Distributed LTL Model-Checking

Jiří Barnat, Luboš Brim, Ivana Černá

ParaDiSe
Parallel & Distributed Systems Laboratory

Faculty of Informatics
Masaryk University Brno

SENVA workshop, November 16, 2005, Grenoble

Enumerative LTL Model-Checking

Automata Approach – Basic Principle

- The LTL model-checking problem "$A \models \varphi$?" is reduced to

  is the language recognized by $A \times B_{\neg\varphi}$ empty ?

- BA C can be represented as a graph G_C
- $L(C)$ is non-empty iff G_C has a reachable accepting cycle

Graph problem:

Given: Digraph with a source vertex and subset of vertices marked as accepting.

Question: Does there exist a cycle which contains at least one accepting vertex and is reachable from the source ?

In positive case generate generate the cycle and a path to it from source.

Distributed LTL Model-Checking

Platform

- Network of workstations (NOWs).
- No shared memory (combined memory).
- Communication by message passing.

Graph distribution

- Graph given implicitly by $(F_{init}, F_{successor})$
- Distributed data – **partition function** assigns vertices to workstations

Graph problem: Detection of a reachable accepting cycle in a distributed graph.

Distributed Algorithms

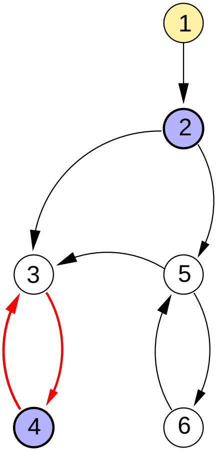
- new algorithms needed
  - sequential solution: postorder – difficult to parallelize (PTIME)
  - parallel solution: reachability – efficient parallelization (NC)
- **travel & propagate (repeated reachability)**

Four groups – Six algorithms

- BFS instead of DFS
  [Maximal Predecessors, Back-Level Edges]
- SCC-based approaches
  [Elimination of SCCs – forth and back]
- Reduction to another problem
  [Negative Cycles]
- Additional data
  [Dependency Structure]

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]

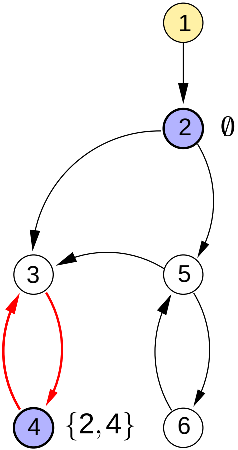


Idea

Each accepting vertex on an accepting cycle is its own predecessor.

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Idea

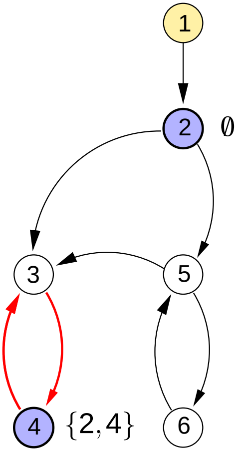
Each accepting vertex on an accepting cycle is its own predecessor.

Algorithm

forall $s \in A$ **do**
  $Acc(s)$ = set of accepting
    predecessors of s **od**
forall $s \in A$ **do**
  if $s \in Acc(s)$ **then return** CYCLE
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



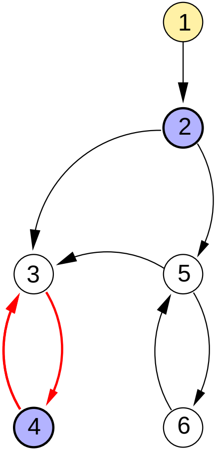
Idea

Each accepting vertex on an accepting cycle is its own predecessor.

- Storing all predecessors is expensive.
- Order accepting vertices and store **the maximal one** only.

