# Kaki: Concurrent Update Synthesis for Regular Policies via Petri Games

March 31, 2023

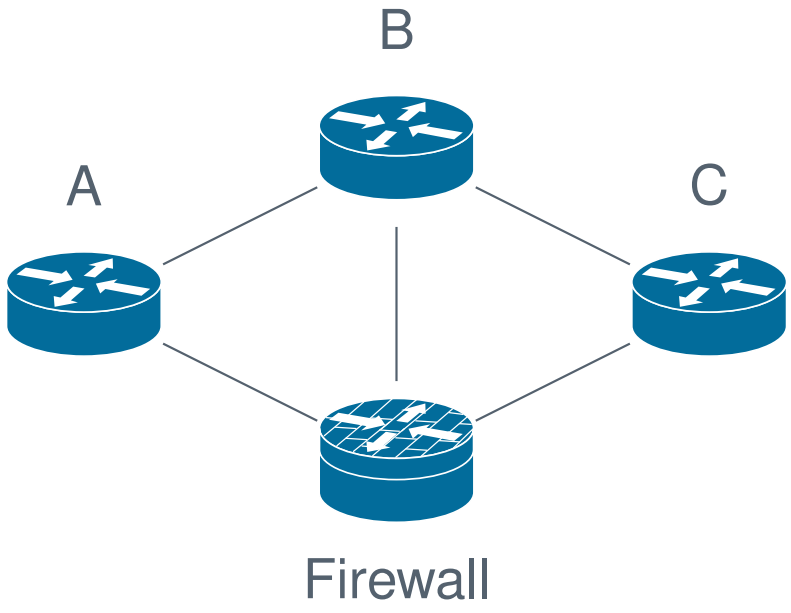
Nicklas S. Johansen, Lasse B. Kær, Andreas L. Madsen, Kristian Ø. Nielsen, Jiri Srba, and Rasmus G. Tollund

Department of Computer Science
Aalborg University
Kingdom of Denmark

AALBORG UNIVERSITY
DENMARK

# Communication Network



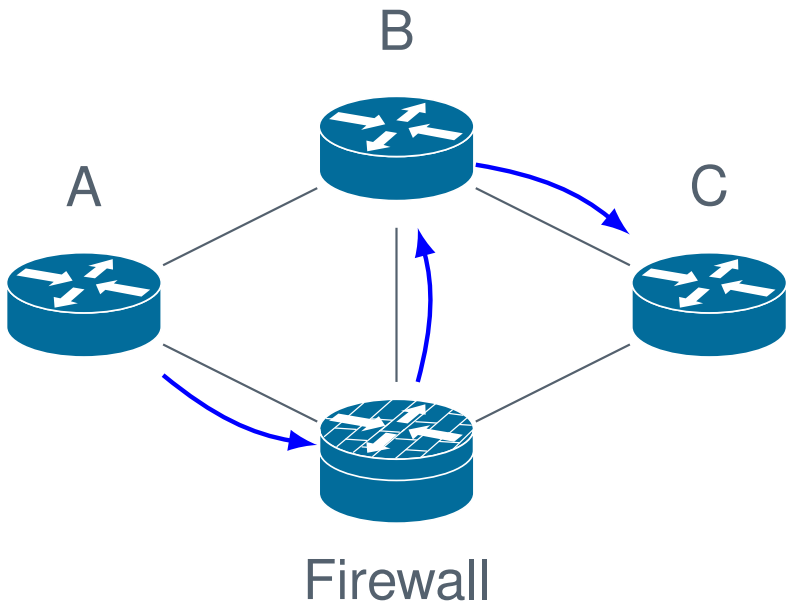



- Switches send packets across a network
- A flow is a source and destination pair for packets
- In Software Defined Networking a central controller manages traffic

# Network Update



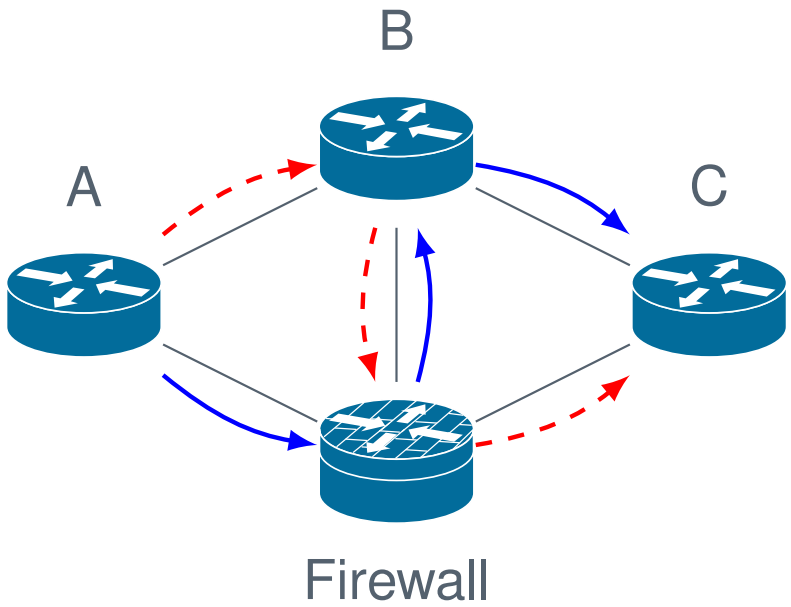
- Flow from A to C
- Initial forwarding rules



# Network Update



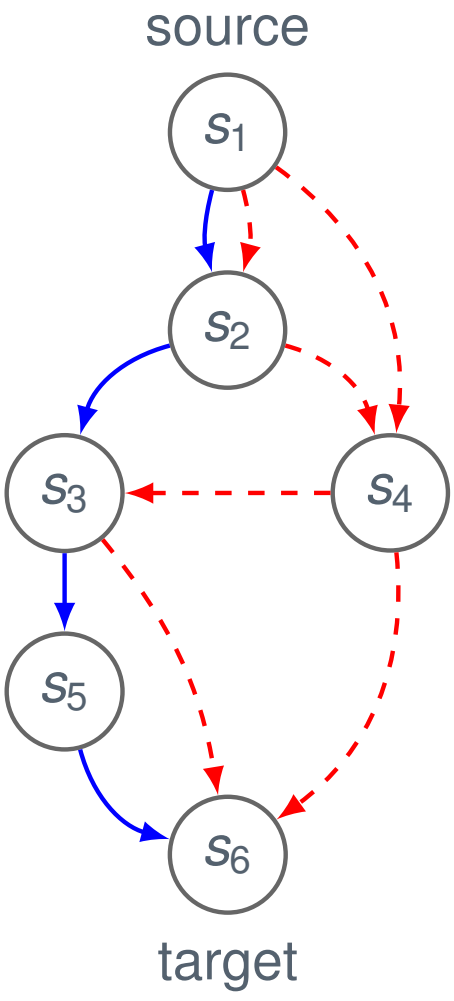
- Flow from A to C
- Initial forwarding rules
- New desired routing



# Routing



- Network
- Initial Routing
  - Trace: $s_1 s_2 s_3 s_5 s_6$
- Final Routing
  - 4 different traces
- Intermediate Routing



# Policy

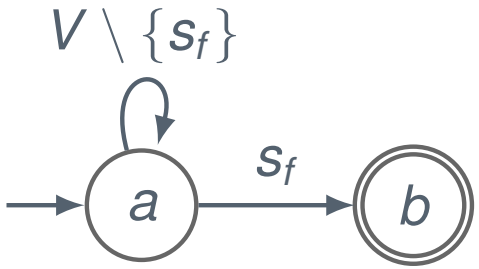


- A policy $P$ is a regular expression over switches $V$, describing the language $\mathcal{L}(P) \subseteq V^*$
- A policy $P$ is satisfied for a routing $R$, if all traces for $R$ is in $\mathcal{L}(P)$

# Policy



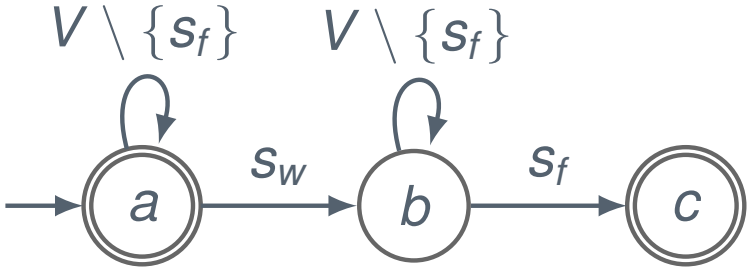
- A policy $P$ is a regular expression over switches $V$, describing the language $\mathcal{L}(P) \subseteq V^*$
- A policy $P$ is satisfied for a routing $R$, if all traces for $R$ is in $\mathcal{L}(P)$
- Policy examples
  - Reachability



