.
- For Büchi acceptance this is not sufficient as we need a "Lasso" :
  $r = (r_{pre}, (r_{cyc})^\omega)$



This can be done using a Nested Dept First Search (NDFS).

NDFS

- The basic idea of NDFS is to have two distinct depth first traversals of the graph.

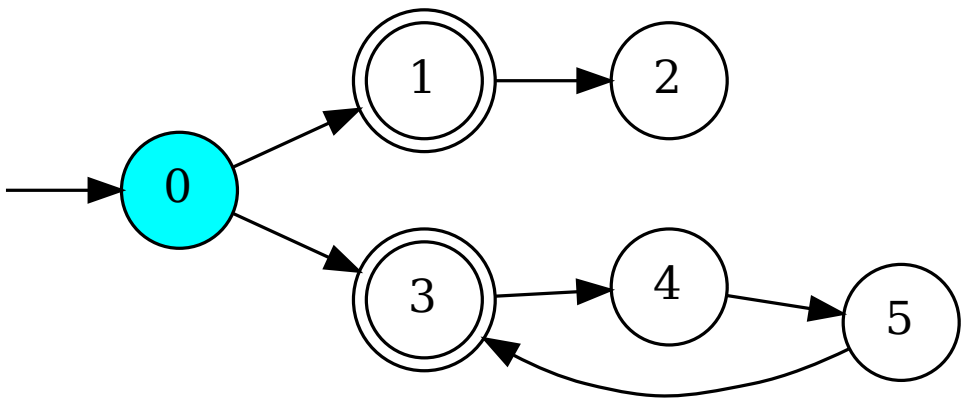
NDFS

- The basic idea of NDFS is to have two distinct depth first traversals of the graph.
- One to find a prefix ("$dfsBlue$") and one to find a cycle ("$dfsBlue$").

NDFS

- The basic idea of NDFS is to have two distinct depth first traversals of the graph.
- One to find a prefix ("$dfsBlue$") and one to find a cycle ("$dfsBlue$").
- Needs to maintain three different sets of states :
  Live, Explored, Dead

NDFS

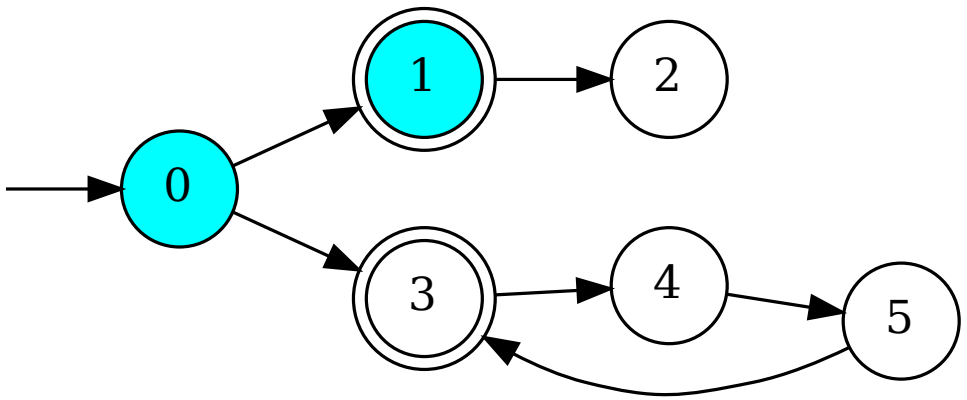


Live=$Cyan$, Explored=$Blue$, Dead=Red

$$Cyan = Blue = Red = \varnothing$$

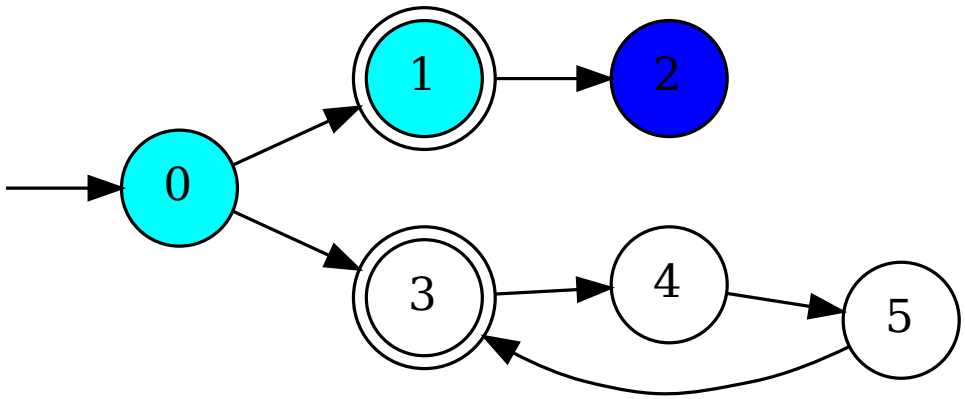
$$dfsBlue(0)$$

NDFS



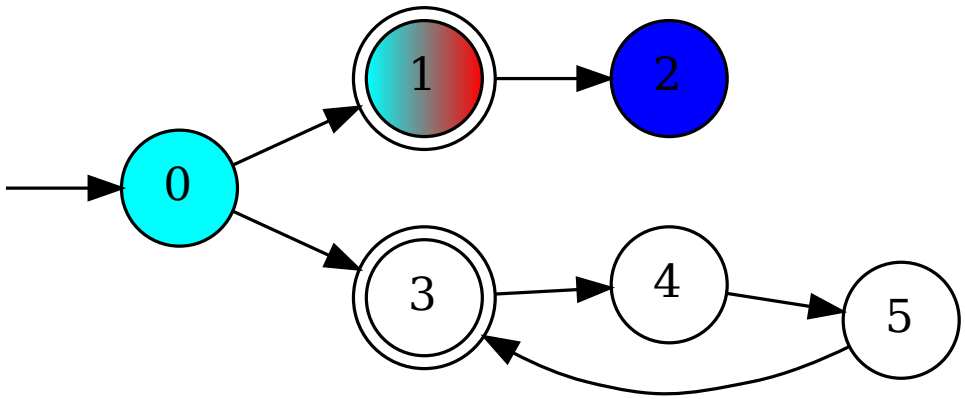
$$succ \notin Blue \wedge succ \notin Cyan \rightarrow dfsBlue(succ)$$

NDFS



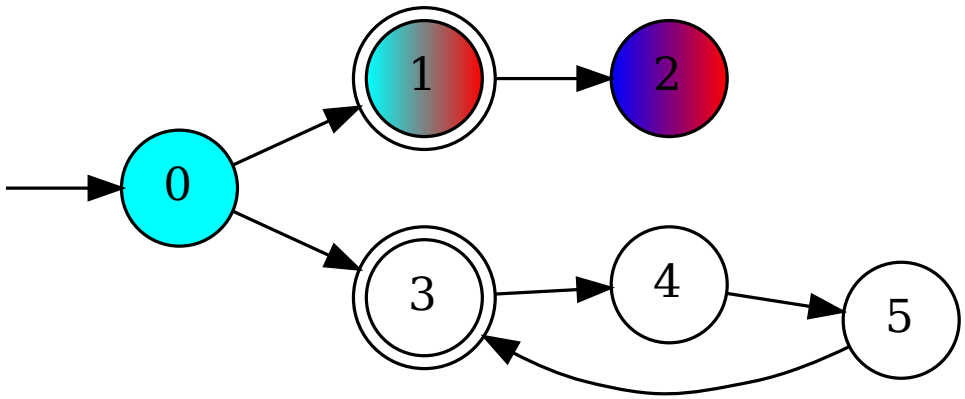
$$s \notin L_F \rightarrow Blue = Blue \cup s; Cyan = Cyan \setminus s$$

NDFS



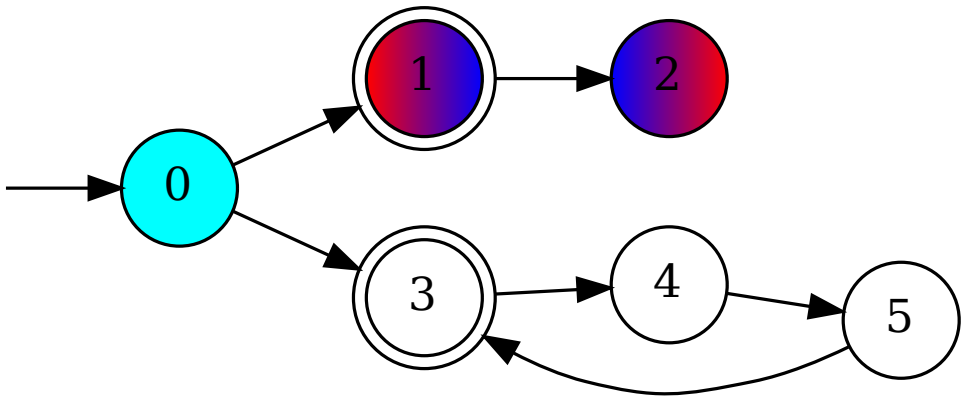
$$s \in L_F \rightarrow dfsRed(s)$$

NDFS



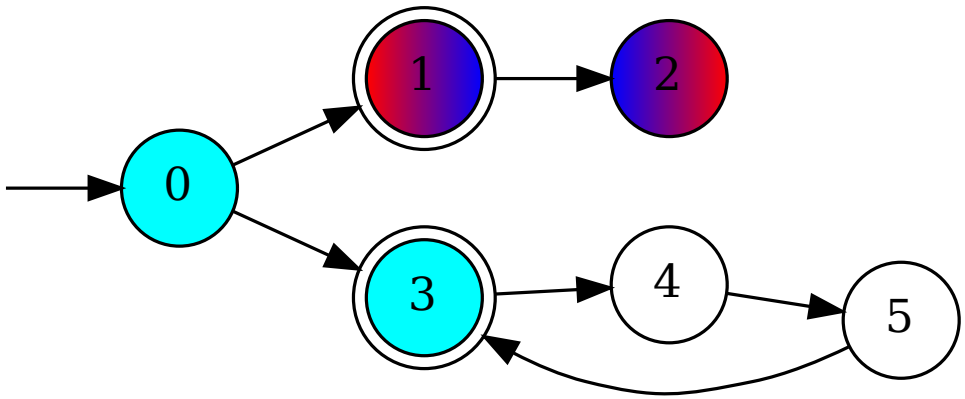
$$succ \notin Red \rightarrow dfsRed(succ)$$

NDFS



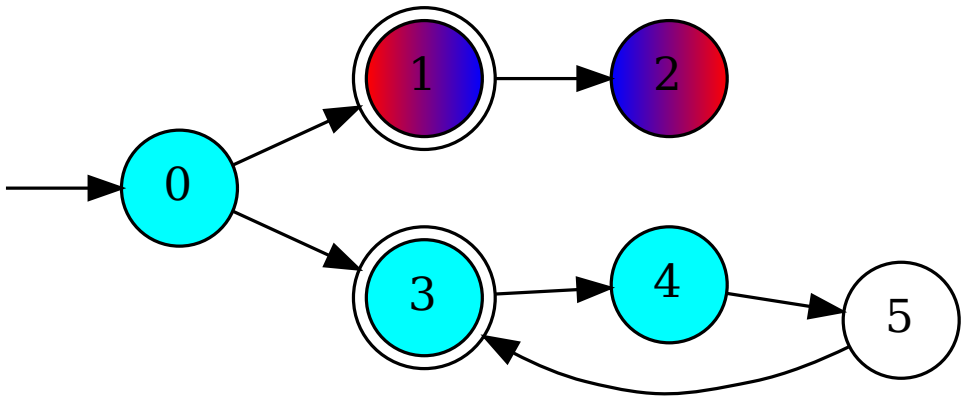
s is explored $\rightarrow Blue = Blue \cup s; Cyan = Cyan \setminus s$

NDFS



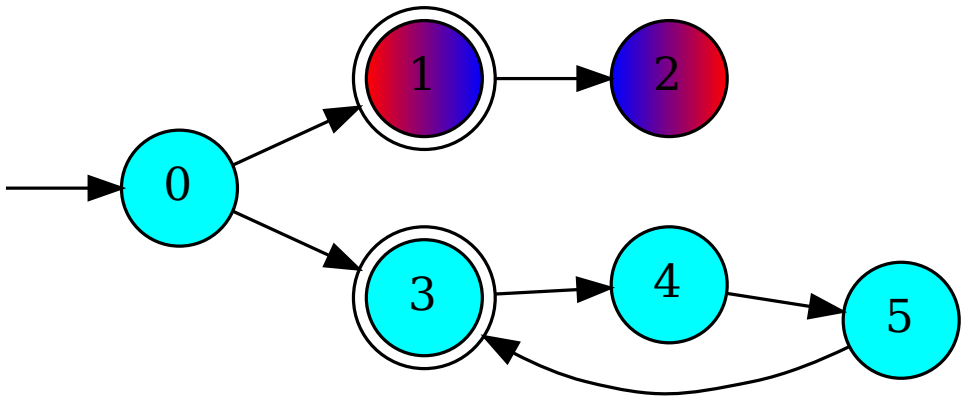
$$succ \notin Blue \wedge succ \notin Cyan \rightarrow dfsBlue(succ)$$

NDFS



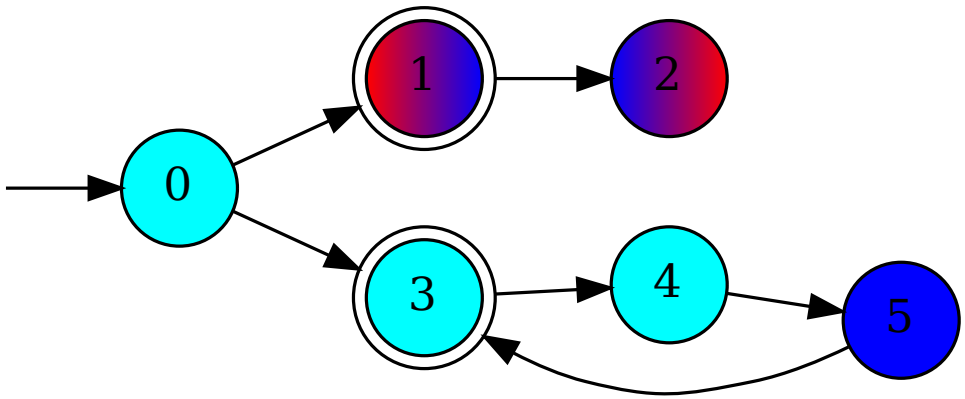
$$succ \notin Blue \wedge succ \notin Cyan \rightarrow dfsBlue(succ)$$

