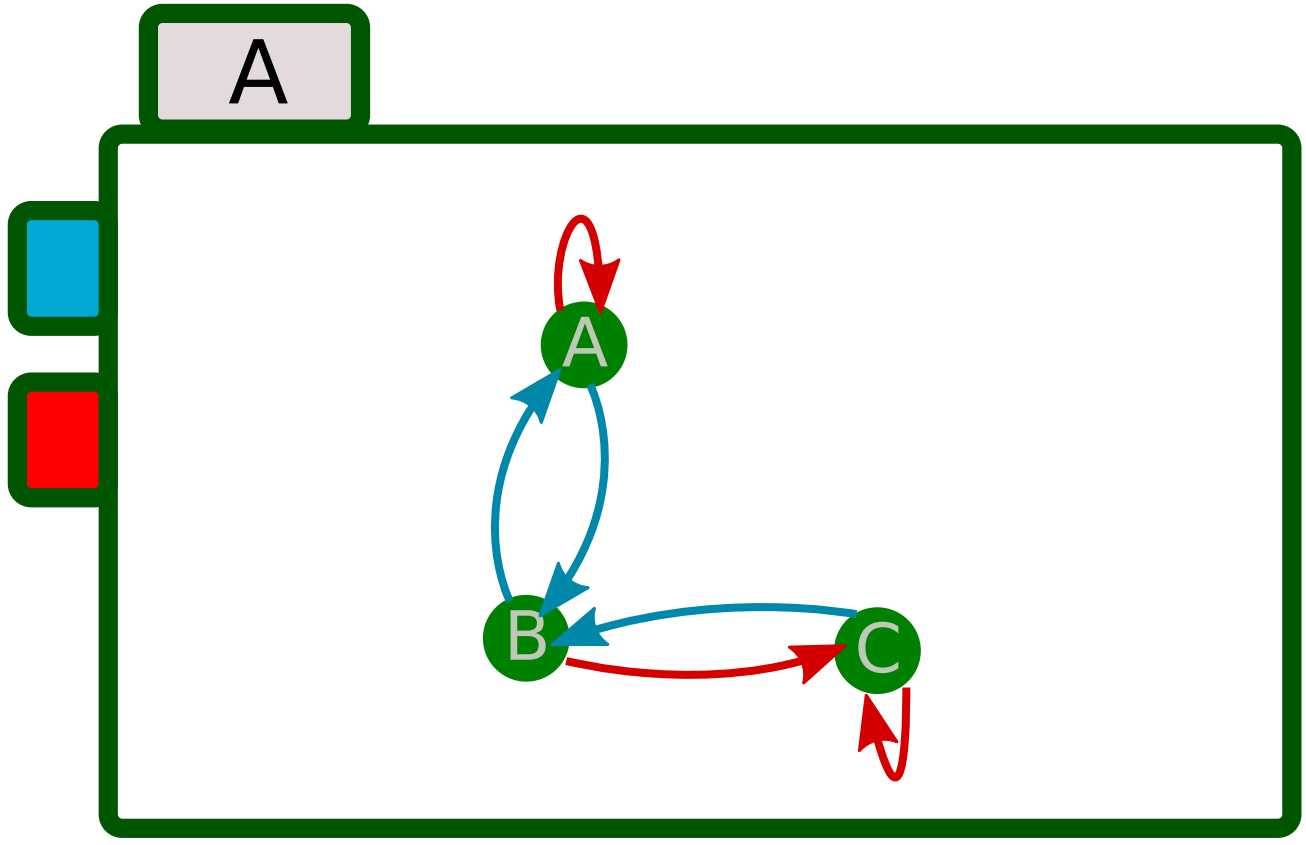
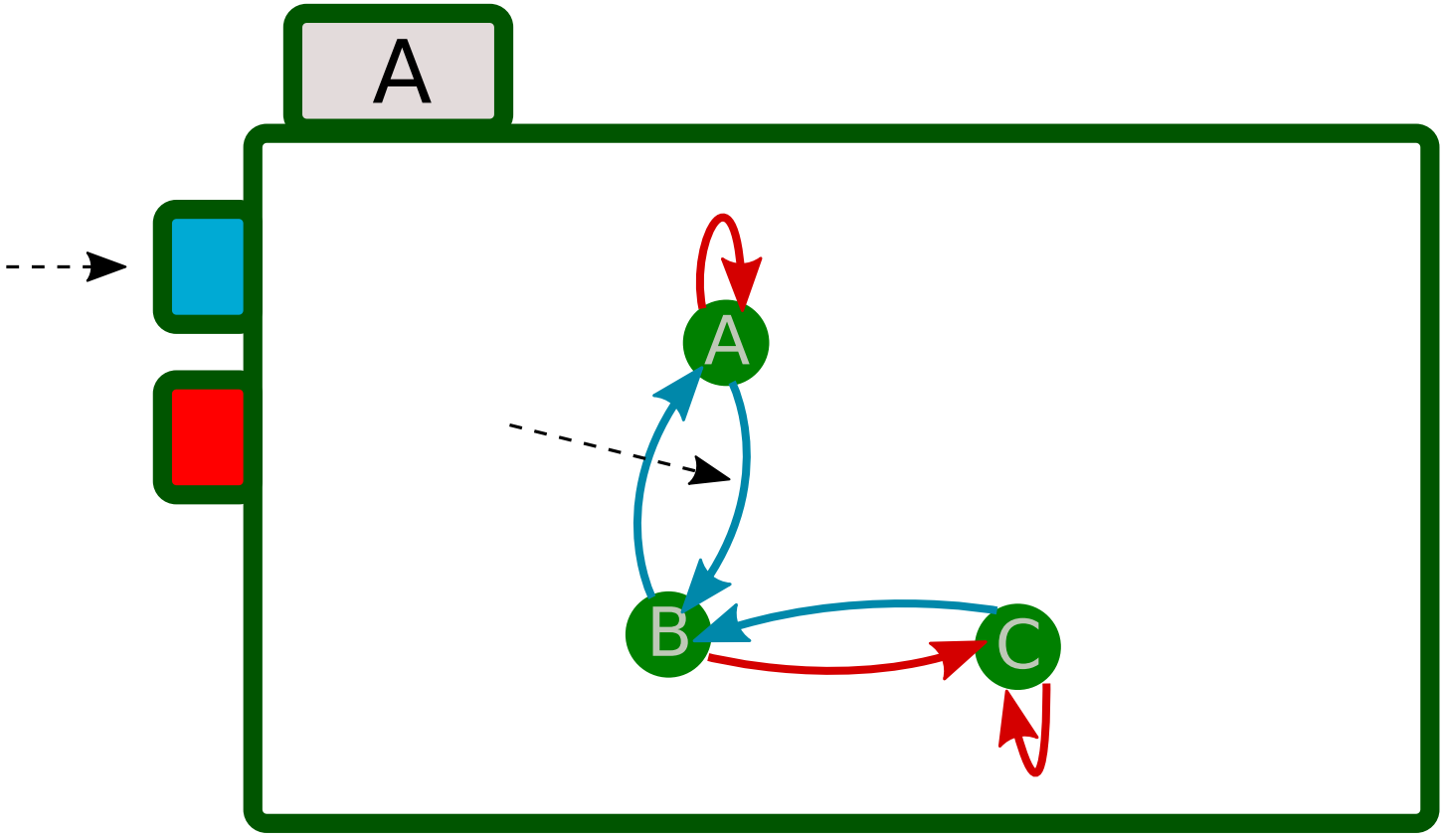
HayDi: Generating discrete structures & HPC