# Policy



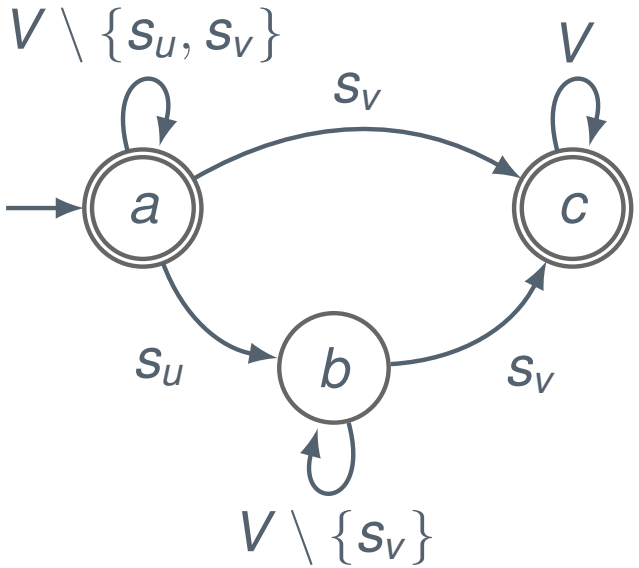
- A policy $P$ is a regular expression over switches $V$, describing the language $\mathcal{L}(P) \subseteq V^*$
- A policy $P$ is satisfied for a routing $R$, if all traces for $R$ is in $\mathcal{L}(P)$
- Policy examples
  - Reachability
  - Waypoint



# Policy



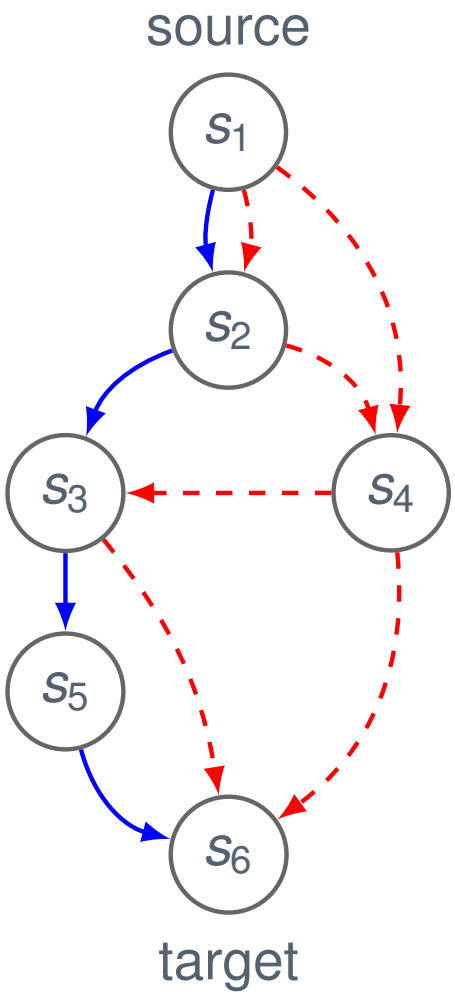
- A policy $P$ is a regular expression over switches $V$, describing the language $\mathcal{L}(P) \subseteq V^*$
- A policy $P$ is satisfied for a routing $R$, if all traces for $R$ is in $\mathcal{L}(P)$
- Policy examples
  - Reachability
  - Waypoint
  - Conditional enforcement ($s_u \Rightarrow s_v$)



# Concurrent Update Synthesis Problem



Policy: $V^*\{s_5, s_4\}V^*s_6$



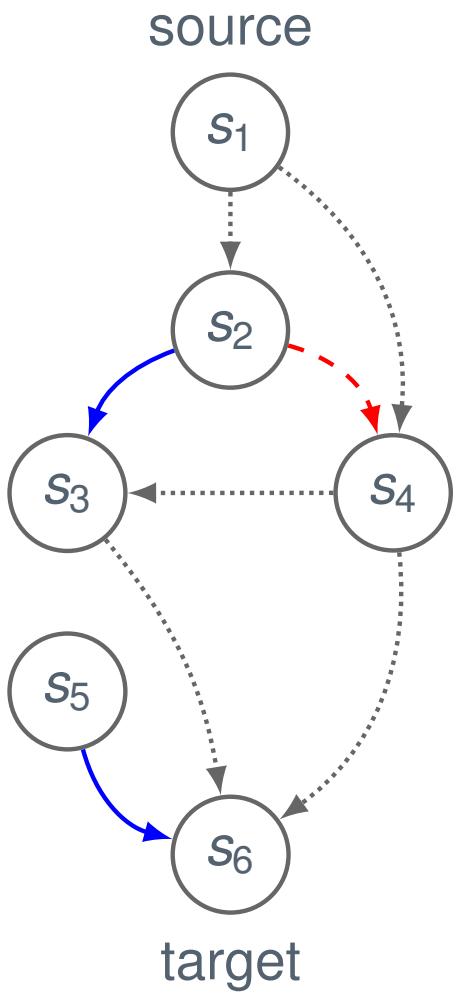
# Concurrent Update Synthesis Problem



Policy: $V^*\{s_5, s_4\}V^*s_6$

Concurrent Update Sequence:

- Batch 1: $\{s_4\}$
- Batch 2: $\{s_1, s_2, s_3\}$?
  - Six different permutations
  - $\mathbf{s_1} \rightarrow \mathbf{s_3} \rightarrow s_2$ policy violation!
  - Violating trace: $s_1 s_2 s_3 s_6 \notin \mathcal{L}(P)$



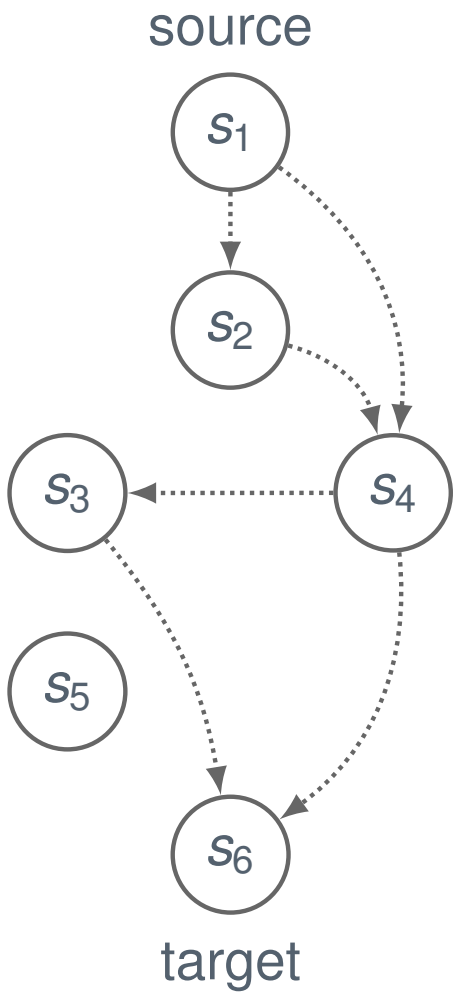
# Concurrent Update Synthesis Problem



Policy: $V^*\{s_5, s_4\}V^*s_6$

Concurrent Update Sequence:

- Batch 1: $\{s_4\}$
- Batch 2: $\{s_1, s_2\}$
- Batch 3: $\{s_3\}$
- Batch 4: $\{s_5\}$
- $\omega = \{s_4\}\{s_1, s_2\}\{s_3\}\{s_5\}$



# Translation to Petri Game

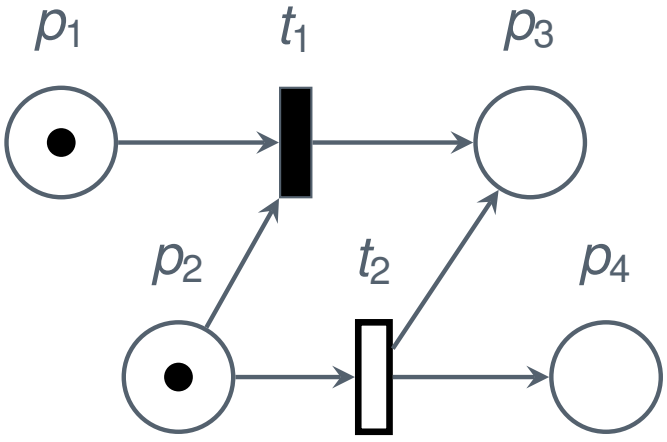


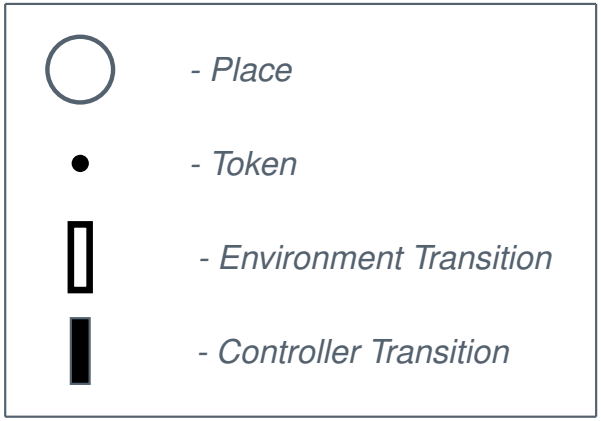
- Encode CUSP as Petri game
- Find winning strategy for controller in Petri game
- Extract *concurrent update sequence* from strategy