NDFS



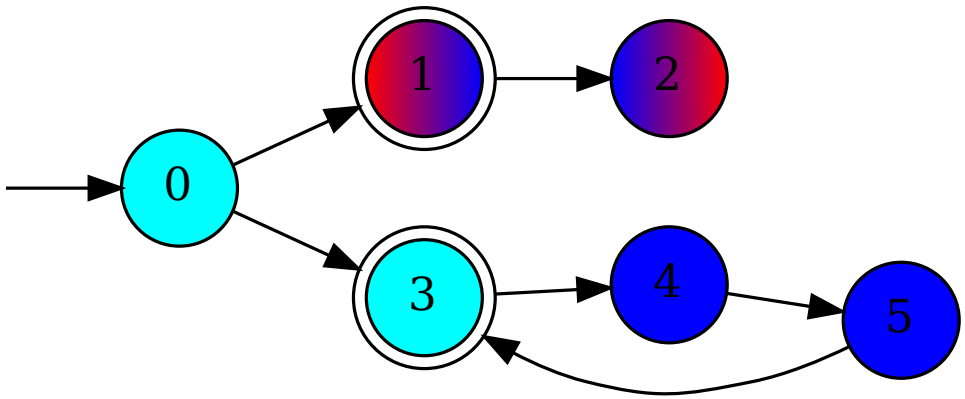
$$succ \notin Blue \wedge succ \notin Cyan \rightarrow dfsBlue(succ)$$

NDFS



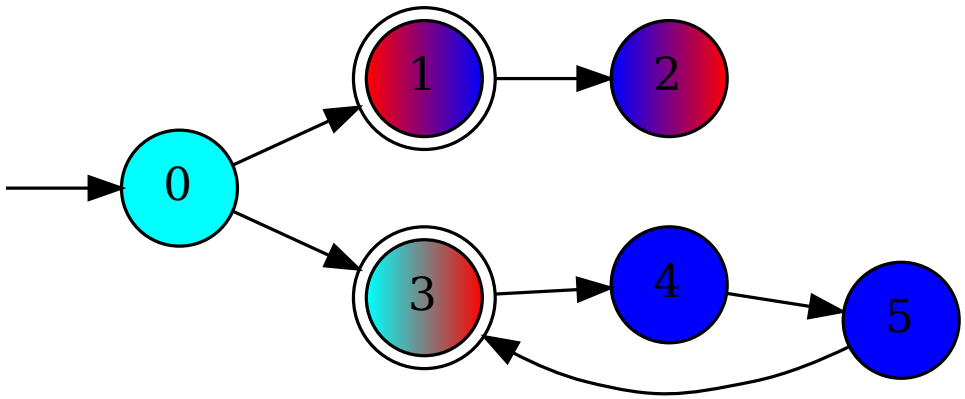
s is explored $\rightarrow Blue = Blue \cup s; Cyan = Cyan \setminus s$

NDFS



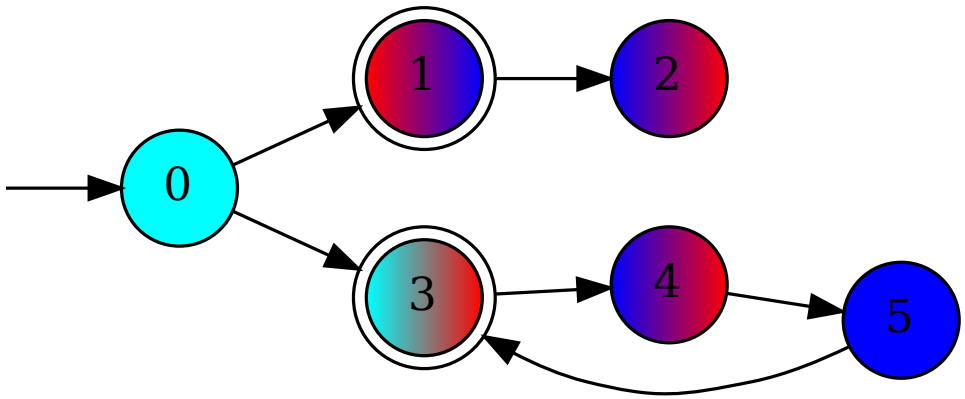
s is explored $\rightarrow Blue = Blue \cup s; Cyan = Cyan \setminus s$

NDFS



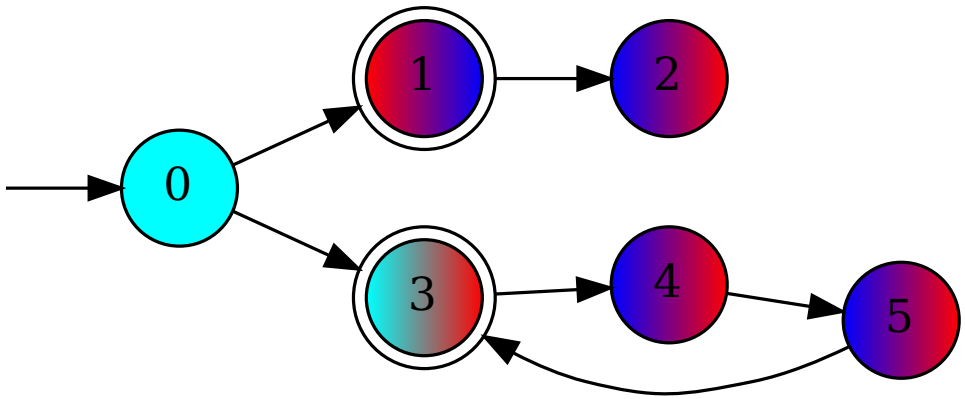
$$s \in L_F \rightarrow dfsRed(succ)$$

NDFS



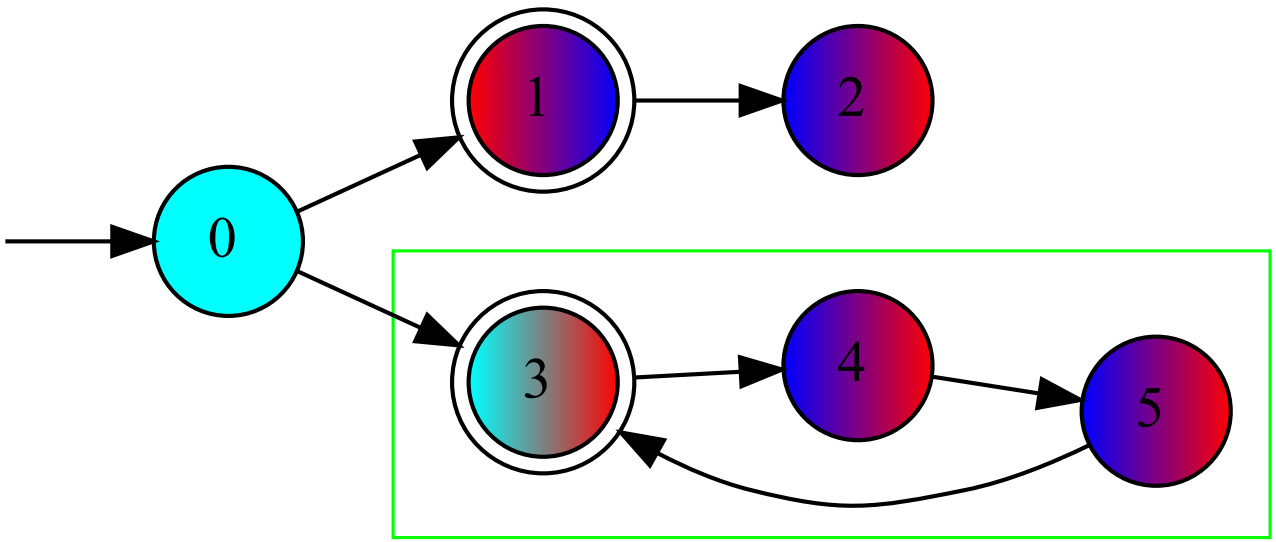
$$succ \notin Red \rightarrow dfsRed(succ)$$

NDFS



$$succ \notin Red \rightarrow dfsRed(succ)$$

NDFS



$succ \in Cyan \rightarrow$ report cycle

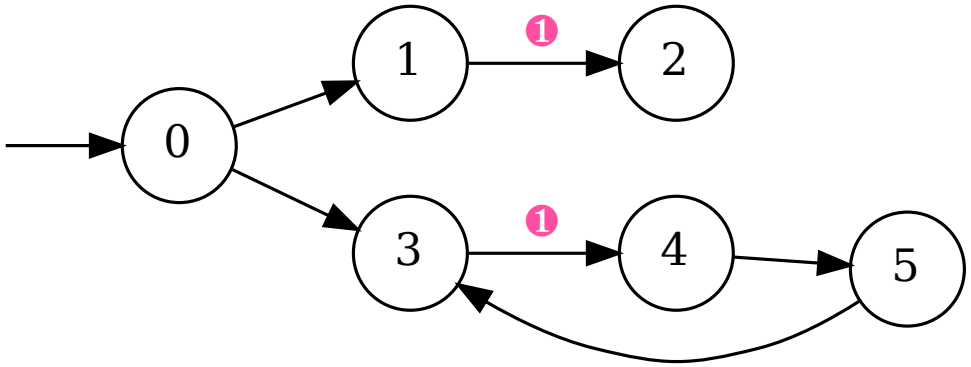
Couvreur

Another popular approach for checking Büchi emptiness is Couvreur's algorithm [1].

It's main idea is to find an SCC that satisfies the acceptance condition.

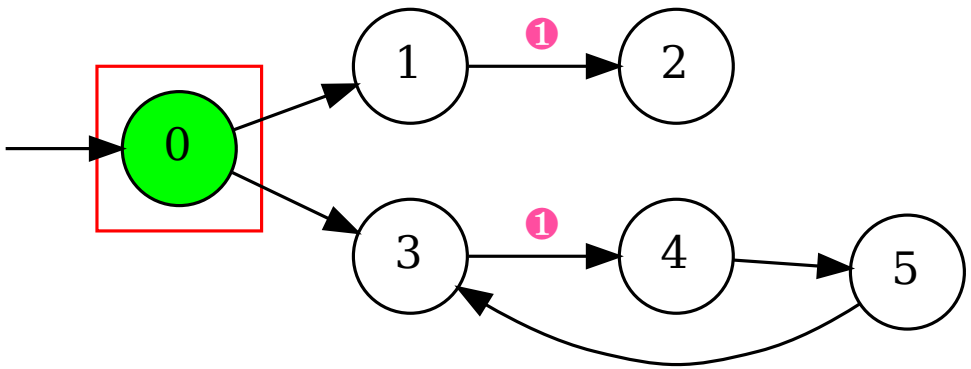
1. Couvreur, J. M., Duret-Lutz, A., & Poitrenaud, D. (2005, August). On-the-fly emptiness checks for generalized Büchi automata. In International SPIN Workshop on Model Checking of Software (pp. 169-184). Springer, Berlin, Heidelberg.

Couvreur



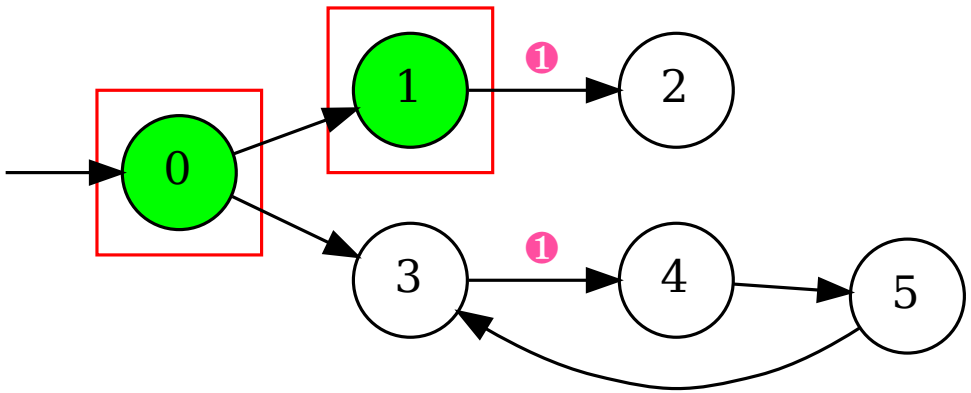
To be coherent with SPOT,
we will use a transition based acceptance here.