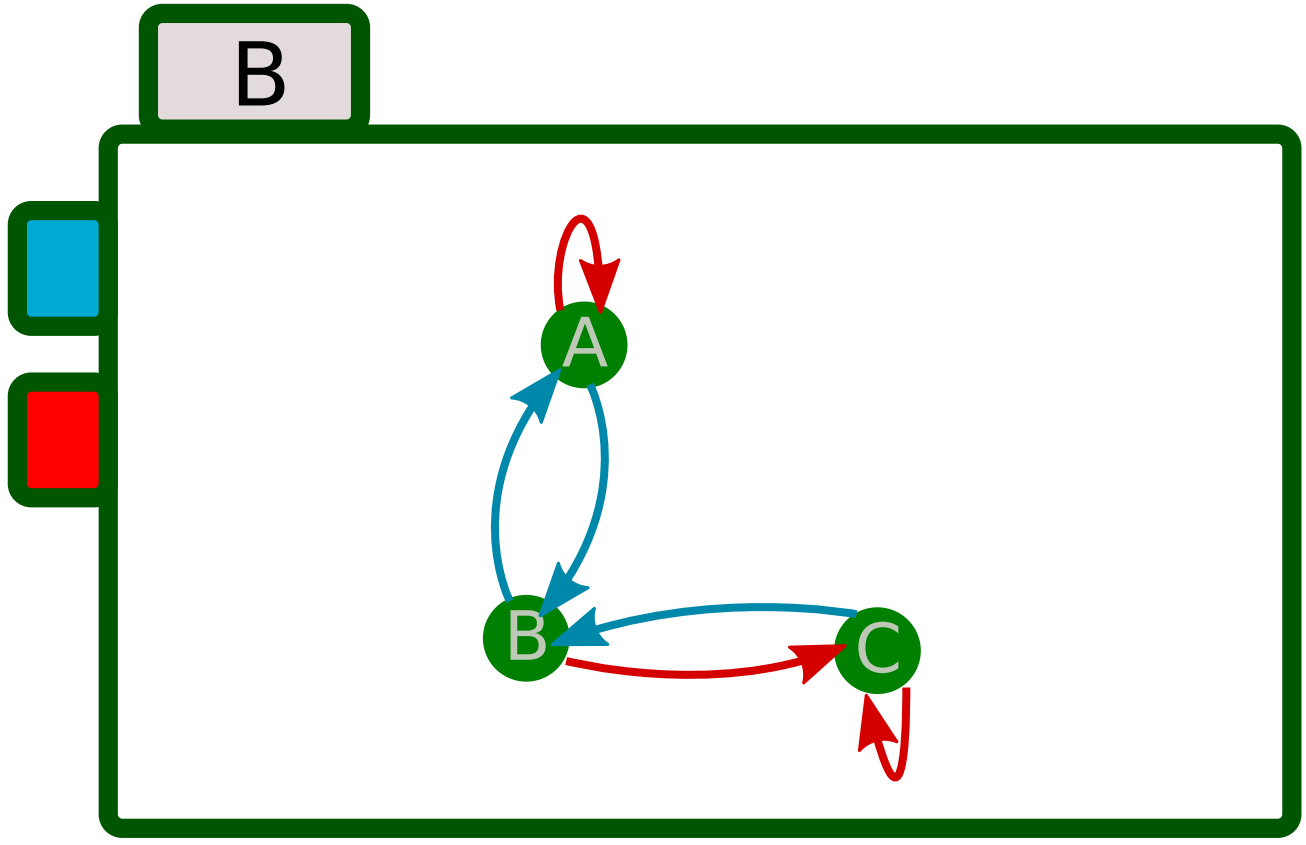
Stanislav Böhm, Jakub Beránek, Martin Šurkovský