Maximal Accepting Predecessors

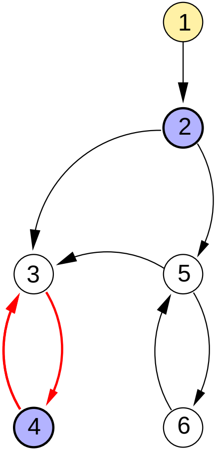
[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Improved idea

If an accepting vertex is the maximal accepting predecessor of itself, then it belongs to an accepting cycle.

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Improved idea

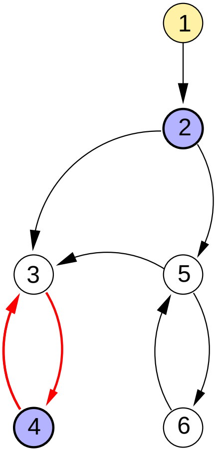
If an accepting vertex is the maximal accepting predecessor of itself, then it belongs to an accepting cycle.

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

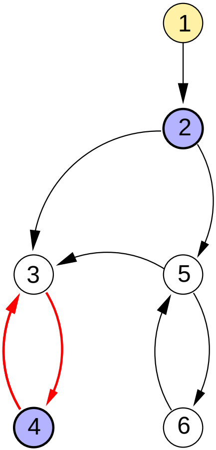
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

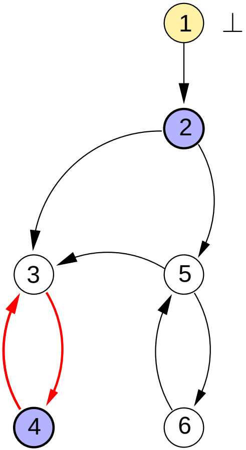
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

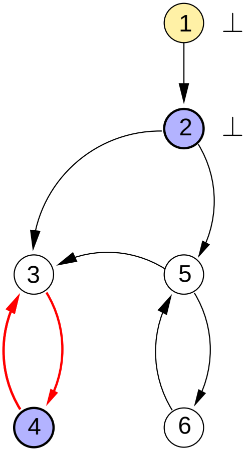
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

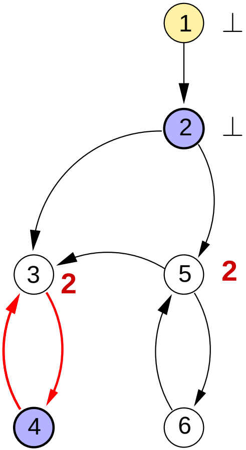
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

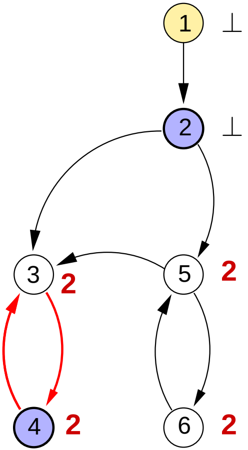
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

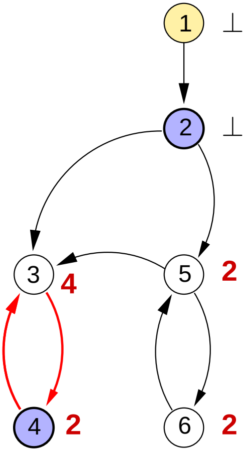
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

$4 > 2 > \bot$

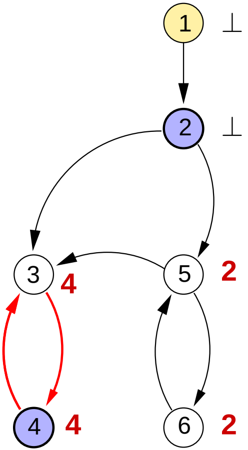
Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

ParaDiSe Parallel & Distributed Systems Laboratory

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

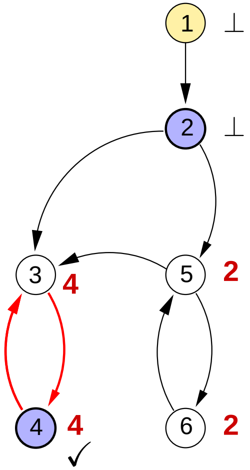
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