Strategy synthesis using TAPAAL Petri game engine

# Petri Game
## Translation



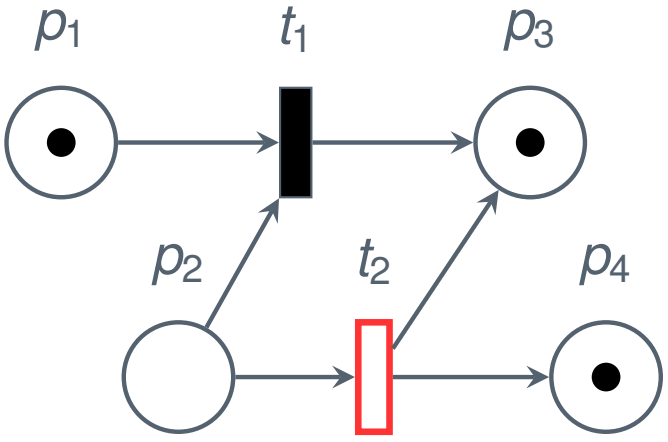


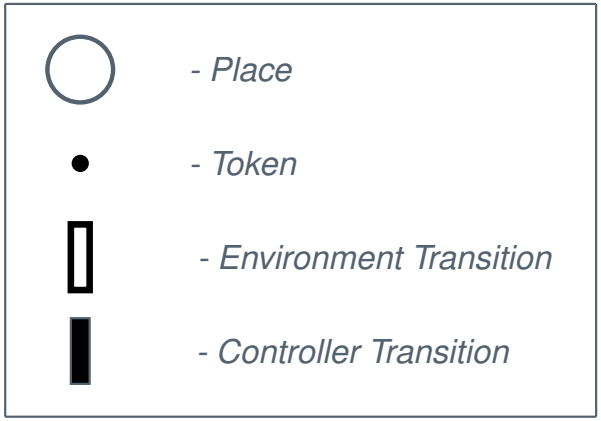




CTL query: $AF\ p3 = 1$

# Petri Game
## Translation



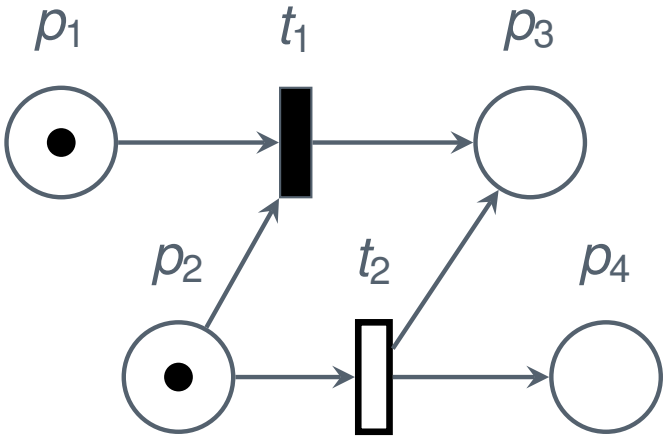


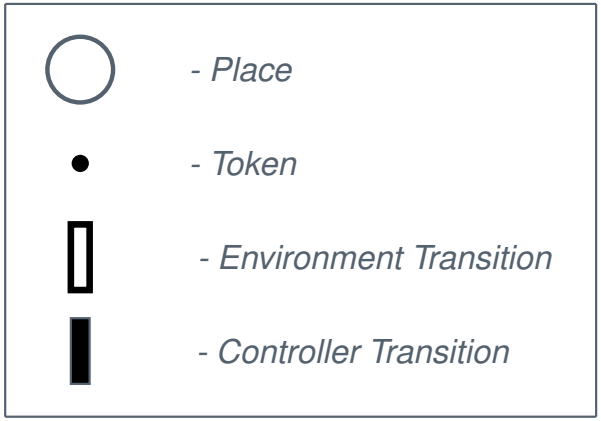




CTL query: $AF\ p3 = 1$

# Petri Game
## Translation



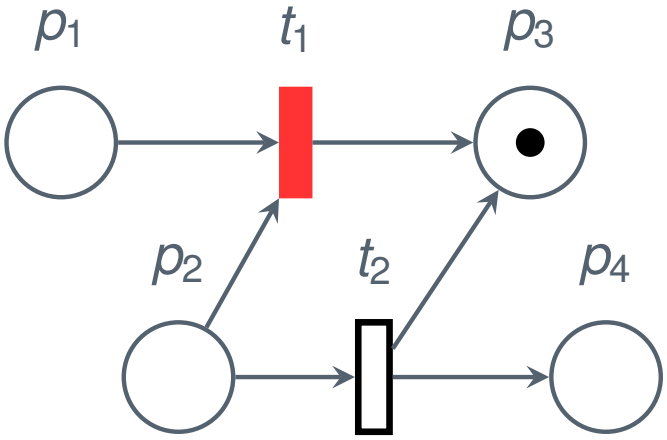


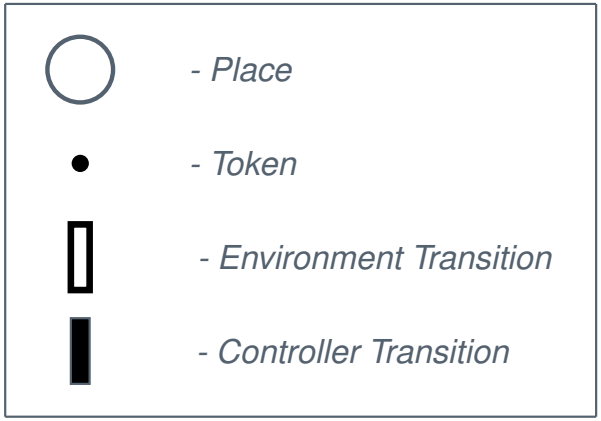




CTL query: $AF\ p3 = 1$

# Petri Game
## Translation



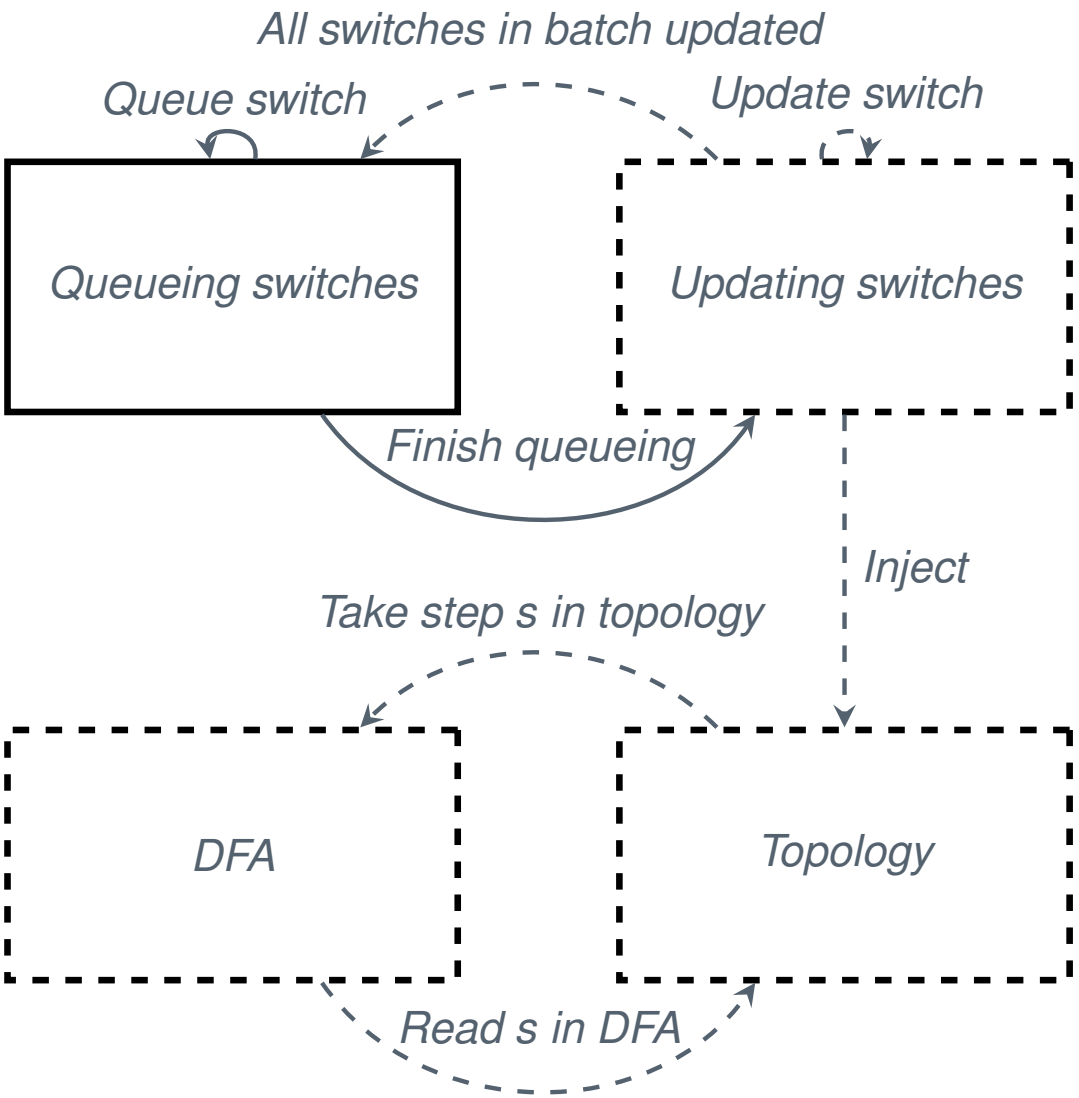