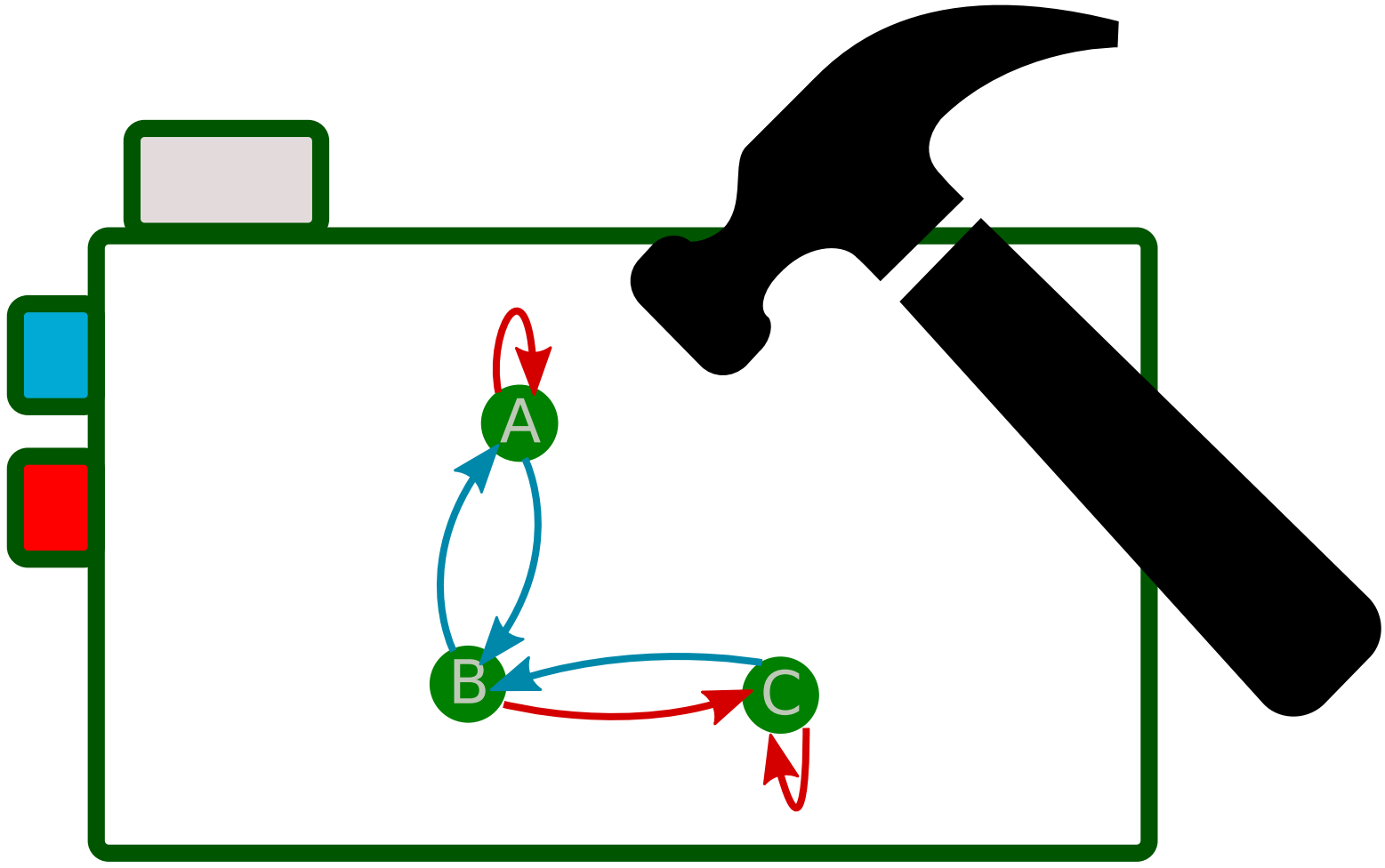
supported by GACR 15-13784S

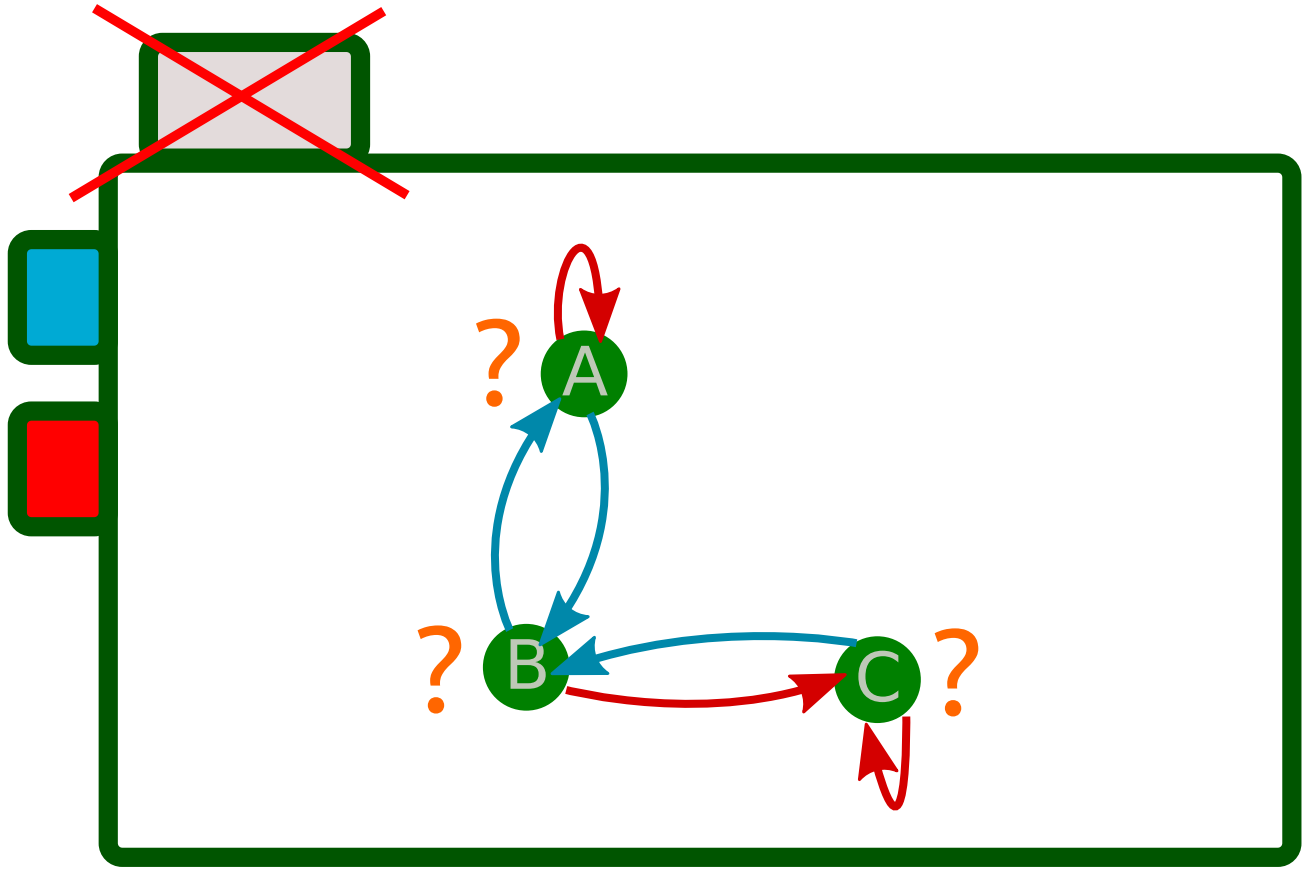

A

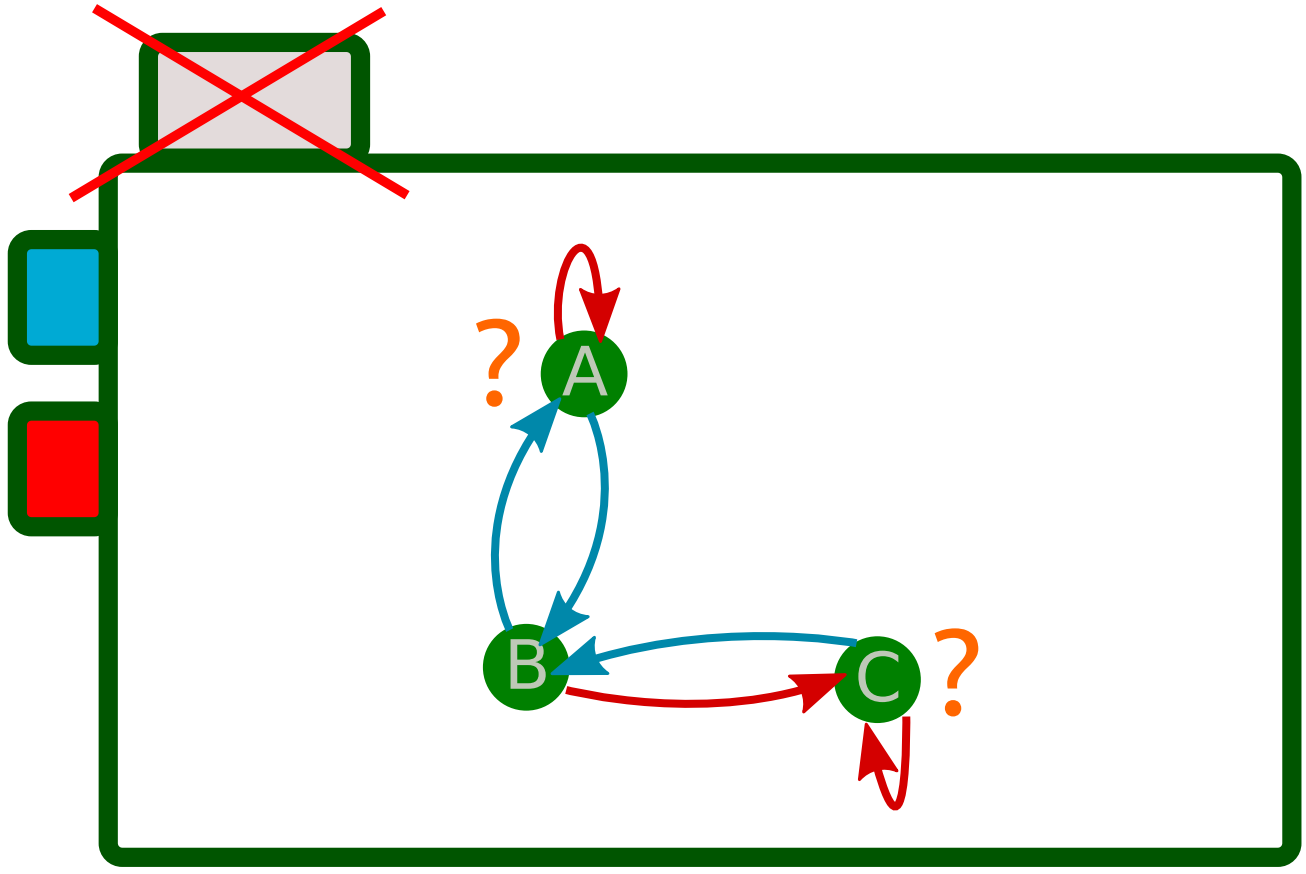