Couvreur



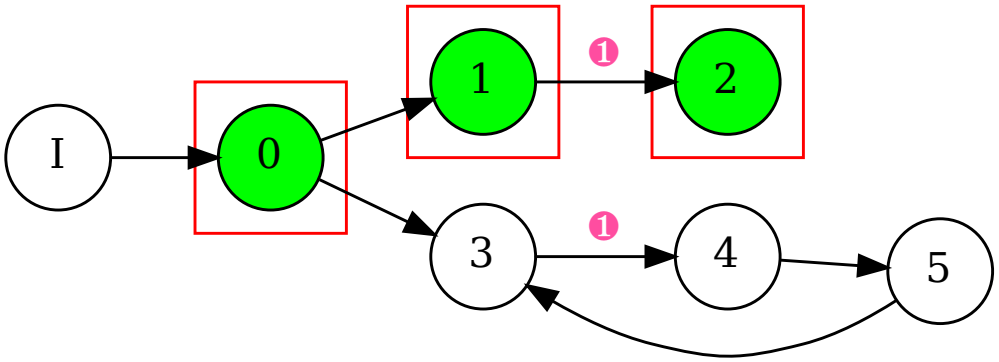
When discovering a new state, it gets its own scc and is put on the live stack $Green$

Couvreur



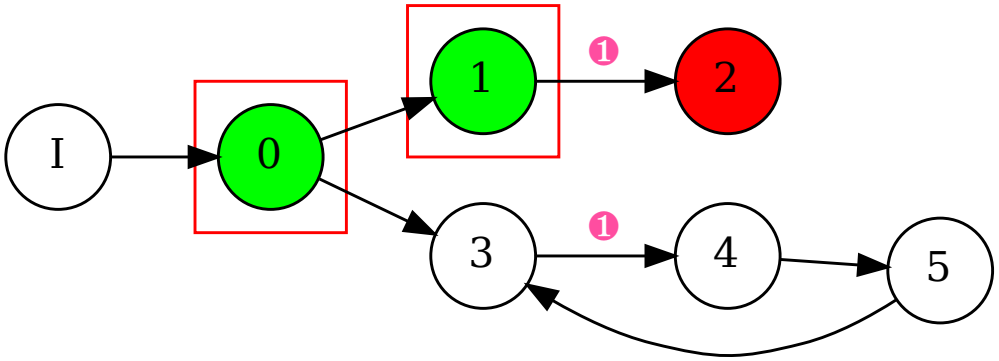
When discovering a new state, it gets its own scc and is put on the live stack $Green$

Couvreur



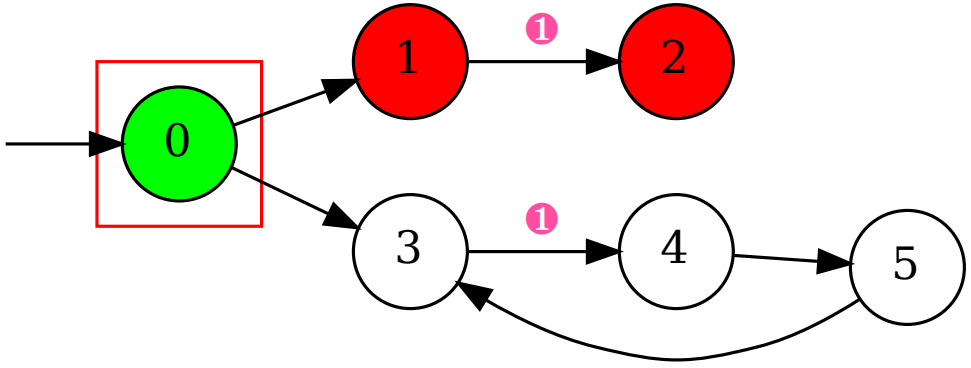
When discovering a new state, it gets its own scc and is put on the live stack $Green$

Couvreur



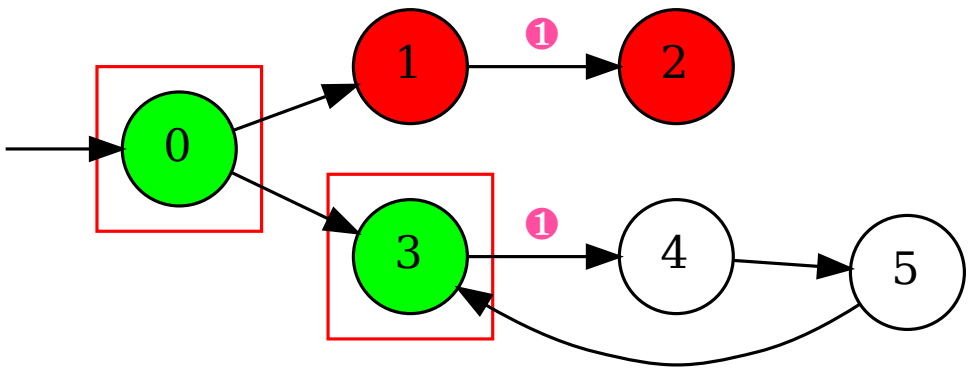
When backtracking the root of an scc,
all of the states are "declared dead" and added to Red.

Couvreur



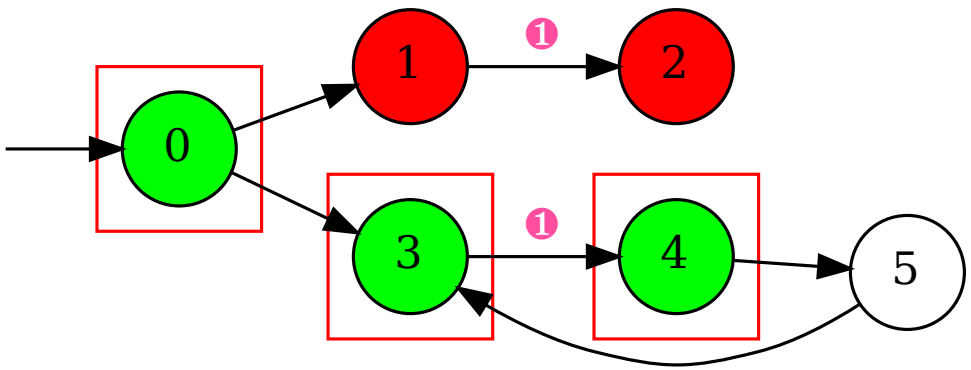
When backtracking the root of an scc,
all of the states are "declared dead" and added to Red.

Couvreur



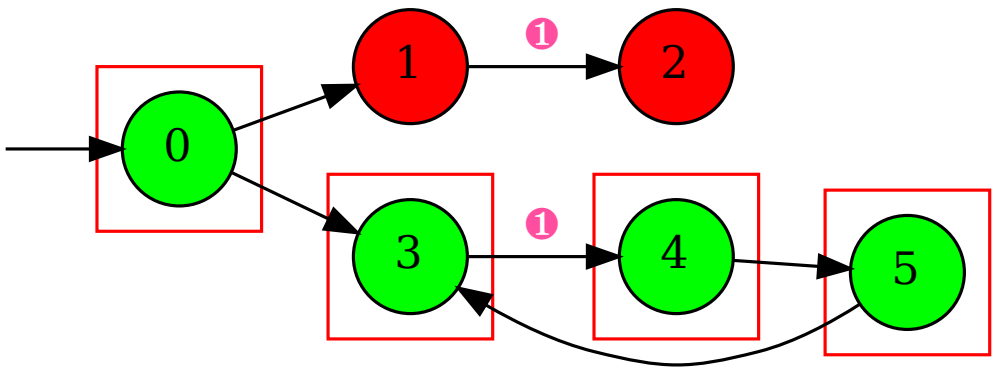
We keep on exploring.

Couvreur



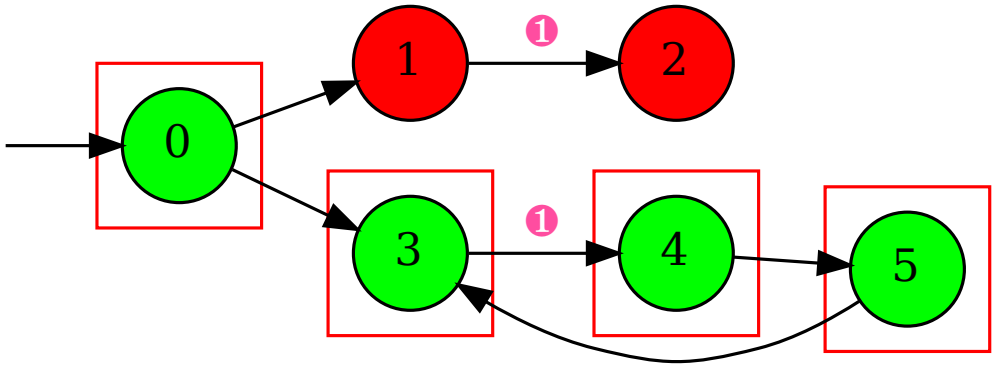
We keep on exploring.

Couvreur



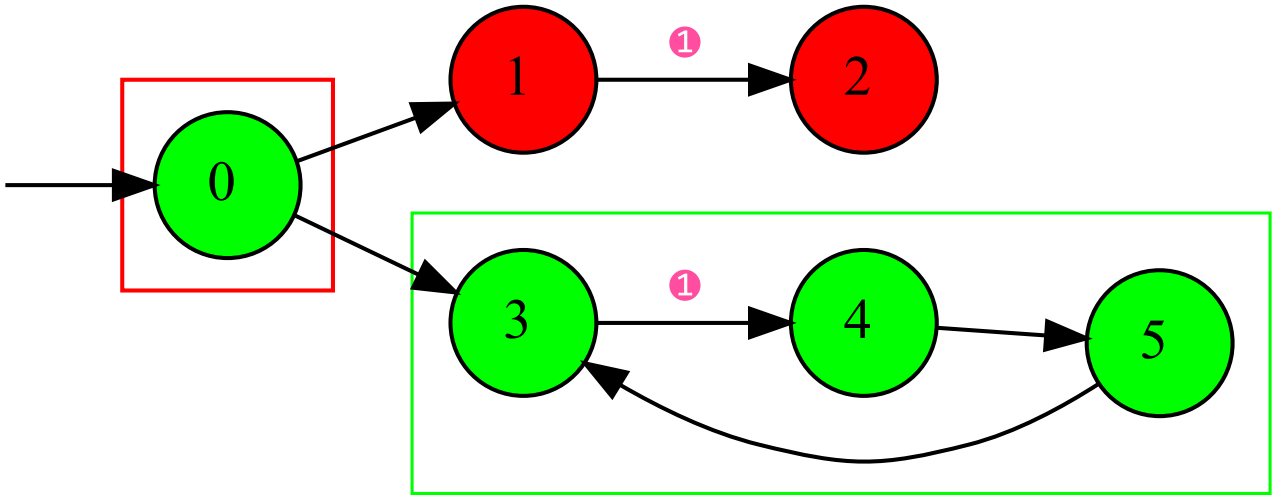
We keep on exploring.

Couvreur



Until we find a back-edge. That is an edge which take us back to a live state (one that is in $Green$)

Couvreur



All scc's between them get merged.
The acceptanace marks are accumulated.

Using subsumption

- Both algorithms handle nodes like discrete states
- Using subsumption where appropriate
  is likely to speed-up calculations

Using subsumption

NDFS

Couvreur

Using subsumption

NDFS

Dead subsumption :
$s \in Red$
becomes
$\exists s' \in Red\colon\ s \preceq s'$

Couvreur

Dead subsumption :
$s \in Red$
becomes
$\exists s' \in Red\colon\ s \preceq s'$

Using subsumption

NDFS

Dead subsumption :
$s \in Red$
becomes
$\exists s' \in Red\colon\ s \preceq s'$

$Red \rightarrow Cyan$ subsumption :
$\exists s' \in Cyan\colon s' \preceq s$ report cycle

Couvreur

Dead subsumption :
$s \in Red$
becomes
$\exists s' \in Red\colon\ s \preceq s'$

Livestack subsumption :
All live states for which
$s' \in Green,\ s' \preceq succ$ holds are valid successors as well

Using subsumption

NDFS

Dead subsumption :
$s \in Red$
becomes
$\exists s' \in Red\colon\ s \preceq s'$

$Red \to Cyan$ subsumption :
$\exists s' \in Cyan\colon s' \preceq s$ report cycle

Equality for $Blue$ and $Cyan$:
Subsumption cannot be used on
$succ \in Blue \cup Cyan$.

Couvreur

Dead subsumption :
$s \in Red$
becomes
$\exists s' \in Red\colon\ s \preceq s'$

Livestack subsumption :
All live states for which
$s' \in Green,\ s' \preceq succ$ holds are valid successors as well