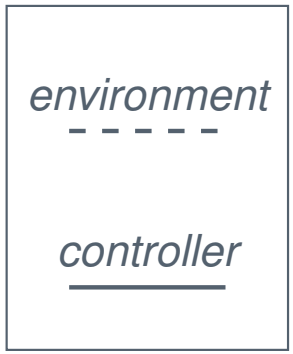




CTL query: $AF\ p3 = 1$

# Flowchart for Petri Game



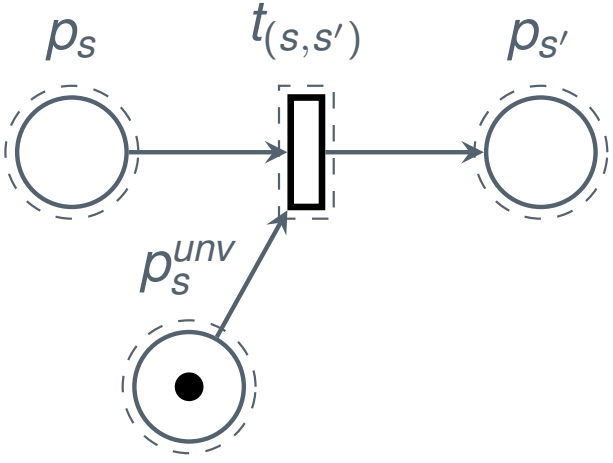


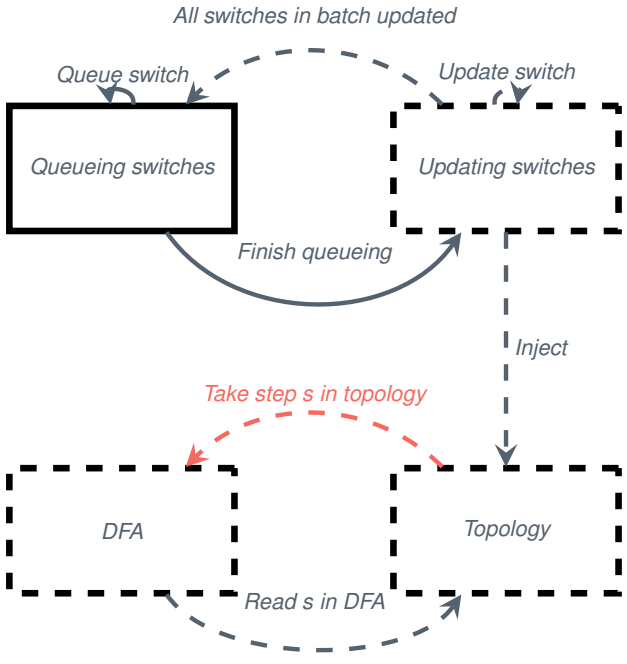



# Topology Step
## Translation



For $s, s' \in V$ where $s' \in R_i(s) \cup R_f(s)$

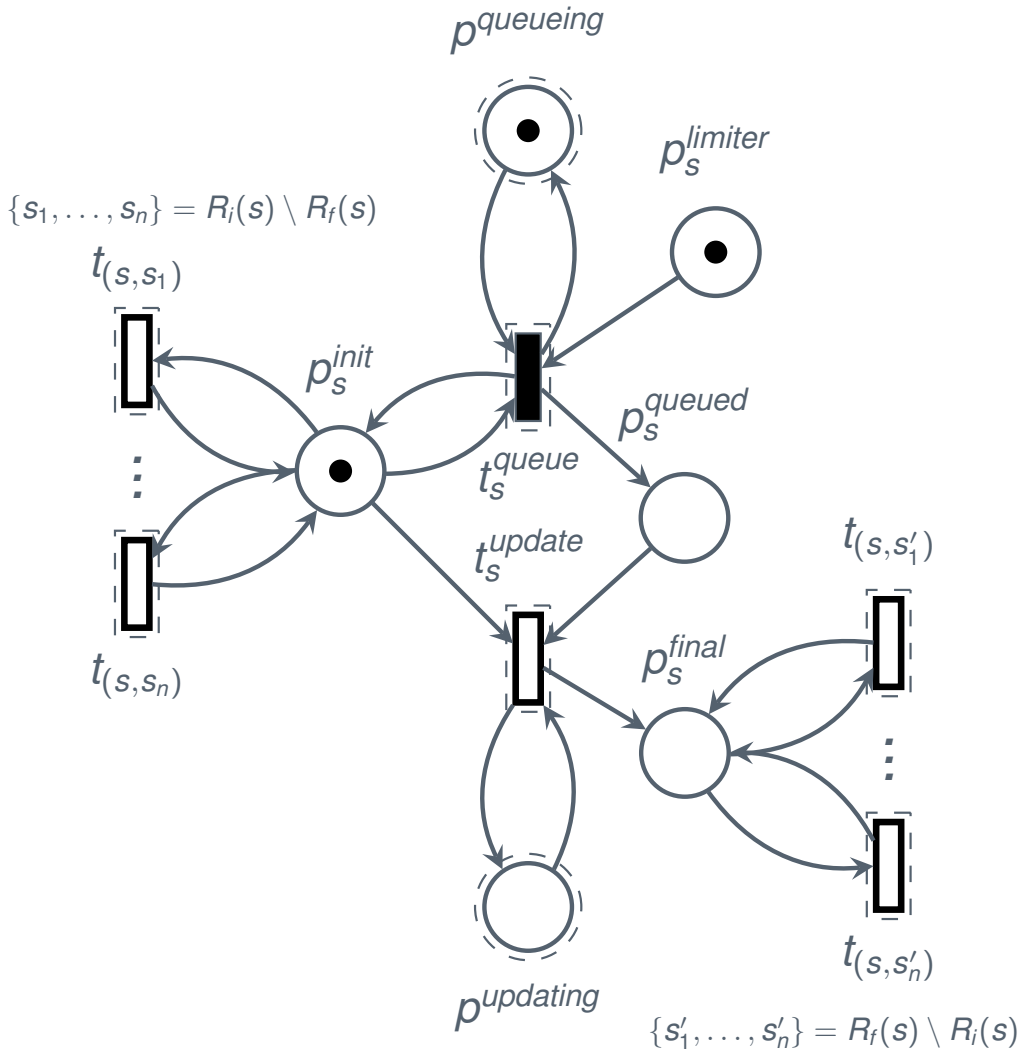


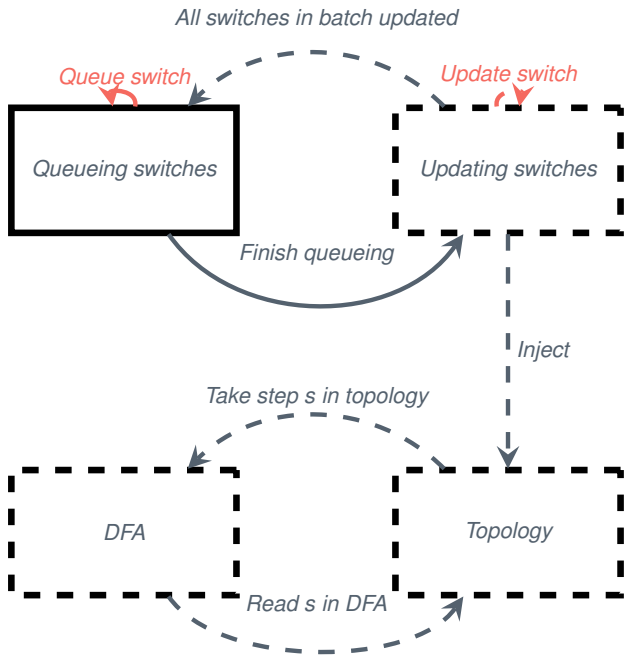




- A routing from $s$ to $s'$ is encoded by a transition $t_{(s,s')}$
- The policy is encoded as a DFA in the Petri game
- The topology and DFA components are synchronised