A
A
B
C

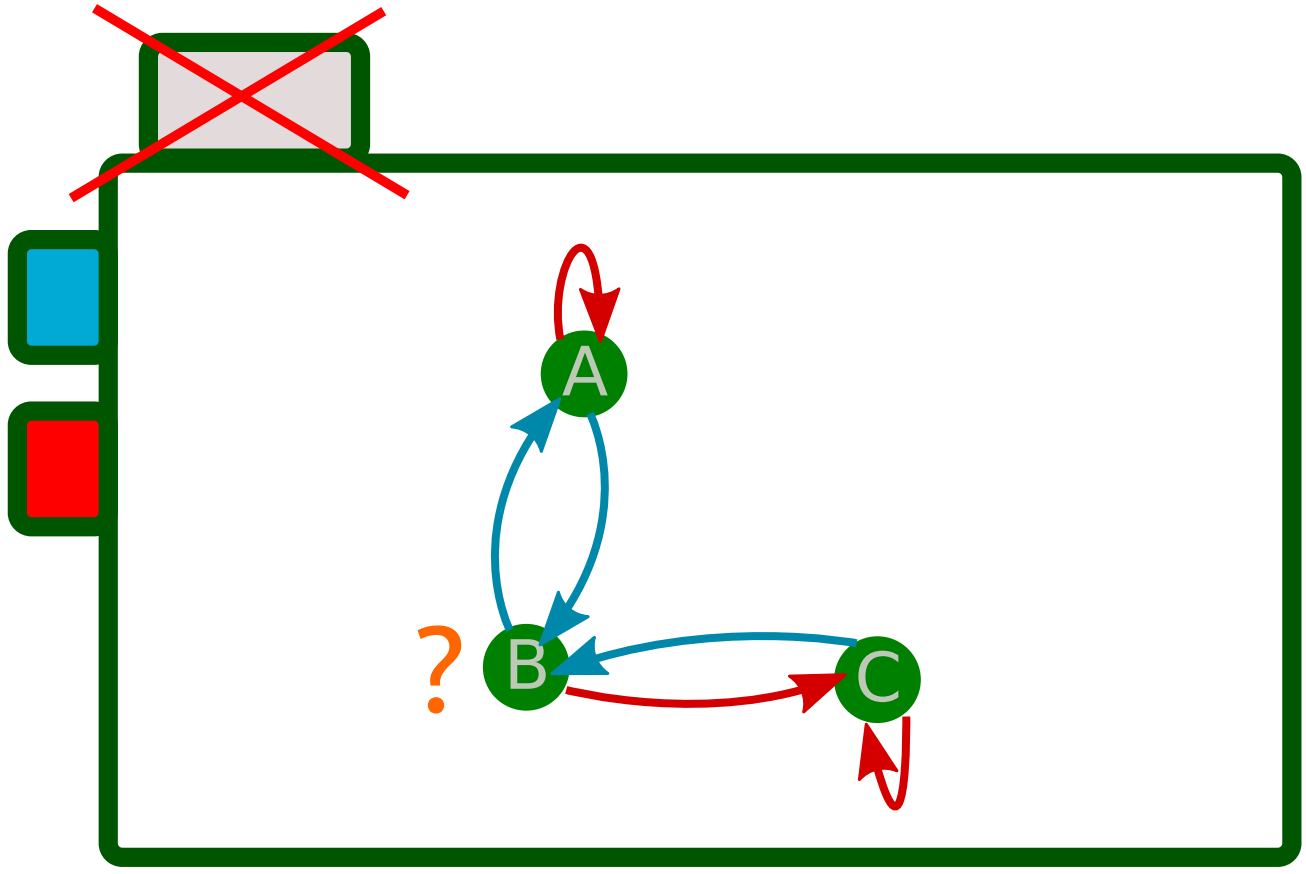

A
A
B
C

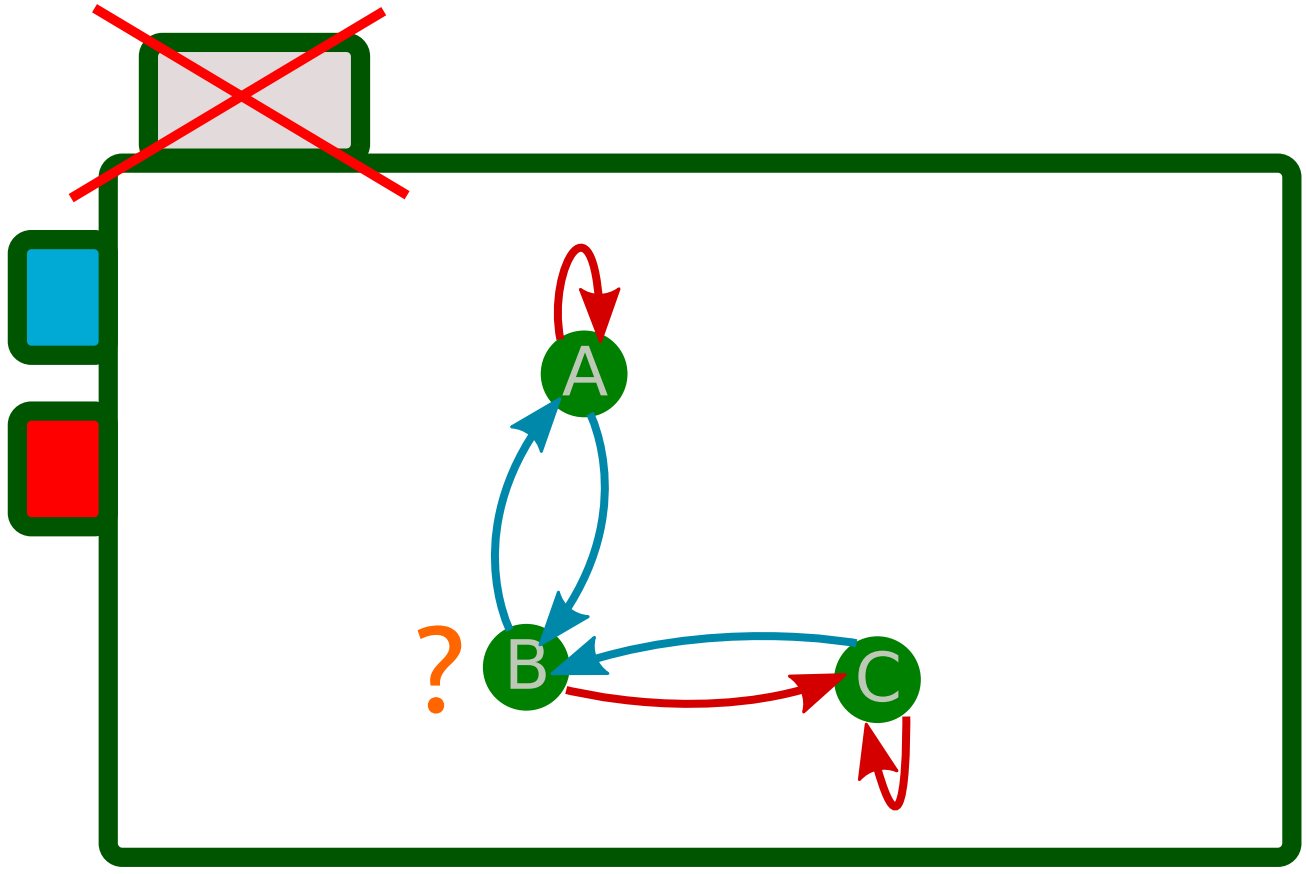

B
A
B
C

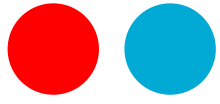