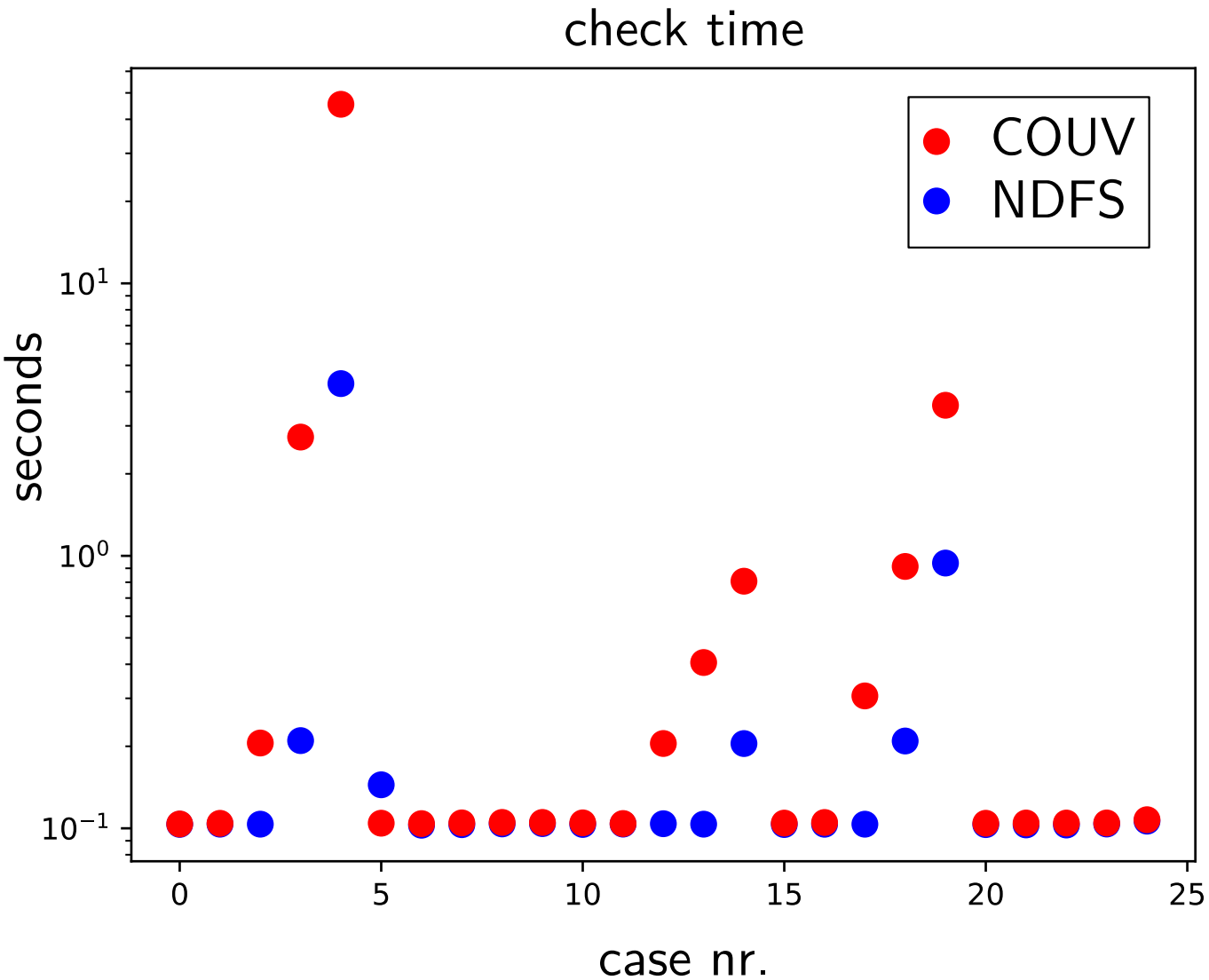
Compairison

Positive and negative aspects

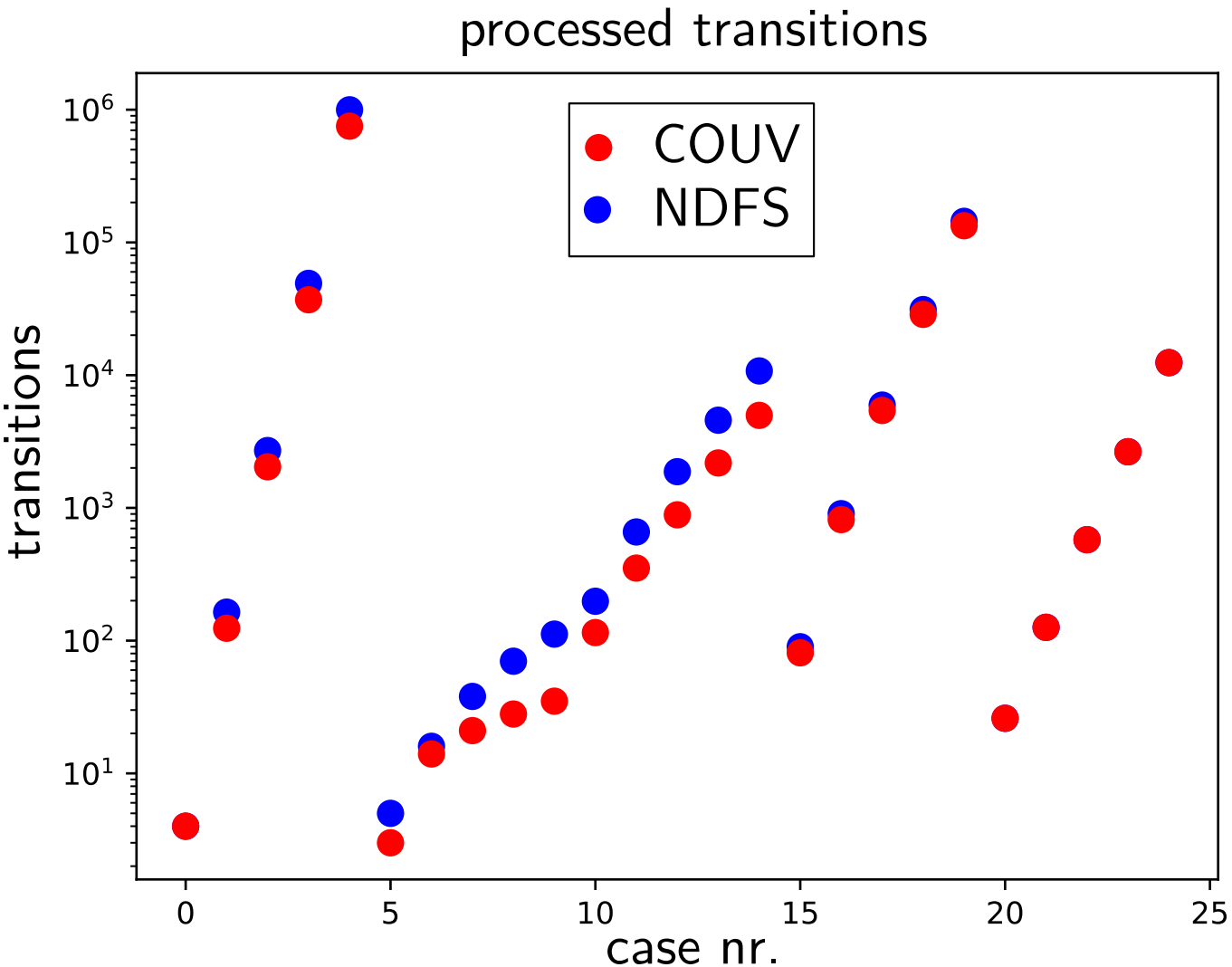
+ Faster : only requires one traversal
+ Faster : usually less compairisons
+ Easier : Only two sets of states
+ Requires less memory
+ Directly applicable to generalized Büchi acceptance
− Provides no actual counterexample

1. Laarman, A., Olesen, M. C., Dalsgaard, A. E., Larsen, K. G., & Van De Pol, J. (2013, July). Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In International Conference on Computer Aided Verification (pp. 968-983). Springer, Berlin, Heidelberg.