# Switch Component
## Translation





# Query
## Translation



- Solve CUSP by finding a winning strategy for controller

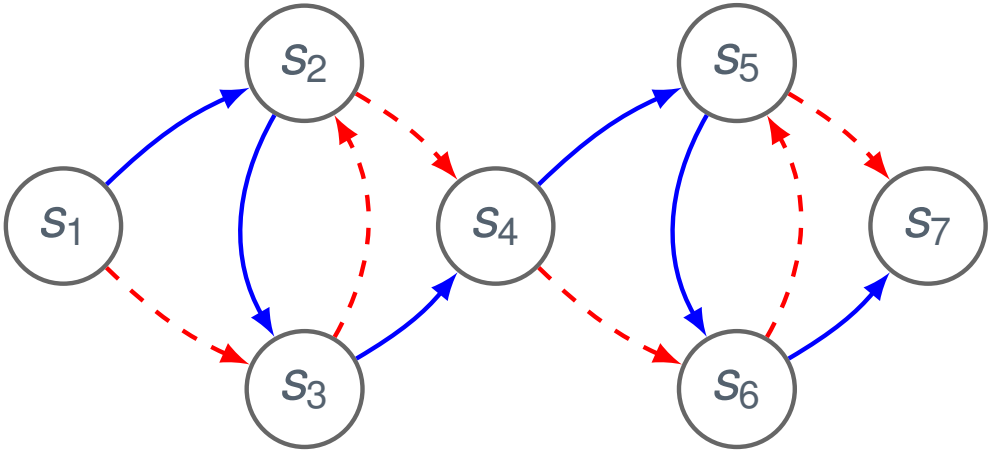
$$AF\left(p_{batches} \leq k \wedge \left(\bigvee_{q \in F} p_q = 1\right)\right)$$

- Bisection search to find lowest $k$
- Proved to find optimal solution to CUSP

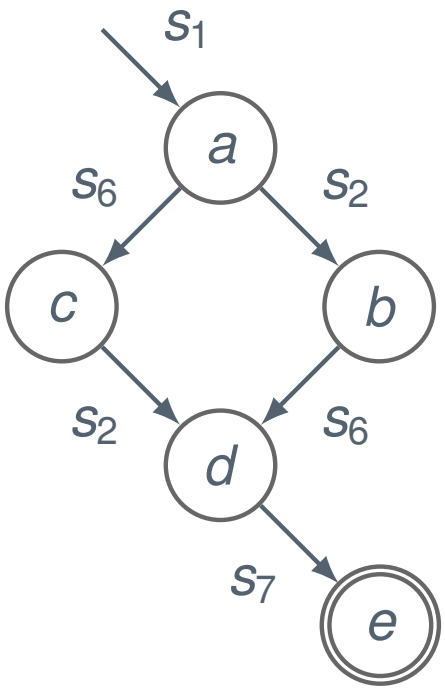
# Topological Decomposition



- Split problem into two independent subproblems
- Find potential NFA states at switches



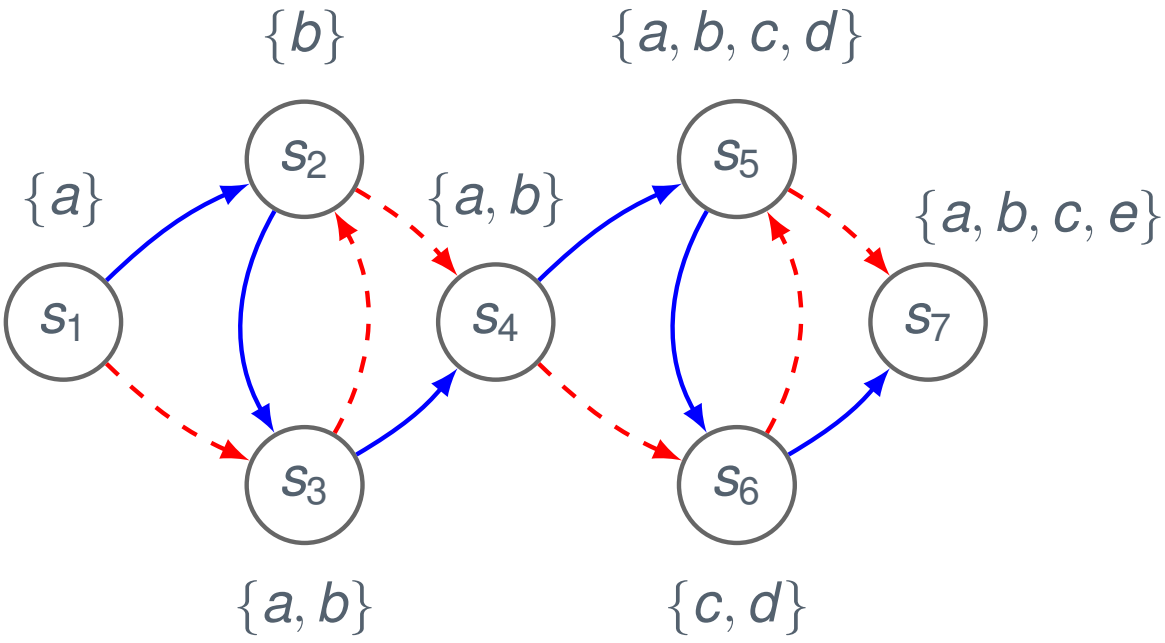

Waypoint: $s_2, s_6$



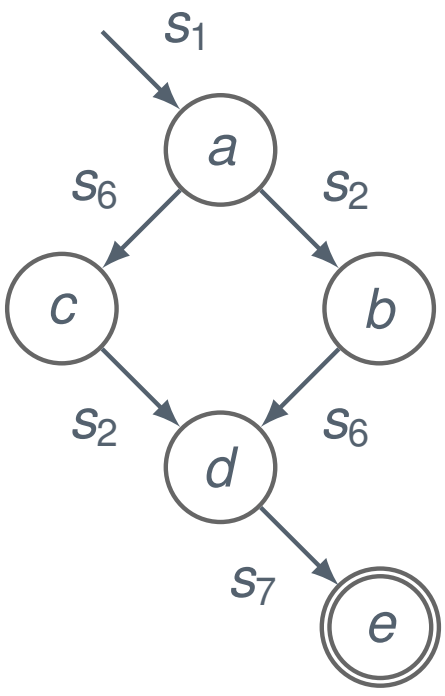
# Topological Decomposition



- Split problem into two independent subproblems
- Find potential NFA states at switches



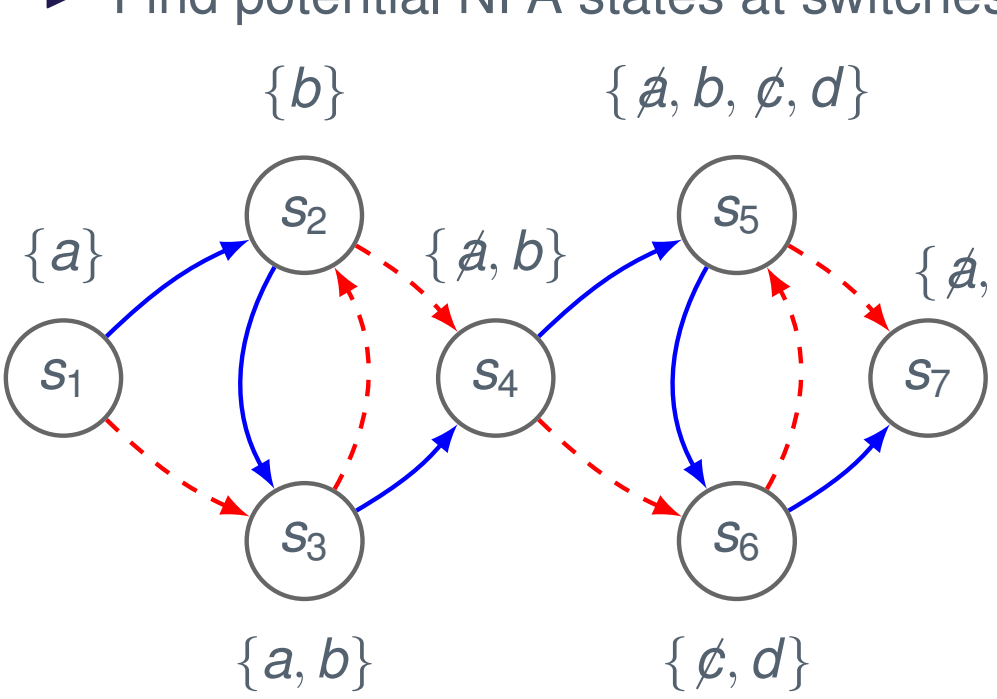

Waypoint: $s_2, s_6$



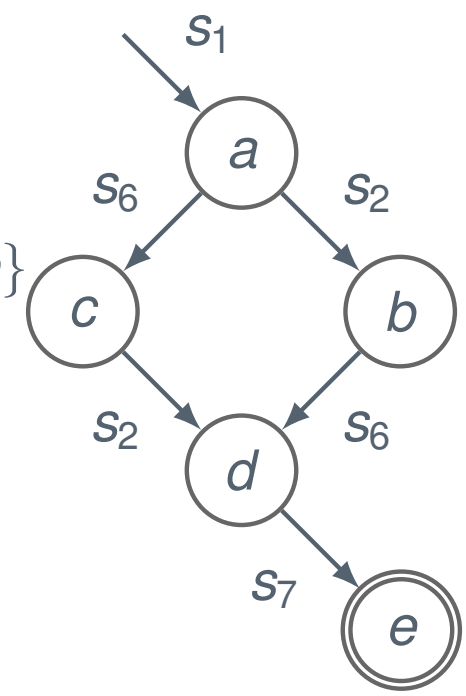
# Topological Decomposition



- Split problem into two independent subproblems
- Find potential NFA states at switches