A
B
C


?
A
?
B
C
?

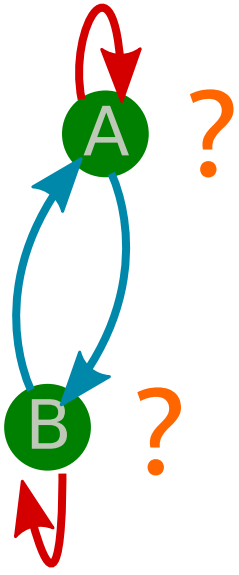

?
A
B
C
?


A
B
C
?



Reset word: 

Machine with no reset word



Let us have a machine M with n states.

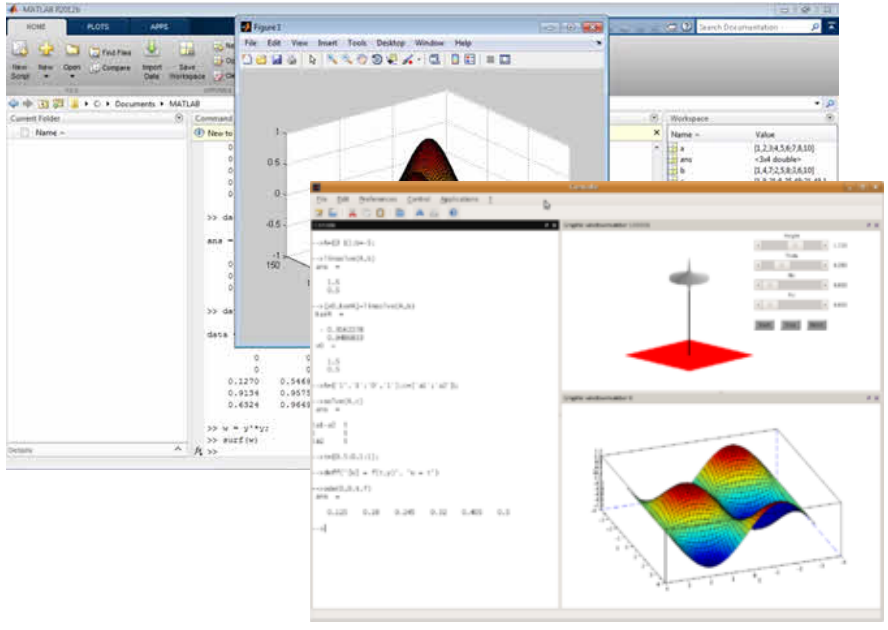
Question:
If a reset word for M exists,
can we bound the length of
the minimal reset word w.r.t. n?