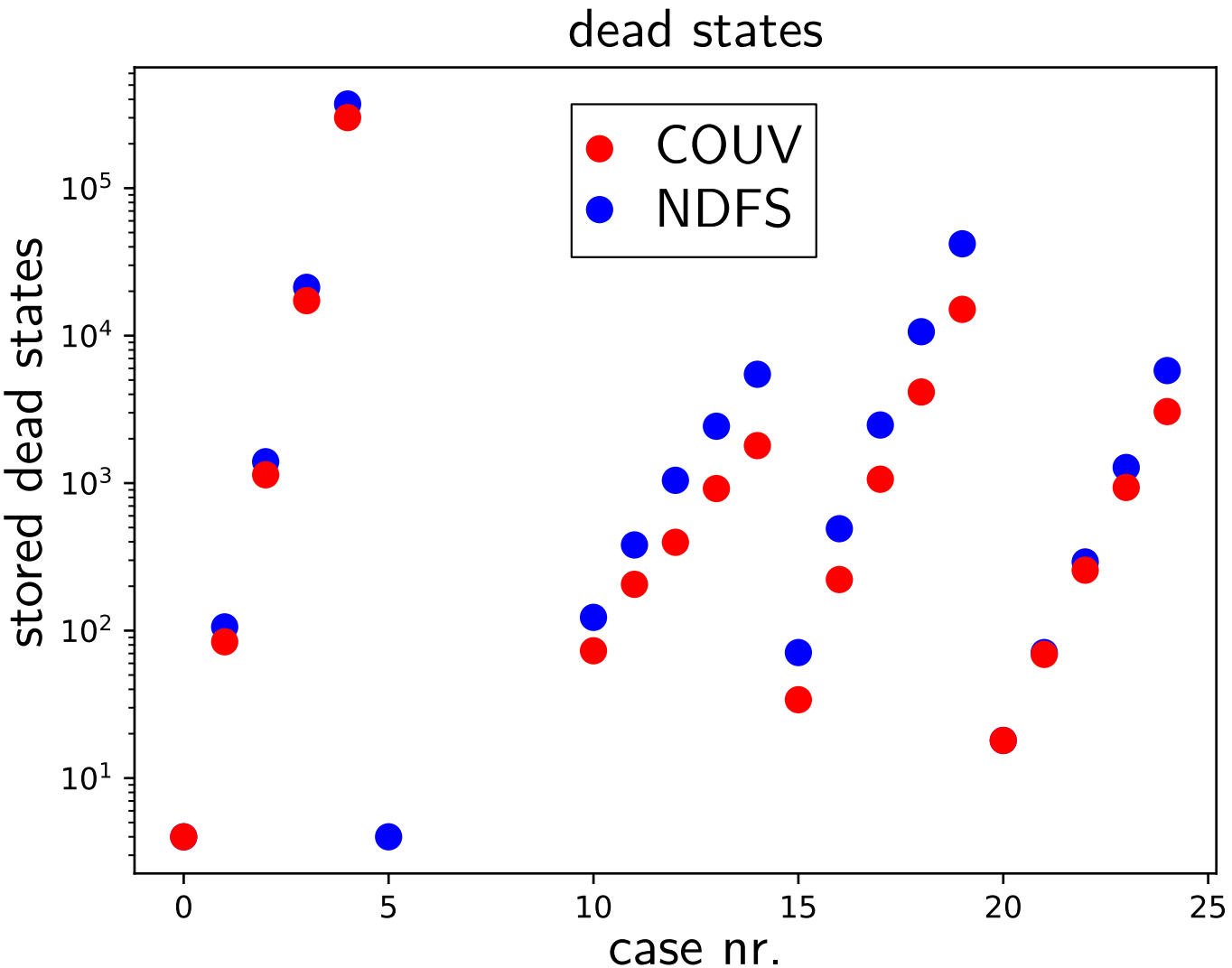
Benchmarks 1



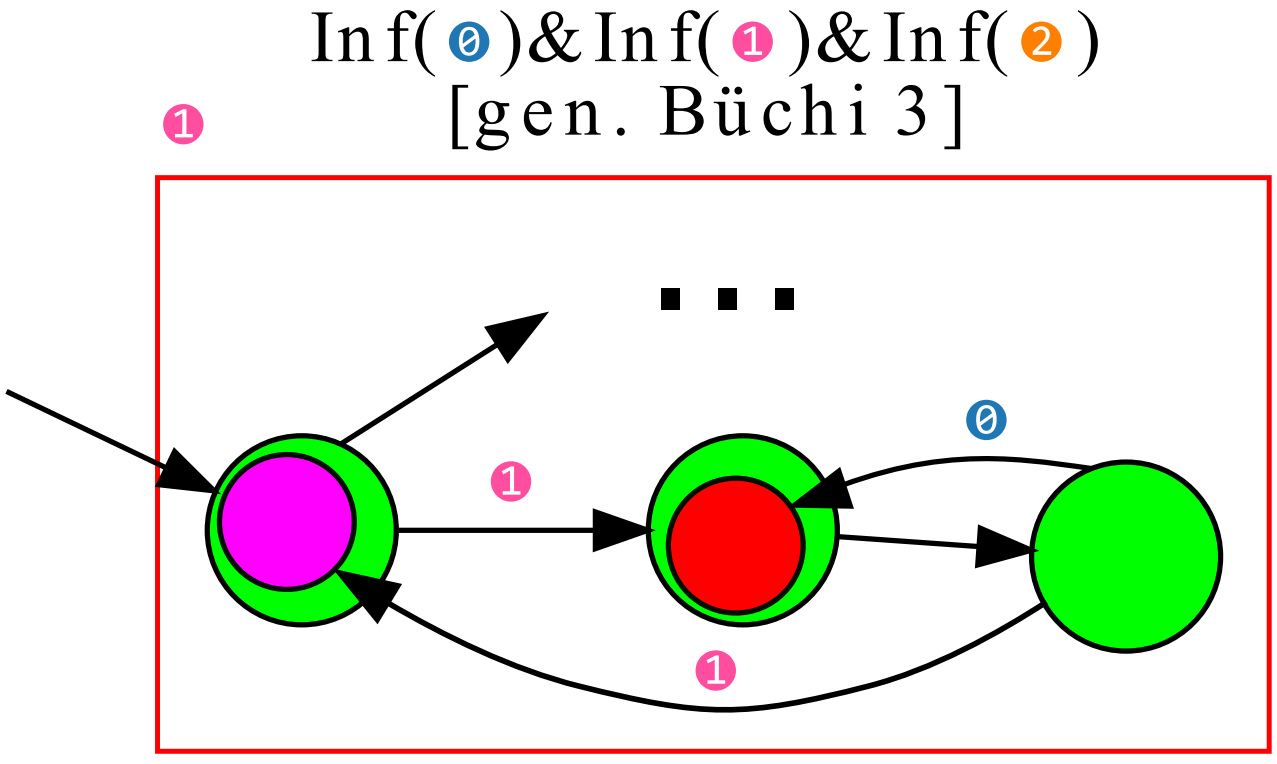
Benchmarks 1



Benchmarks 1

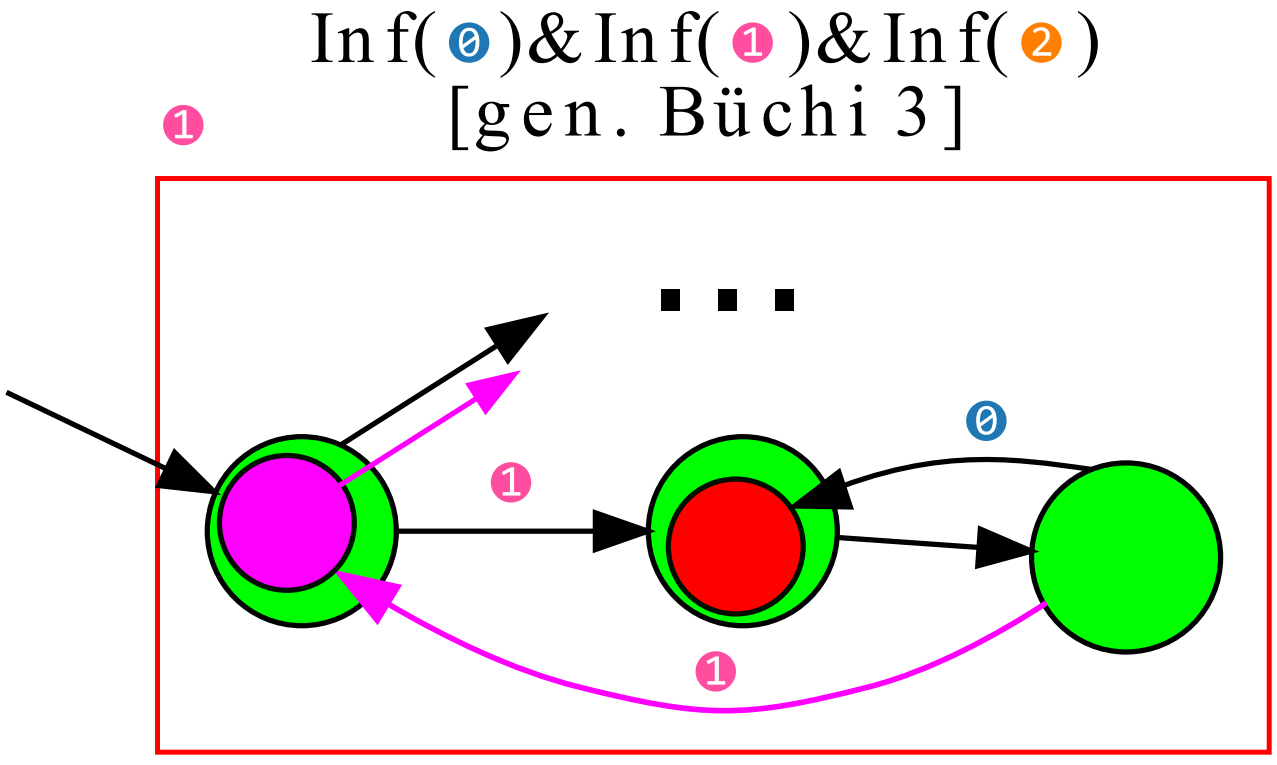


Further optimizations 1



Pruning based on last SCC

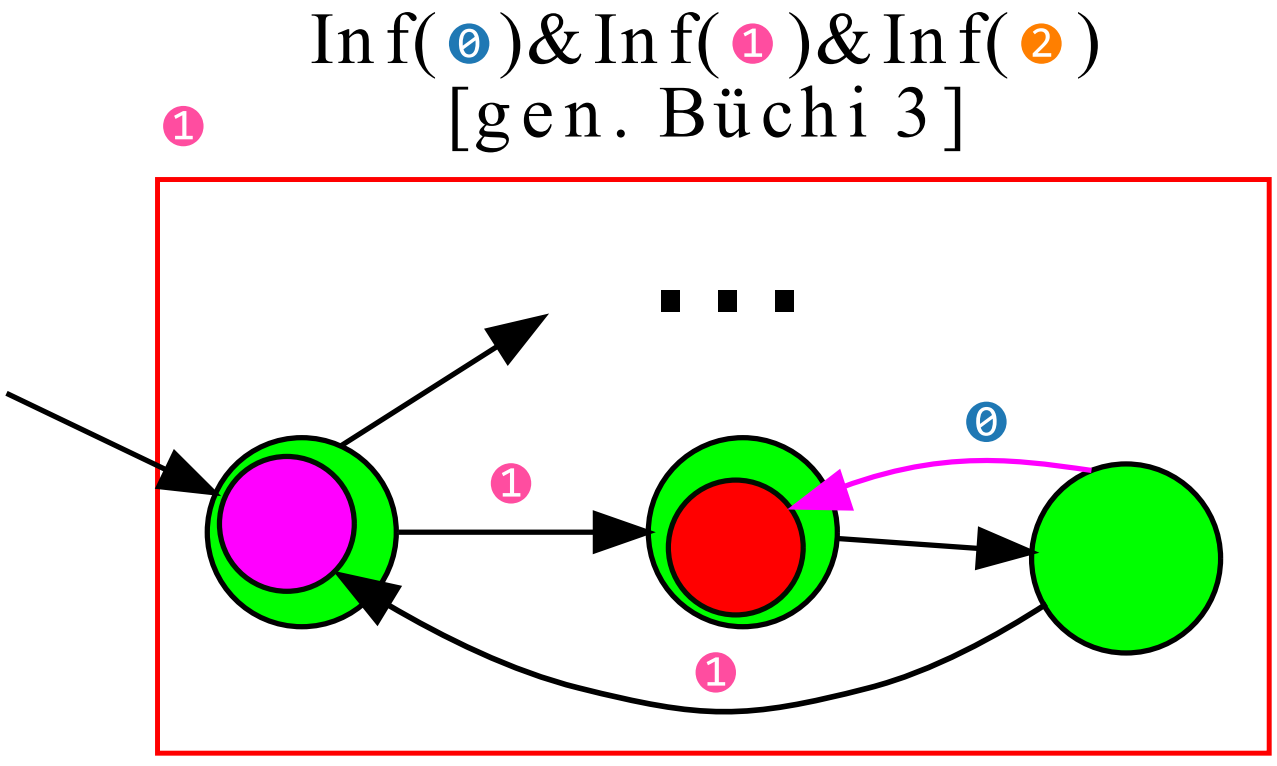
Further optimizations 1



Pruning based on last SCC

We can skip the successor if covered by some node in the scc and the transition has no new color.

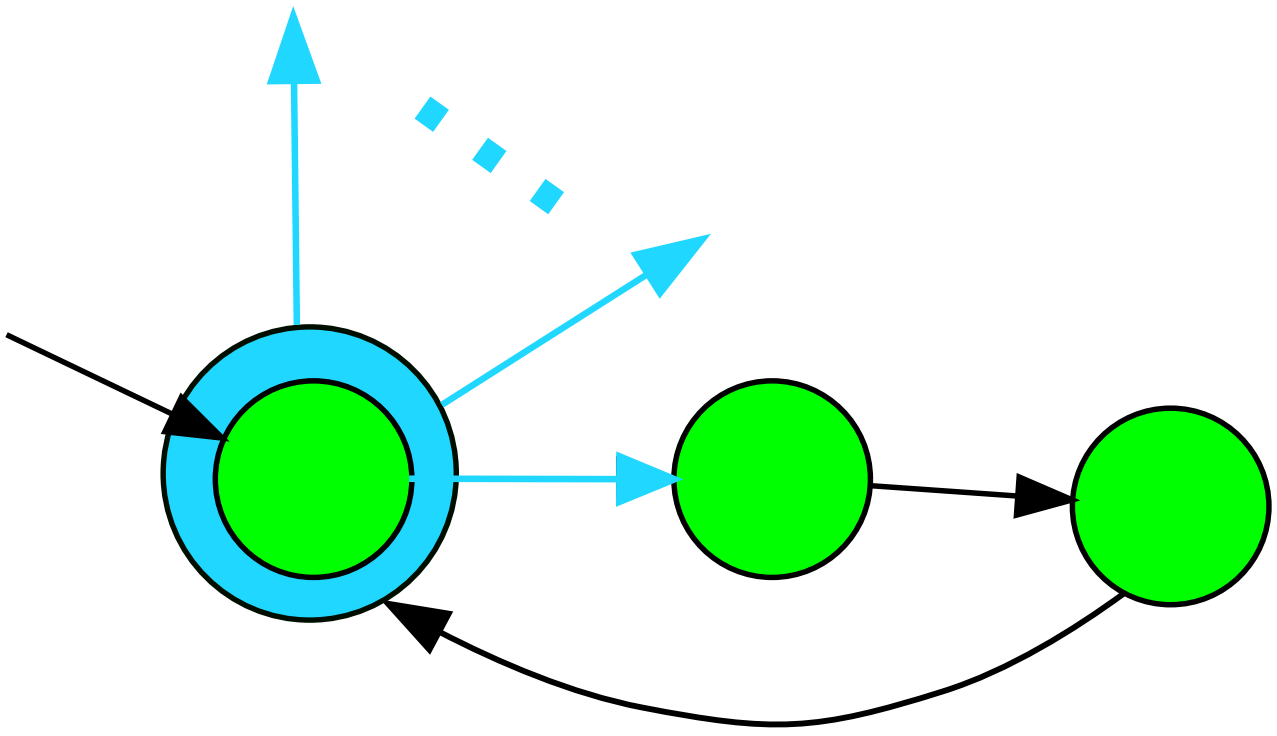
Further optimizations 1



Pruning based on last SCC

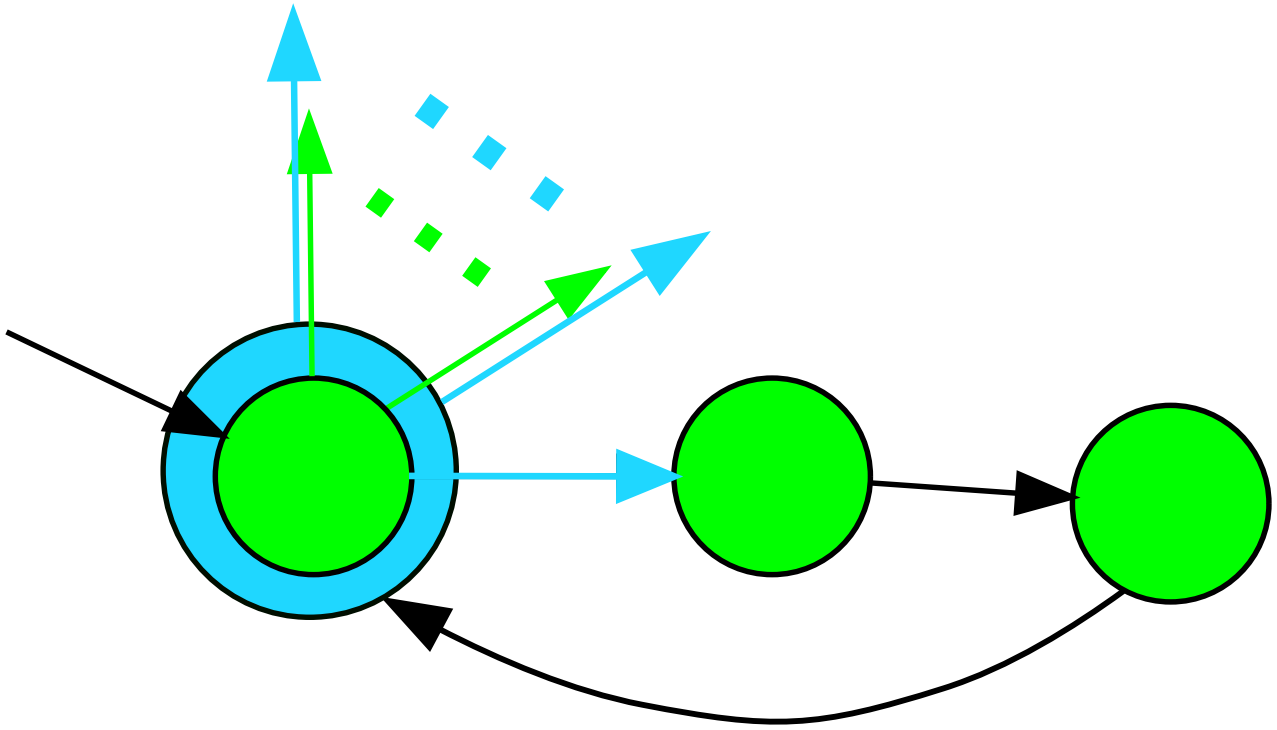
We can skip the successor if covered by some node in the scc and the transition has no new color.

Further optimizations 2



Superseded nodes

Further optimizations 2



Superseded nodes

We can skip the successors of a node if a larger node was already explored.

Further optimizations 3

The set of dead states Red (NDFS/Couvreur) are typically implemented as hash maps.
This means we have to check the states against a sub-set sharing the same hash value.

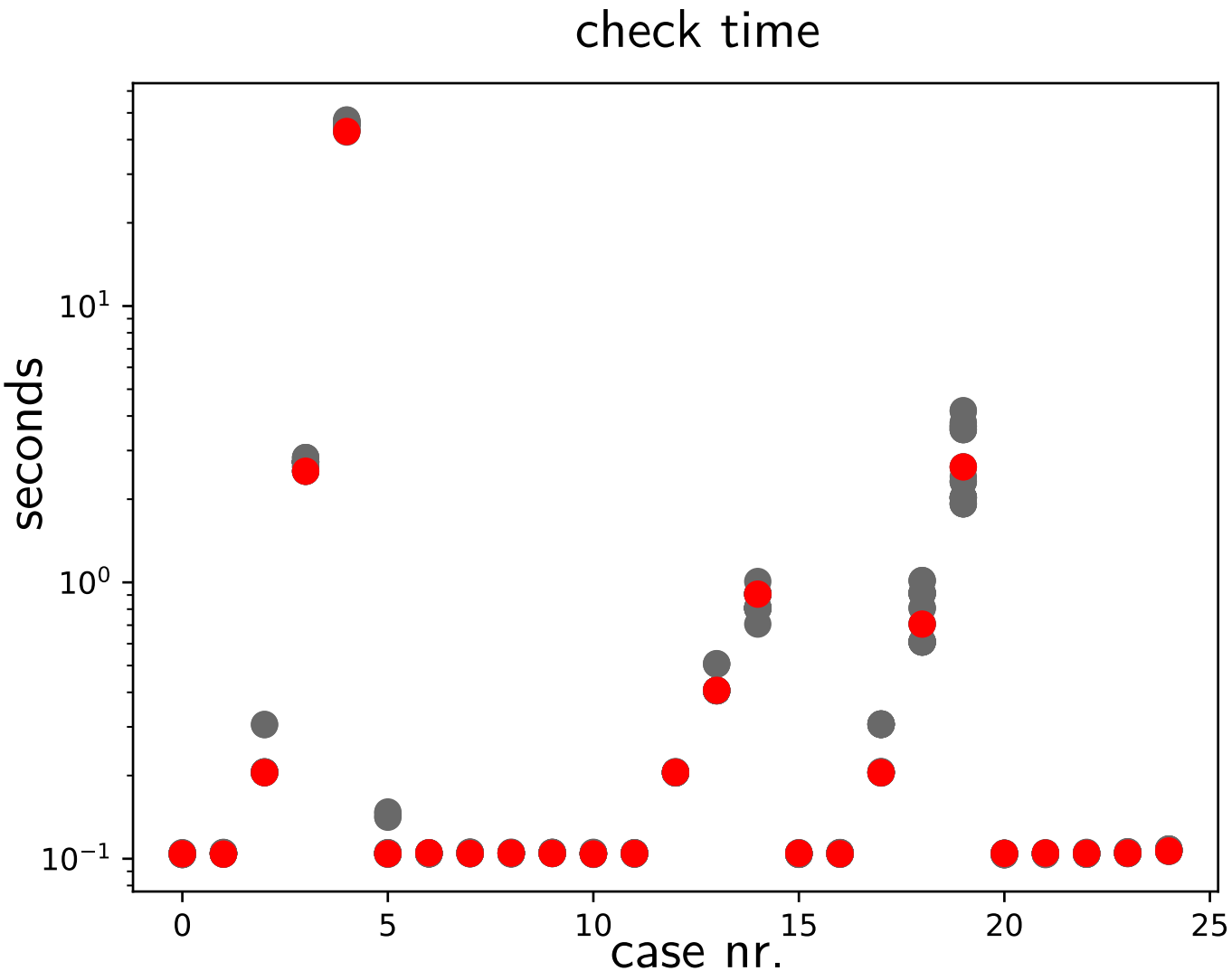
In order to decrease the (average) number of comparisons, we sort the set of candidate states.