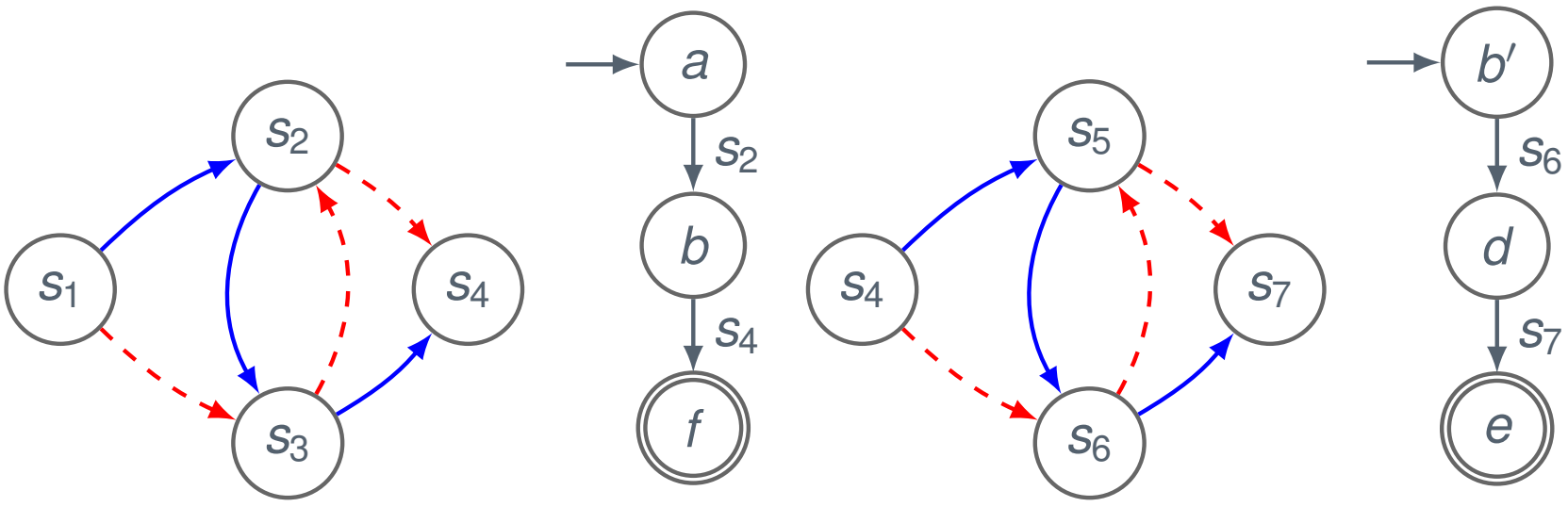

Waypoint: $s_2, s_6$




Switch $s_4$ is a topological decomposition point!

# Subproblems
## Topological Decomposition






- Original CUSP is split into 2 smaller CUSPs
- Proven: Combine optimal solution for subproblems into an optimal solution to the original problem

# Comparison with FLIP

## Results



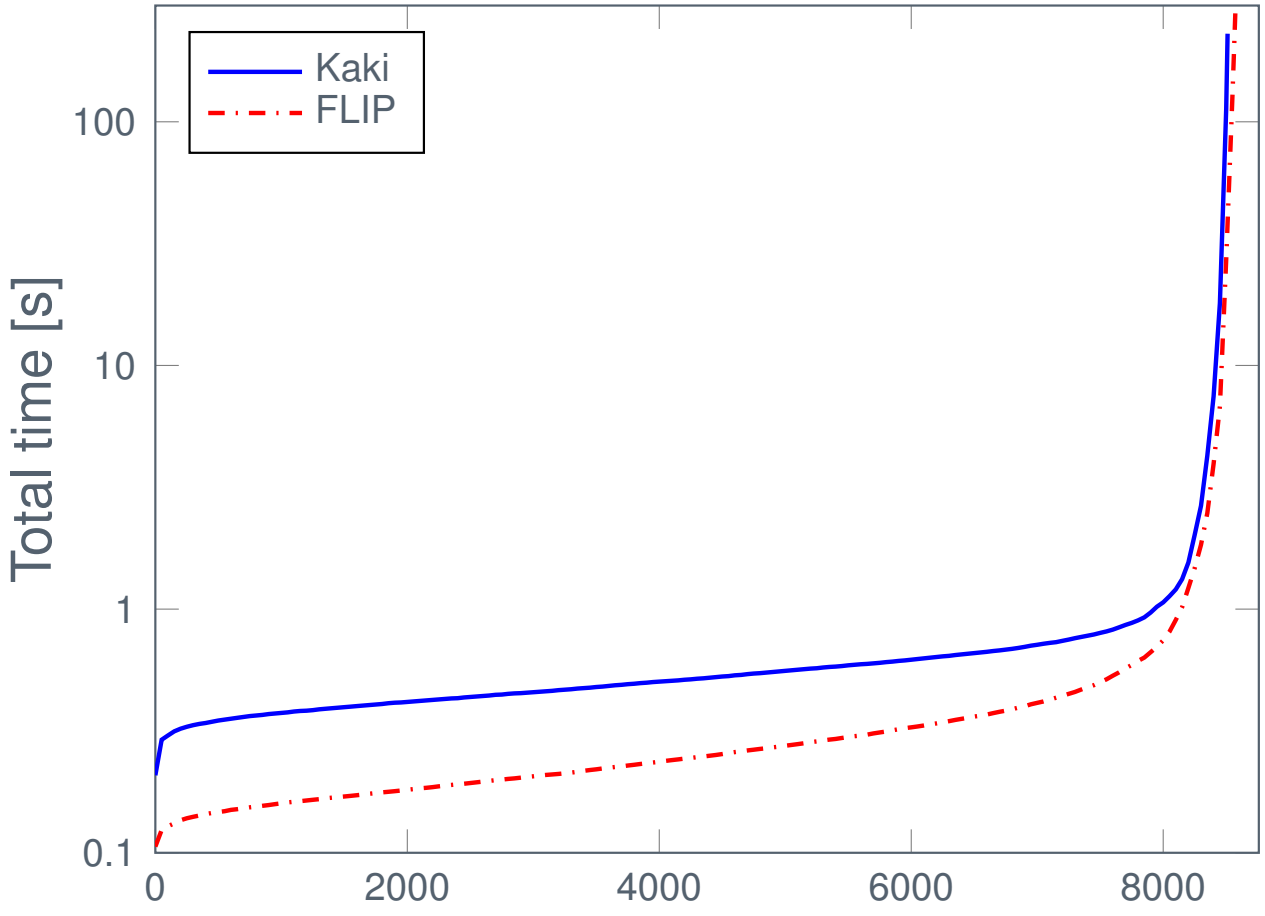



Figure: Kaki benchmarked against FLIP

| | All Problems |
|---|---|
| Total | 8759 |
| Only Kaki | 133 |
| Only FLIP | 196 |
| Suboptimal | 787 |
| Tagging | 268 |