Černý conjecture (1964)

$$\leq n^2 - 1$$

Number of states

Prototyping



Haystack Diver

https://haydi.readthedocs.io/

$$G = (\{v_1, \ldots, v_5\}, E)$$
$$E \subseteq V \times V$$

$$G = (\{v_1, \ldots, v_5\}, E)$$
$$E \subseteq V \times V$$

```
import haydi as hd

vertices = hd.USet(5, "v")
graphs = hd.Subsets(vertices * vertices)
```

Directed graphs

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

Iterate all elements:

```
>>> list(graphs.iterate())
```

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

Iterate all elements:

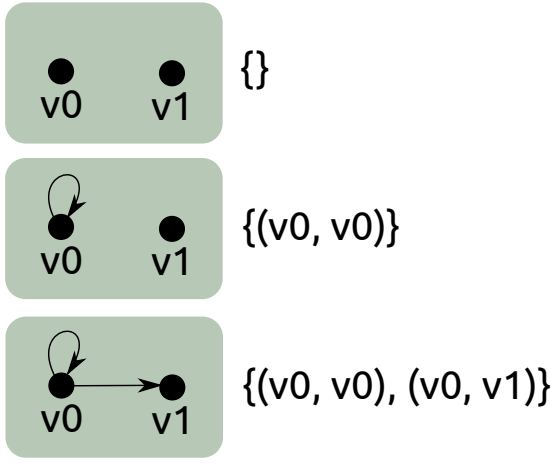
```
>>> list(graphs.iterate())
[{}, {(v0, v0)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0,
# ... 3 lines removed ...
v1)}, {(v1, v0)}, {(v1, v0), (v1, v1)}, {(v1, v1)}]
```

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

Iterate all elements:

```
>>> list(graphs.iterate())
[{}, {(v0, v0)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0,
# ... 3 lines removed ...
v1)}, {(v1, v0)}, {(v1, v0), (v1, v1)}, {(v1, v1)}]
```



...

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

Iterate all elements:

```
>>> list(graphs.iterate())
[{}, {(v0, v0)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0,
# ... 3 lines removed ...
v1)}, {(v1, v0)}, {(v1, v0), (v1, v1)}, {(v1, v1)}]
```

Random elements:

```
>>> list(graphs.generate(3))
[{(v1, v0)}, {(v1, v1), (v0, v0)}, {(v0, v1), (v1, v0)}]
```

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

Iterate all elements:

```
>>> list(graphs.iterate())
[{}, {(v0, v0)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0,
# ... 3 lines removed ...
v1)}, {(v1, v0)}, {(v1, v0), (v1, v1)}, {(v1, v1)}]
```

Random elements:

```
>>> list(graphs.generate(3))
[{(v1, v0)}, {(v1, v1), (v0, v0)}, {(v0, v1), (v1, v0)}]
```

Canonical forms:

```
>>> list(graphs.cnfs())  # cnfs = canonical forms
[{}, {(v0, v0)}, {(v0, v0), (v1, v1)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0,
v1), (v1, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0, v0), (v0, v1), (v1, v0),
(v1, v1)}, {(v0, v0), (v1, v0)}, {(v0, v1)}, {(v0, v1), (v1, v0)}]
```

```
import haydi as hd

vertices = hd.USet(2, "v")
graphs = hd.Subsets(vertices * vertices)
```

Iterate all elements:

```
>>> list(graphs.iterate())
[{}, {(v0, v0)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0,
# ... 3 lines removed ...
v1)}, {(v1, v0)}, {(v1, v0), (v1, v1)}, {(v1, v1)}]
```

Random elements:

```
>>> list(graphs.generate(3))
[{(v1, v0)}, {(v1, v1), (v0, v0)}, {(v0, v1), (v1, v0)}]
```

Canonical forms:

```
>>> list(graphs.cnfs())  # cnfs = canonical forms
[{}, {(v0, v0)}, {(v0, v0), (v1, v1)}, {(v0, v0), (v0, v1)}, {(v0, v0), (v0,
v1), (v1, v1)}, {(v0, v0), (v0, v1), (v1, v0)}, {(v0, v0), (v0, v1), (v1, v0),
(v1, v1)}, {(v0, v0), (v1, v0)}, {(v0, v1)}, {(v0, v1), (v1, v0)}]
```