$$\left[s_0 \quad s_1 \quad s_2 \quad s_3 \quad s_4 \right]$$

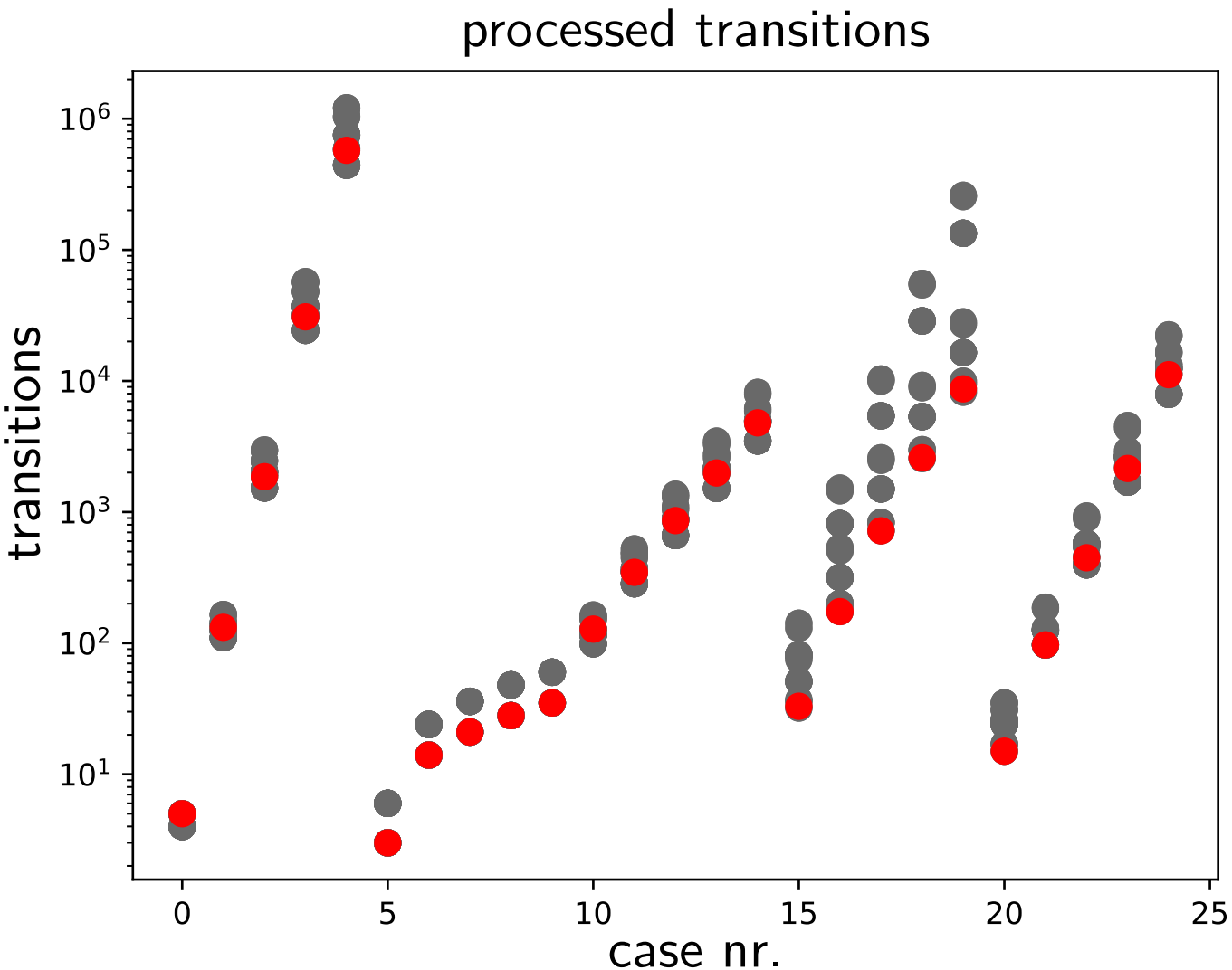
$$\downarrow \quad succ \preceq s_2$$

$$\left[s_2 \quad s_0 \quad s_1 \quad s_3 \quad s_4 \right]$$

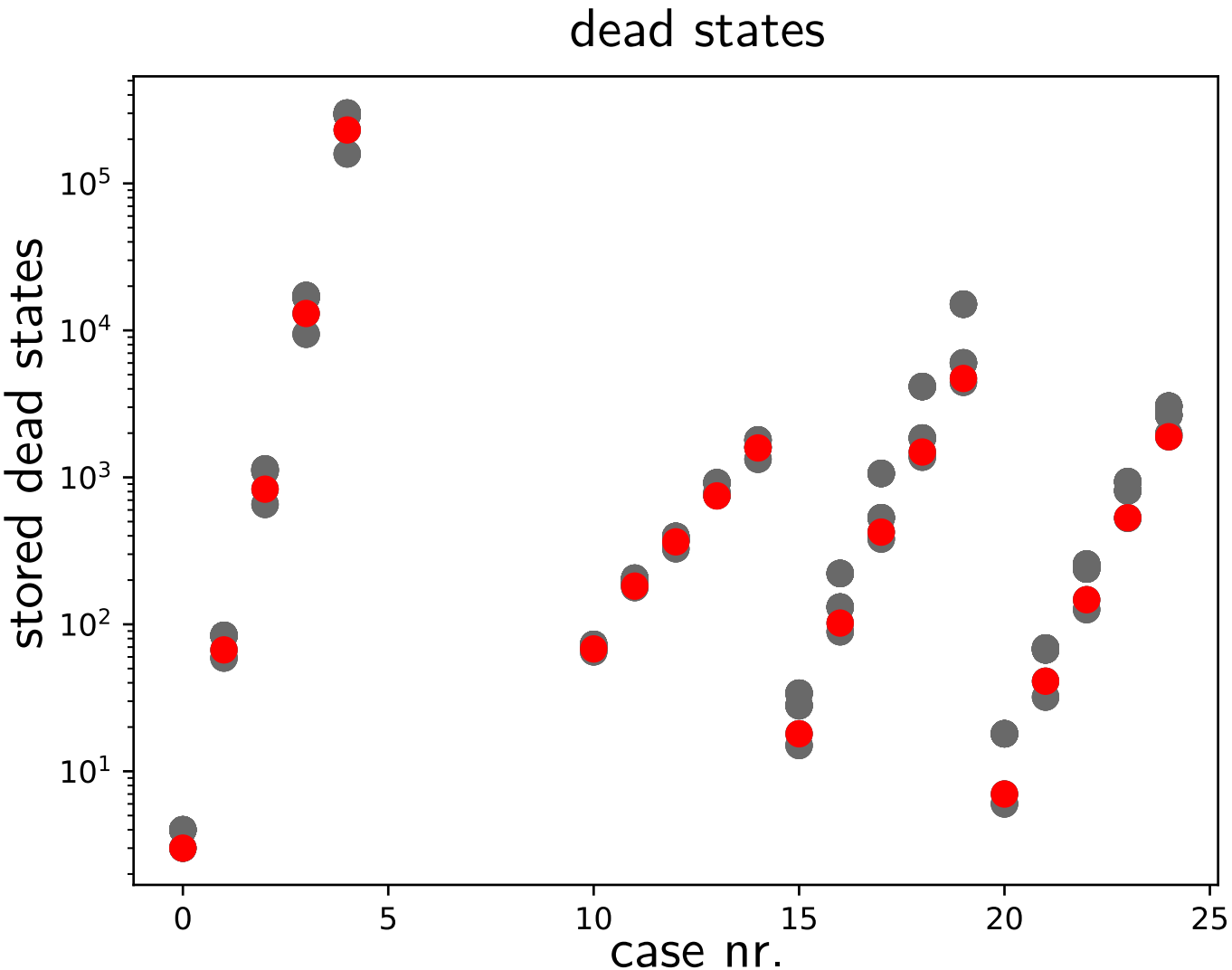
Benchmarks 1 - Optimization impact



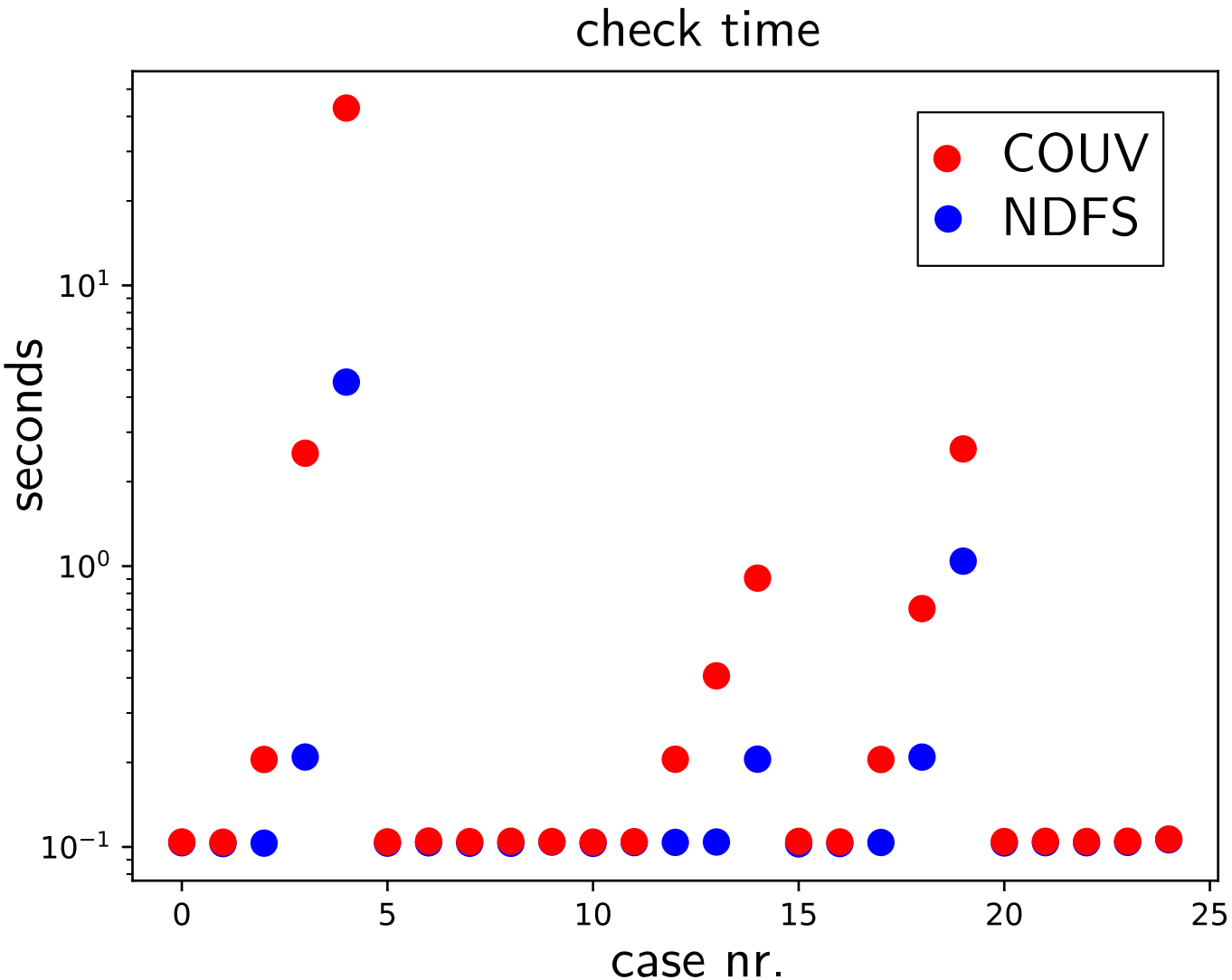
Benchmarks 1 - Optimization impact



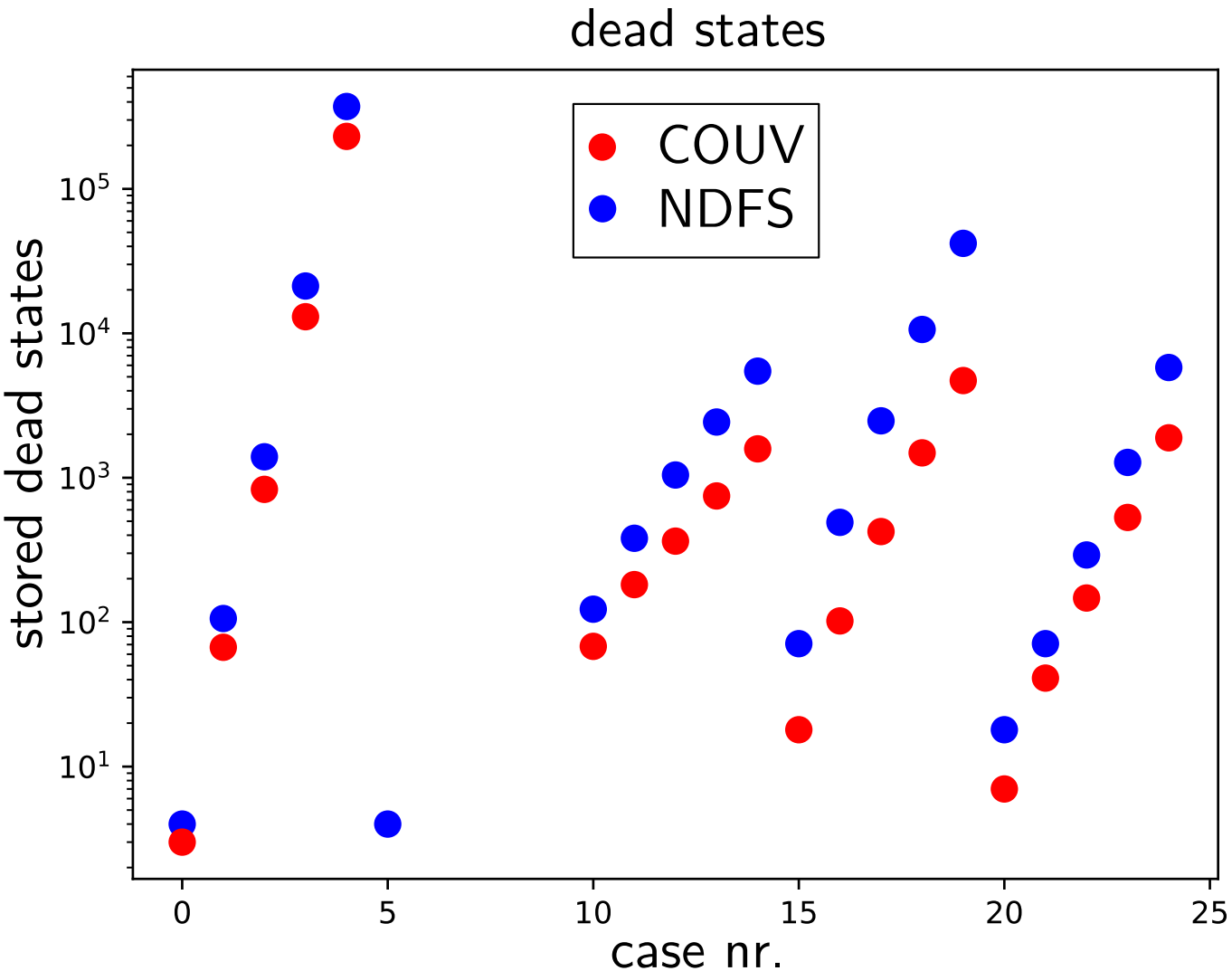
Benchmarks 1 - Optimization impact



Benchmarks 1 - NDFS vs COUVREUR



Benchmarks 1 - NDFS vs COUVREUR



The problem of zeno runs

A run is considered zeno if it makes infinitely many switches in a finite amount of time. That is time gets "frozen".