# Comparison with NetStack

## Results



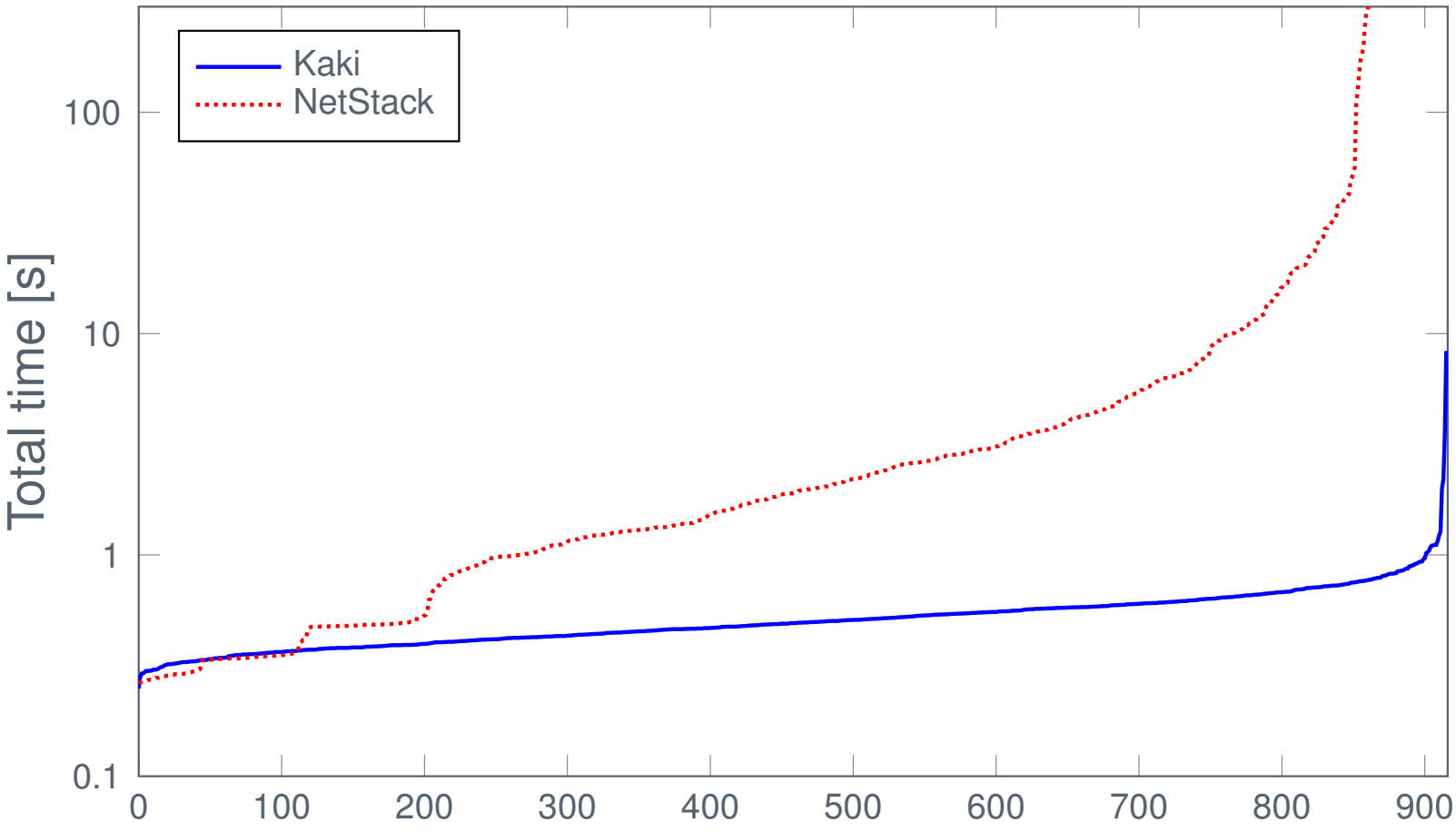



Figure: Comparion with NetStack

# Conclusion



- Kaki capabilities
  - Find optimal concurrent updates
  - Supports splittable flows
  - Supports arbitrary regular policies
- Performed on par with FLIP

# Conclusion


- Kaki capabilities
  - Find optimal concurrent updates
  - Supports splittable flows
  - Supports arbitrary regular policies
- Performed on par with FLIP
- Faster if subproblems are solved parallel