Directed graphs

Directed graphs

```
graphs.iterate().map(do_something).max()
```

dask/distributed

https://distributed.readthedocs.io/en/latest/

```python
import haydi as hd
from haydi.algorithms import search

n_states = 4   # Number of states
n_symbols = 2  # Number of symbols in alphabet

states = hd.USet(n_states, "q")  # set of states q0, q1, ..., q_{n_states-1}
alphabet = hd.USet(n_symbols, "a")  # set of symbols a0, ..., a_{a_symbols-1}

# All Mappings (states * alphabet) -> states
delta = hd.Mappings(states * alphabet, states)

pipeline = delta.cnfs().map(check_automaton).max(size=1)
result = pipeline.run()


# Let us precompute some values that will be repeatedly used
init_state = frozenset(states)
max_steps = (n_states**3 - n_states) / 6
# Known result is that we do not need more than (n^3 - n) / 6 steps

def check_automaton(delta):
      def step(state, depth):
          for a in alphabet:
              yield frozenset(delta[(s, a)] for s in state)

      delta = delta.to_dict()
      return search.bfs(
          init_state,  # Initial state
          step,        # Function that takes a node and returns the followers
          lambda state, depth: depth if len(state) == 1 else None,
                       # Run until we reach a single state
          max_depth=max_steps,  # Limit depth of search
          not_found_value=0)    # Return 0 when we exceed depth limit
```

```
import haydi as hd
from haydi.algorithms import search

n_states = 4   # Number of states
n_symbols = 2  # Number of symbols in alphabet

states = hd.USet(n_states, "q")  # set of states q0, q1, ..., q_{n_states-1}
alphabet = hd.USet(n_symbols, "a")  # set of symbols a0, ..., a_{a_symbols-1}

# All Mappings (states * alphabet) -> states
delta = hd.Mappings(states * alphabet, states)

pipeline = delta.cnfs().map(check_automaton).max(size=1)
result = pipeline.run()


# Let us precompute some values that will be repeatedly used
init_state = frozenset(states)
max_steps = (n_states**3 - n_states) / 6
# Known result is that we do not need more than (n^3 - n) / 6 steps

def check_automaton(delta):
      def step(state, depth):
          for a in alphabet:
              yield frozenset(delta[(s, a)] for s in state)

      delta = delta.to_dict()
      return search.bfs(
          init_state,  # Initial state
          step,        # Function that takes a node and returns the followers
          lambda state, depth: depth if len(state) == 1 else None,
                       # Run until we reach a single state
          max_depth=max_steps,  # Limit depth of search
          not_found_value=0)    # Return 0 when we exceed depth limit
```

```
import haydi as hd
from haydi.algorithms import search

n_states = 4   # Number of states
n_symbols = 2  # Number of symbols in alphabet

states = hd.USet(n_states, "q")  # set of states q0, q1, ..., q_{n_states-1}
alphabet = hd.USet(n_symbols, "a")  # set of symbols a0, ..., a_{a_symbols-1}

# All Mappings (states * alphabet) -> states
delta = hd.Mappings(states * alphabet, states)

pipeline = delta.cnfs().map(check_automaton).max(size=1)
result = pipeline.run()


# Let us precompute some values that will be repeatedly used
init_state = frozenset(states)
max_steps = (n_states**3 - n_states) / 6
# Known result is that we do not need more than (n^3 - n) / 6 steps

def check_automaton(delta):
      def step(state, depth):
          for a in alphabet:
              yield frozenset(delta[(s, a)] for s in state)

      delta = delta.to_dict()
      return search.bfs(
          init_state,  # Initial state
          step,        # Function that takes a node and returns the followers
          lambda state, depth: depth if len(state) == 1 else None,
                       # Run until we reach a single state
          max_depth=max_steps,  # Limit depth of search
          not_found_value=0)    # Return 0 when we exceed depth limit
```

```python
import haydi as hd
from haydi.algorithms import search

n_states = 4   # Number of states
n_symbols = 2  # Number of symbols in alphabet

states = hd.USet(n_states, "q")  # set of states q0, q1, ..., q_{n_states-1}
alphabet = hd.USet(n_symbols, "a")  # set of symbols a0, ..., a_{a_symbols-1}

# All Mappings (states * alphabet) -> states
delta = hd.Mappings(states * alphabet, states)

pipeline = delta.cnfs().map(check_automaton).max(size=1)
result = pipeline.run()


# Let us precompute some values that will be repeatedly used
init_state = frozenset(states)
max_steps = (n_states**3 - n_states) / 6
# Known result is that we do not need more than (n^3 - n) / 6 steps

def check_automaton(delta):
      def step(state, depth):
          for a in alphabet:
              yield frozenset(delta[(s, a)] for s in state)

      delta = delta.to_dict()
      return search.bfs(
          init_state,  # Initial state
          step,        # Function that takes a node and returns the followers
          lambda state, depth: depth if len(state) == 1 else None,
                       # Run until we reach a single state
          max_depth=max_steps,  # Limit depth of search
          not_found_value=0)    # Return 0 when we exceed depth limit
```

```python
import haydi as hd
from haydi.algorithms import search

n_states = 4   # Number of states
n_symbols = 2  # Number of symbols in alphabet

states = hd.USet(n_states, "q")  # set of states q0, q1, ..., q_{n_states-1}
alphabet = hd.USet(n_symbols, "a")  # set of symbols a0, ..., a_{a_symbols-1}

# All Mappings (states * alphabet) -> states
delta = hd.Mappings(states * alphabet, states)

pipeline = delta.cnfs().map(check_automaton).max(size=1)
result = pipeline.run()


# Let us precompute some values that will be repeatedly used
init_state = frozenset(states)
max_steps = (n_states**3 - n_states) / 6
# Known result is that we do not need more than (n^3 - n) / 6 steps

def check_automaton(delta):
      def step(state, depth):
          for a in alphabet:
              yield frozenset(delta[(s, a)] for s in state)

      delta = delta.to_dict()
      return search.bfs(
          init_state,  # Initial state
          step,        # Function that takes a node and returns the followers
          lambda state, depth: depth if len(state) == 1 else None,
                       # Run until we reach a single state
          max_depth=max_steps,  # Limit depth of search
          not_found_value=0)    # Return 0 when we exceed depth limit
```

Haystack Diver

https://haydi.readthedocs.io/