Such runs are unrealistic and therefore discarded.

- Strong non-zeno construction $\rightarrow$ Adding a clock [2]

2. Tripakis, S., Yovine, S., & Bouajjani, A. (2005). Checking timed Büchi automata emptiness efficiently. Formal Methods in System Design, 26(3), 267-292.

The problem of zeno runs

A run is considered zeno if it makes infinitely many switches in a finite amount of time. That is time gets "frozen".

Such runs are unrealistic and therefore discarded.

- Strong non-zeno construction $\rightarrow$ Adding a clock [2]
- Guessing zone graph $\rightarrow$ Add information to the nodes [3]

2. Tripakis, S., Yovine, S., & Bouajjani, A. (2005). Checking timed Büchi automata emptiness efficiently. Formal Methods in System Design, 26(3), 267-292.

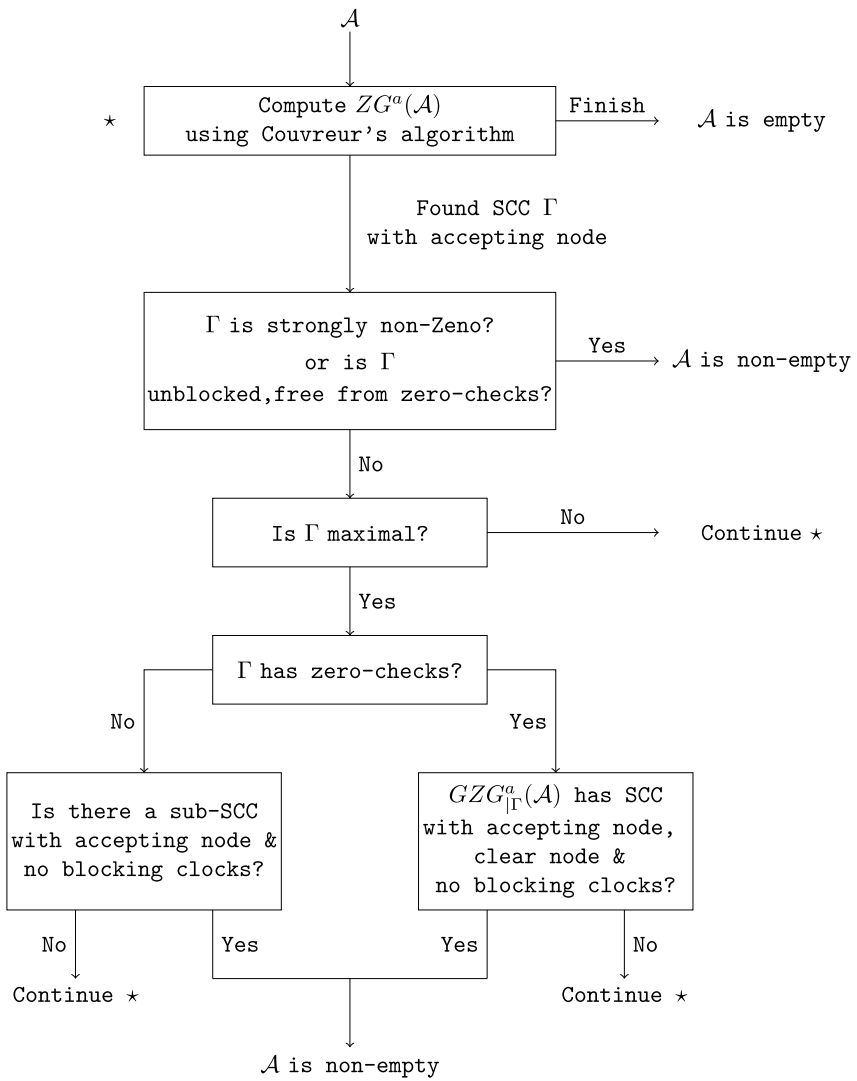
3. Herbreteau, F., Srivathsan, B., & Walukiewicz, I. (2010, July). Efficient emptiness check for timed büchi automata. In International Conference on Computer Aided Verification (pp. 148-161). Springer, Berlin, Heidelberg.

Guessing zone graph

Main idea

- For each node, track set of clocks $Y \subseteq X$ which can evaluate to zero : (q, Z, Y)
  - In transitions : add a clock x to Y if it is reset
  - Add the constraint $\forall \boldsymbol{x} \in X \setminus Y : \ x > 0$ for every transition
- Change the acceptance to see $Y = \varnothing$ infinitely often.
- On each transition : Either keep Y or reset it $Y = \varnothing$

Guessing zone graph - Algo 1



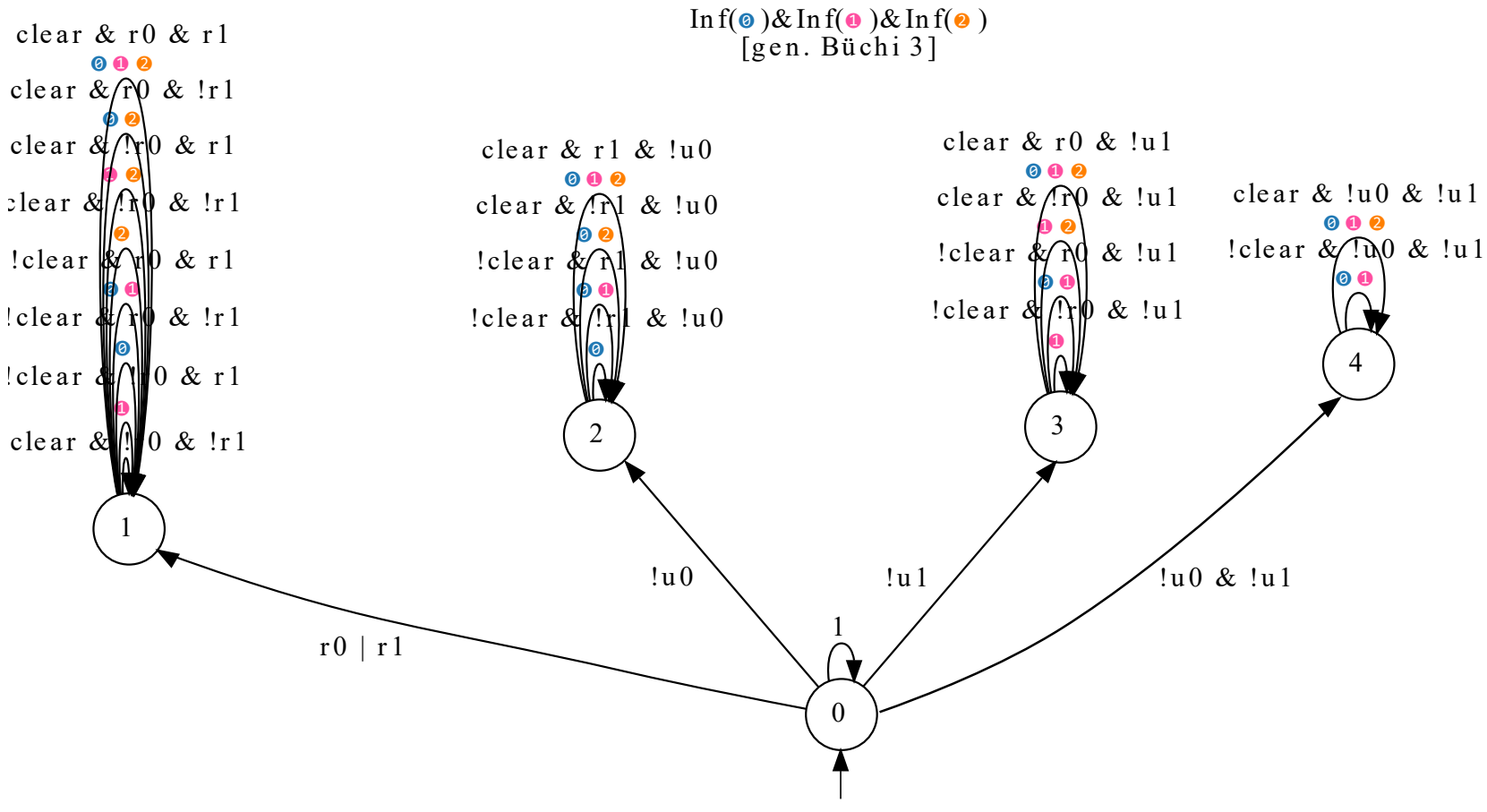
Guessing zone graph - Spec. based

Main idea :
Instead of conceiving a dedicated algorithm,
add the non-zeno requirement to the specification.

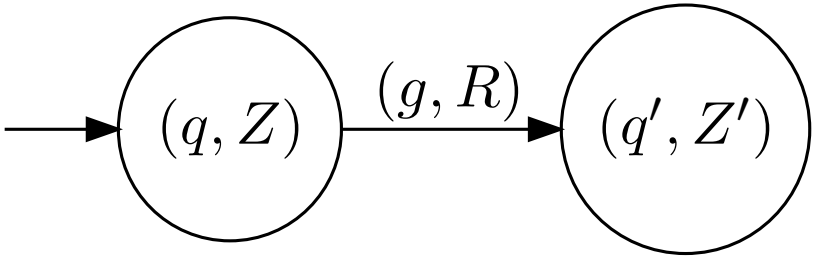
- Clear node has to appear infinitely often
- If a clock encounters infinitely often an upper bound, it also has to be reset infinitely often.

$$spec_{nz} = spec \wedge \mathsf{GF}\, clear \bigwedge_i (\mathsf{FG}\, !up_i | \mathsf{GF}\, res_i)$$

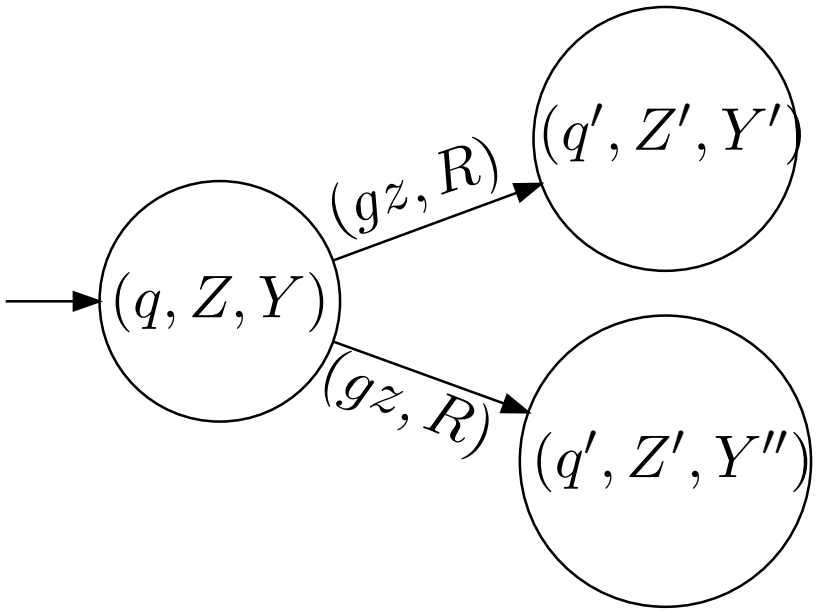
$$\mathsf{GF}\, clear \bigwedge_i (\mathsf{FG}\, !up_i | \mathsf{GF}\, res_i)$$