Presentation Concluded

# Comparison with SegPG

## Results



- Kaki limited to 1 switch per batch
- SeqPG unable to decide 2 problems

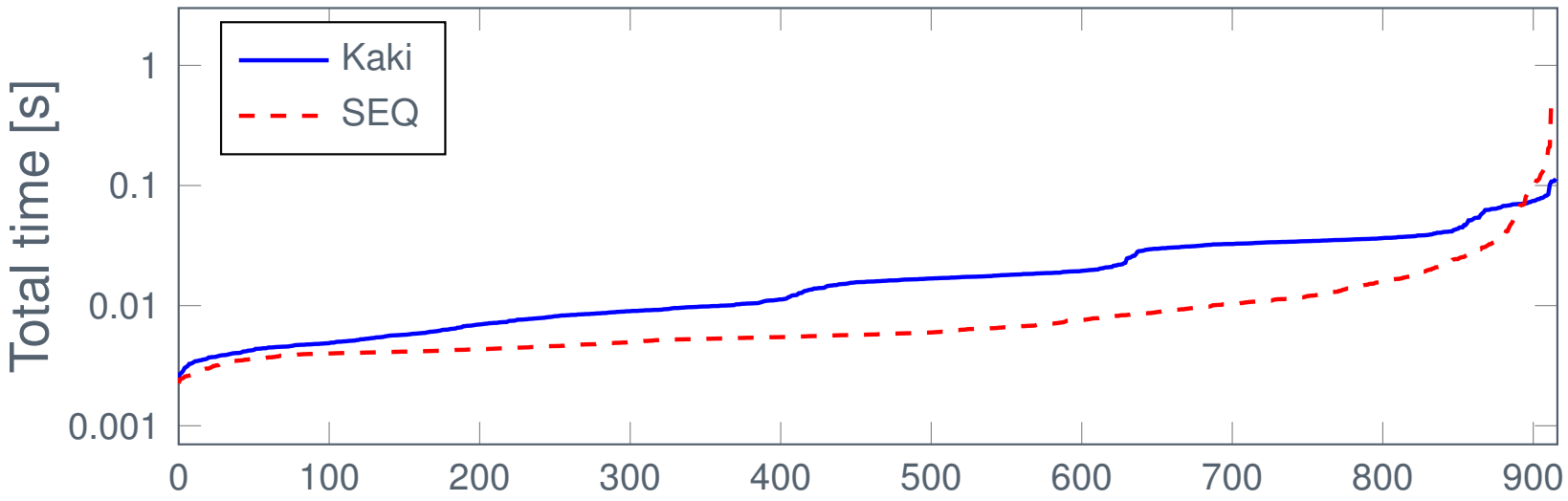



Figure: Kaki benchmarked against SeqPG

# Splitable Flows
## Results



- Comparison with ourselves
- Small overhead

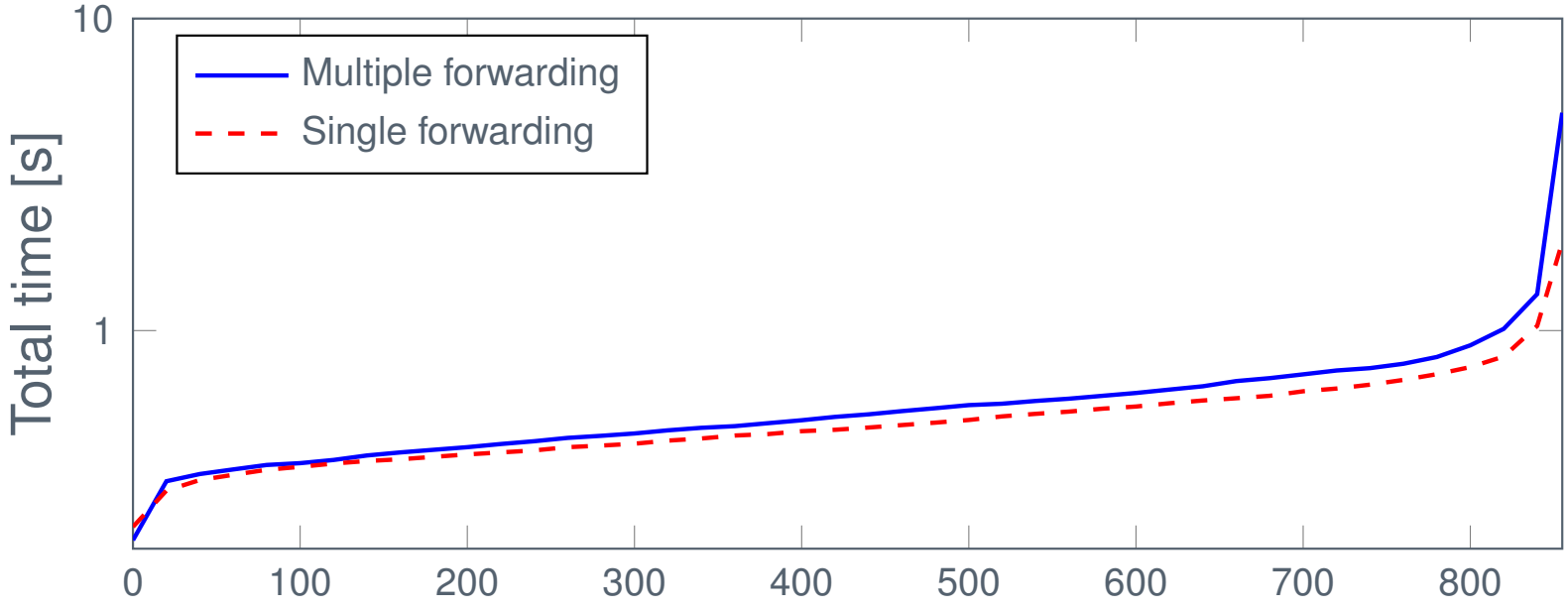



Figure: Splitable flows experiments for Kaki

# Preprocessing Techniques

## Results



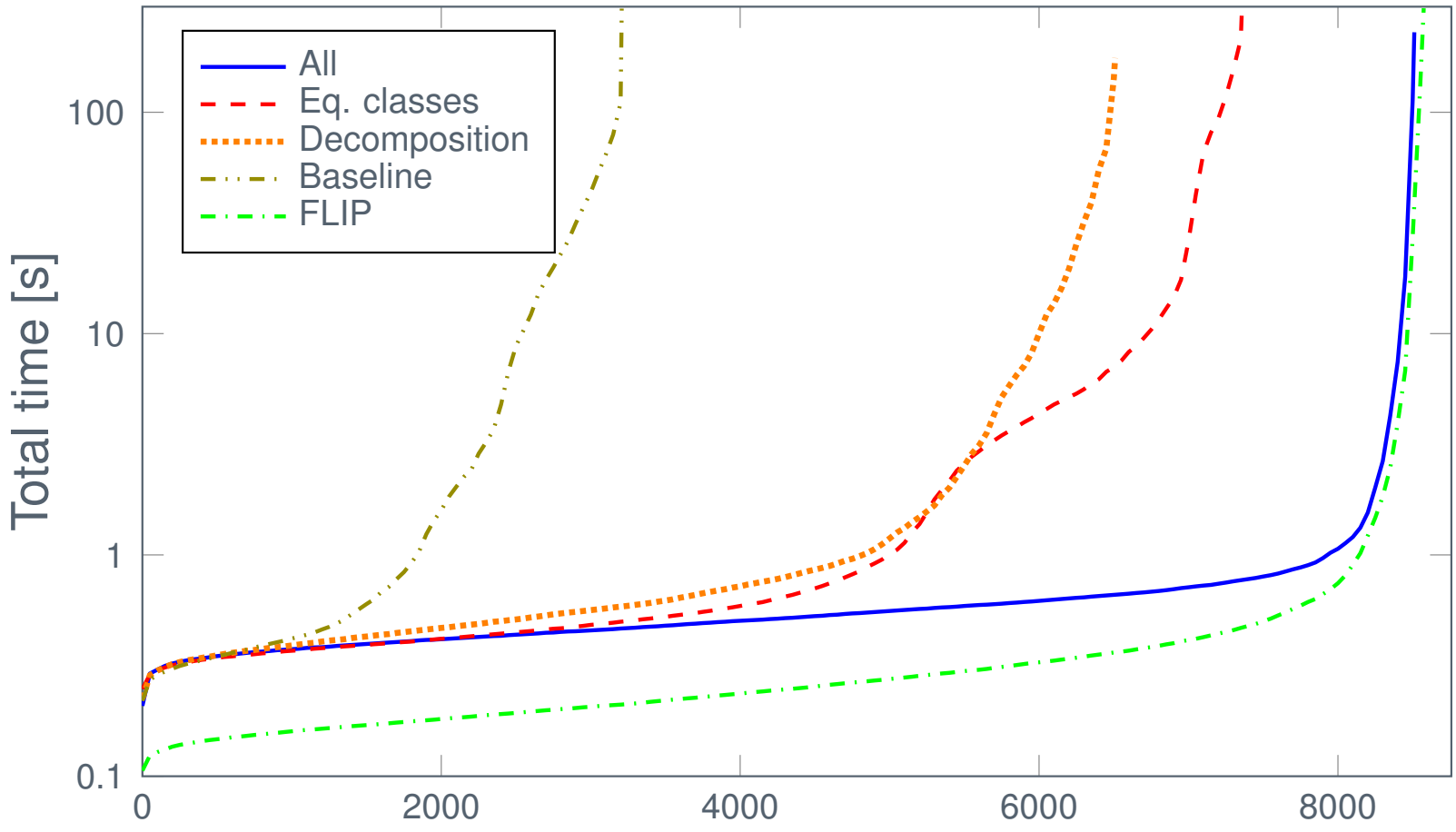
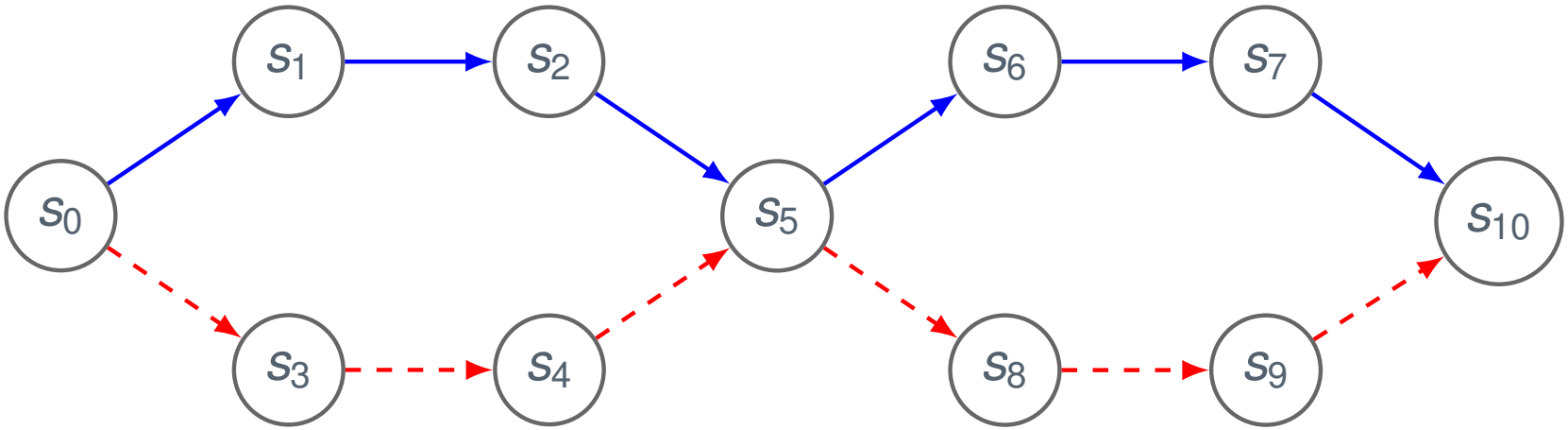



Figure: Experiments of the preprocessing techniques of Kaki

# Update Equivalence Classes



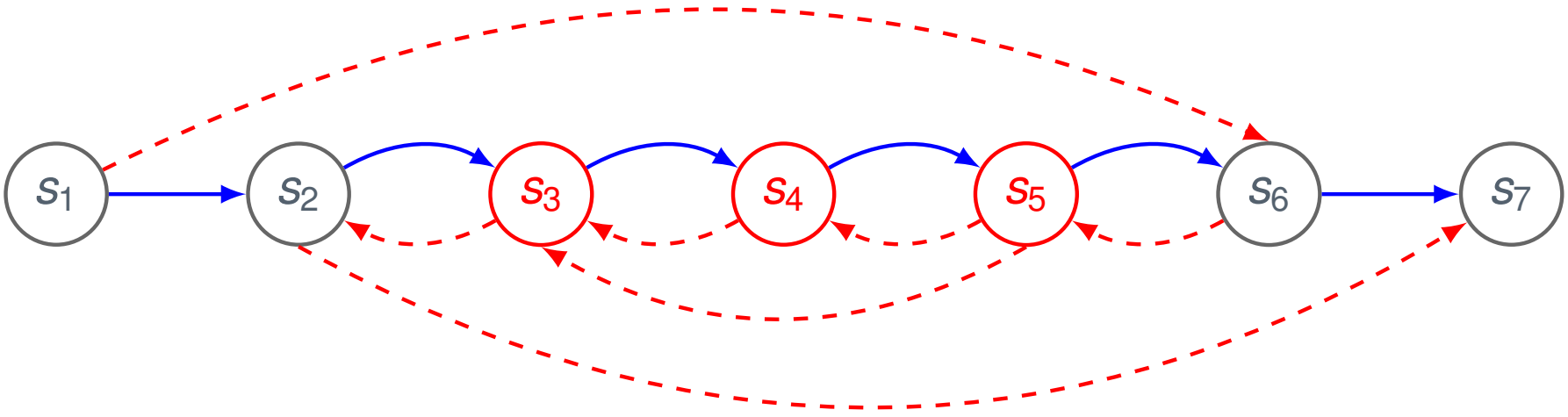



Always put $s_3, s_4, s_8, s_9$ in the first batch
Always put $s_1, s_2, s_6, s_7$ in the last batch

# Update Equivalence Classes

## Chain reduction






Switches $s_3, s_4, s_5$ can always be updated in the same batch