Modified TA

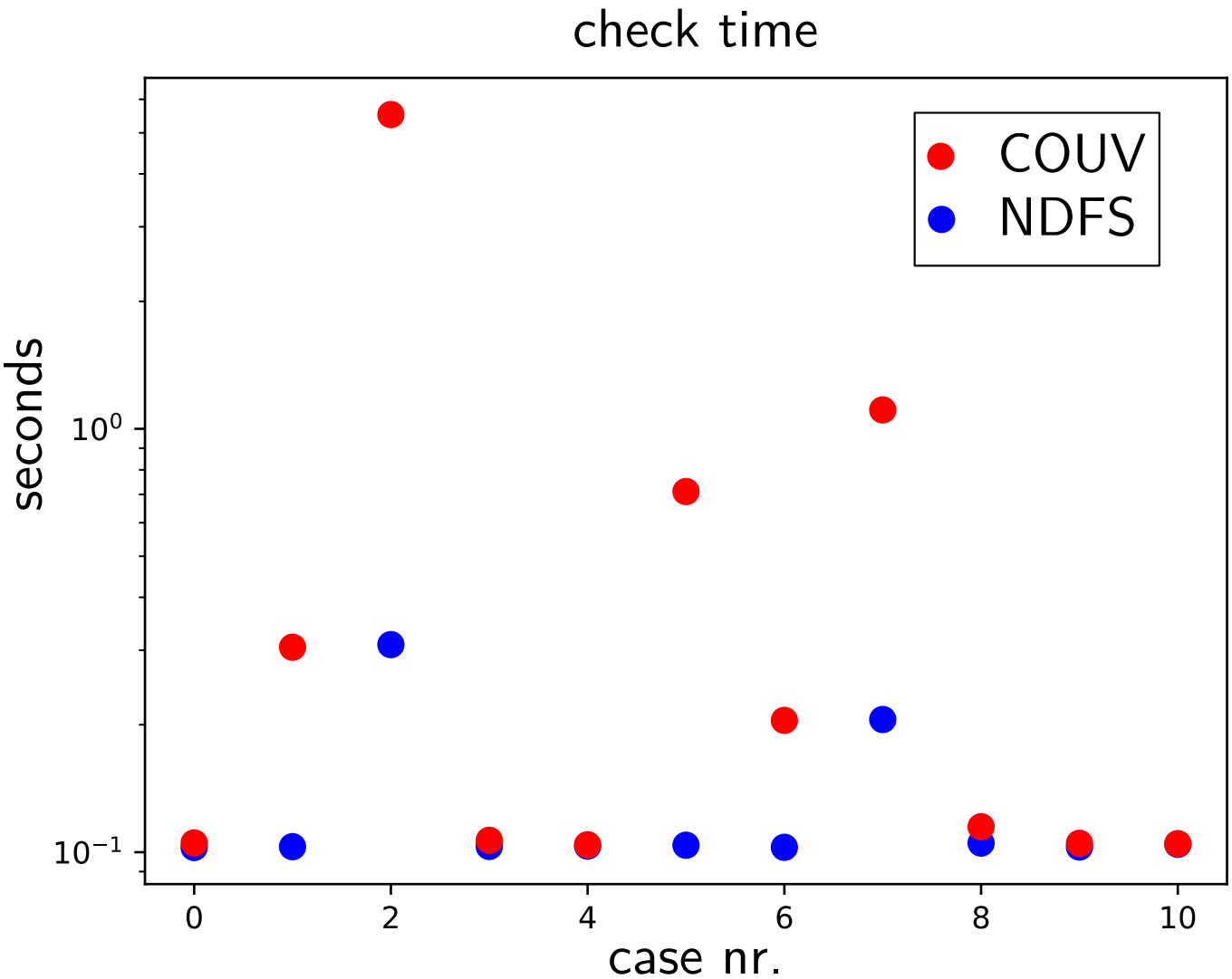


with
$Z' = \overrightarrow{[R](Z \cap g)}$

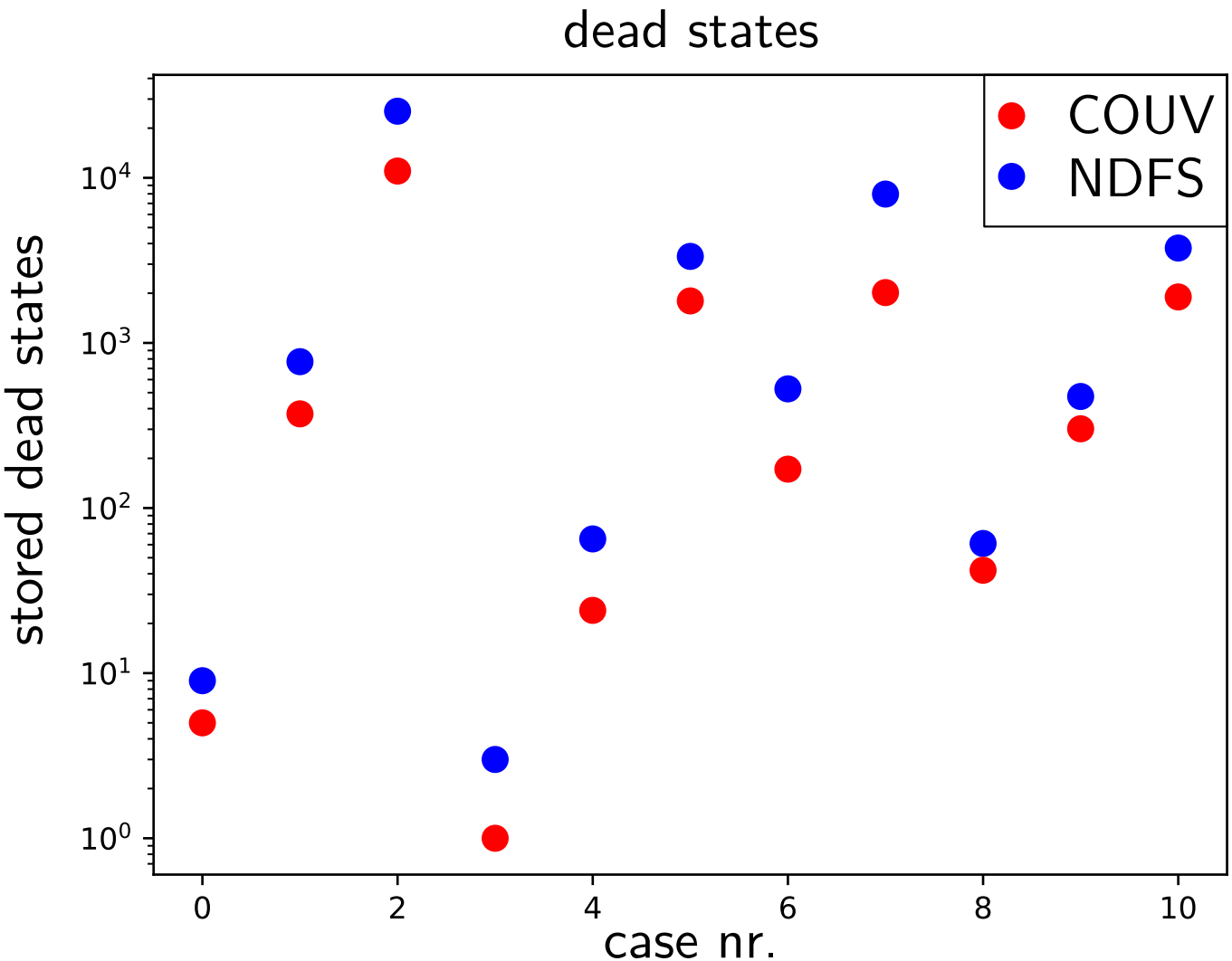


with
$gz = g \cap \{x_i > 0 | x_i \in X \setminus Y\}$
$Z' = \overrightarrow{[R](Z \cap gz)}$
$Y' = Y \cup R$
$Y'' = \varnothing$

Benchmarks 2 - Non-zeno



Benchmarks 2 - Non-zeno



Inclusion relation on Y

- "smaller" $Y \rightarrow$ smaller successor zones
- Y "grows" slowly, but can be reset any time

Inclusion relation

$$(q, Z, Y) \sqsubseteq (q', Z', Y')$$
iff
$$q = q',\ Z \preceq Z' \text{ and } Y \subseteq Y'$$

Inclusion relation on Y

Problem

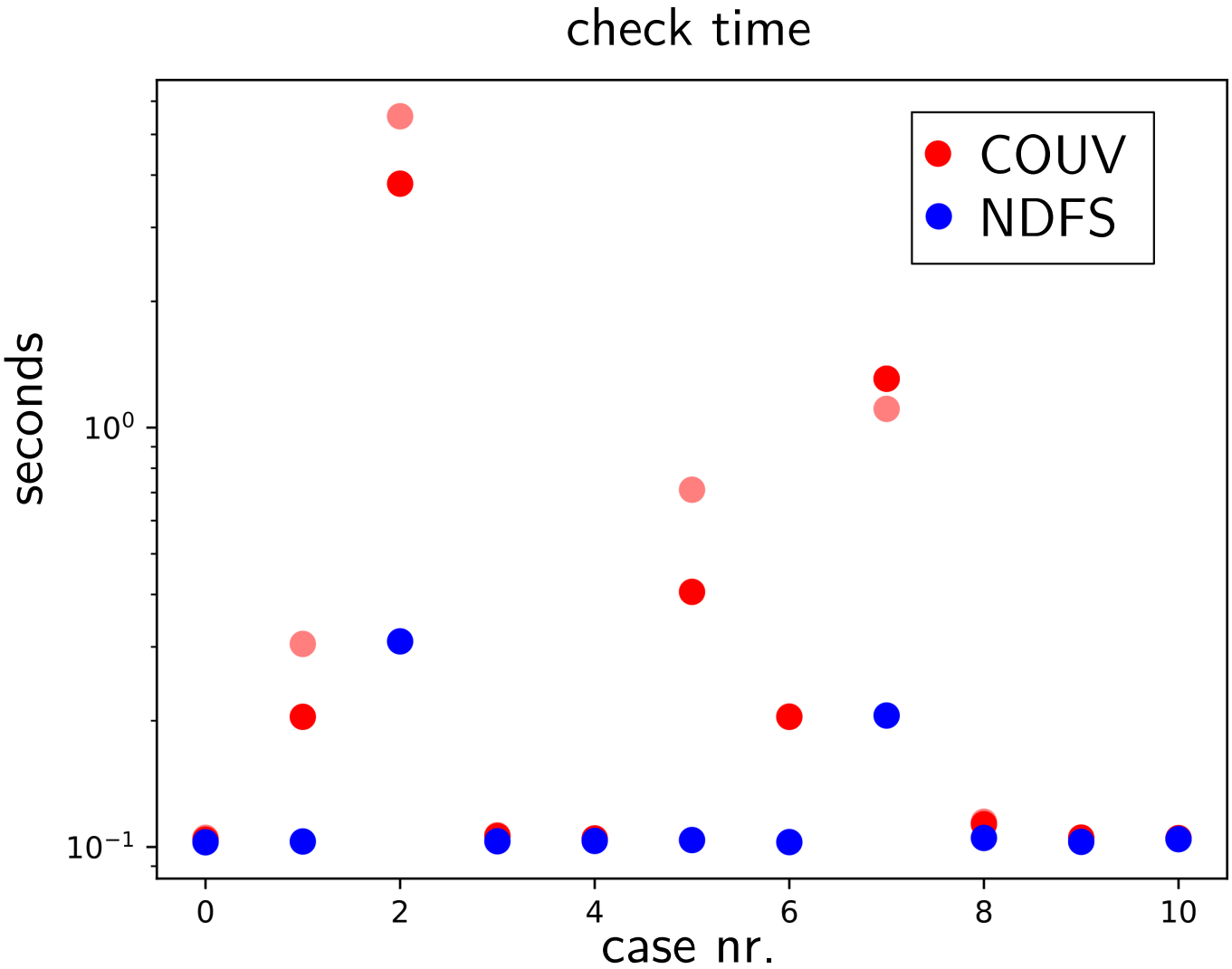
- The inclusion relation disregards the acceptance condition.
- Nodes with $Y = X$ are maximal, but not accepting

For a run to be accepted we have to see the clear node $Y = \varnothing$.

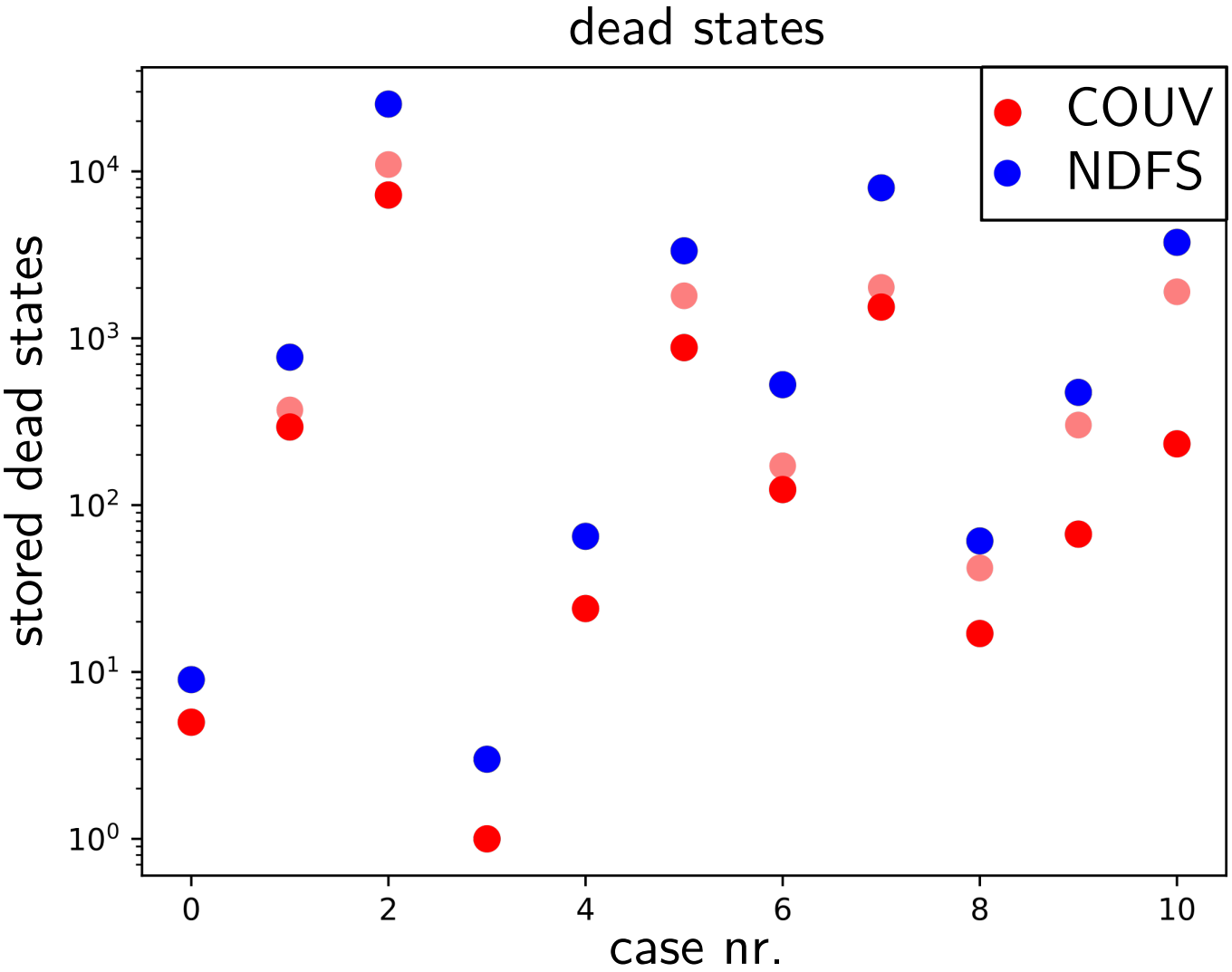
Solution

Always consider the successors with $Y = \varnothing$ first.

Benchmarks 3 - With Inclusion



Benchmarks 3 - With Inclusion



Current work

- Support parallel EC using Bloemen [4]
- Find better formulations for non-zeno condition

4. Bloemen, V., & van de Pol, J. (2016, November). Multi-core SCC-based LTL model checking. In Haifa Verification Conference (pp. 18-33). Springer, Cham.