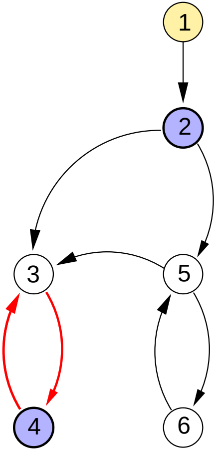
$4 > 2 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

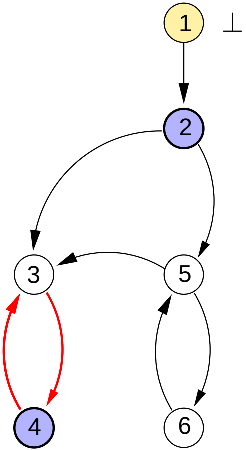
$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

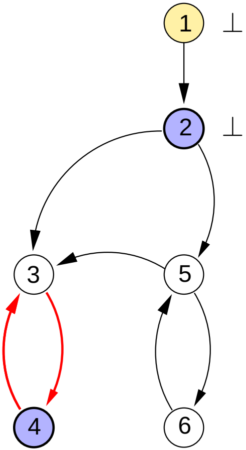
$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

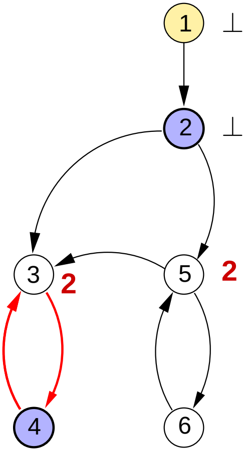
$2 > 4 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

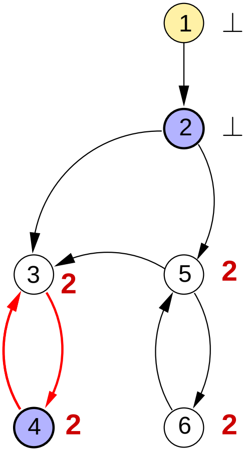
$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

$2 > 4 > \bot$

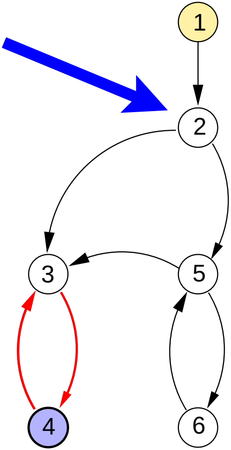
Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

ParaDiSe Parallel & Distributed Systems Laboratory

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

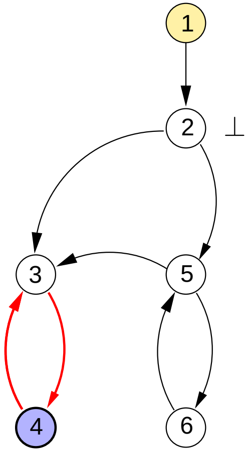
$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

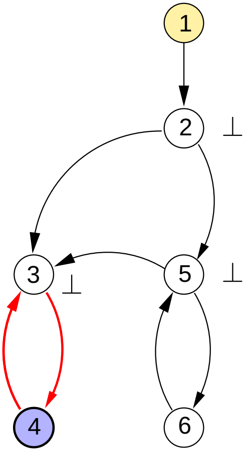
$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
 compute *map*; { max. accepting predecessors }
 if $(\exists u \in A : map(u) = u)$
 then return CYCLE
 else $G = delacc(G)$; { unmark acc. predecessors }
 fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

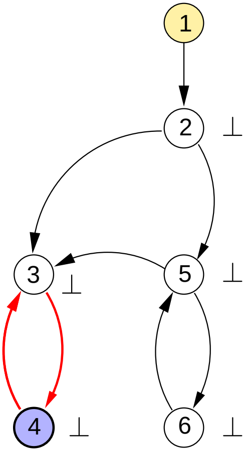
$2 > 4 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

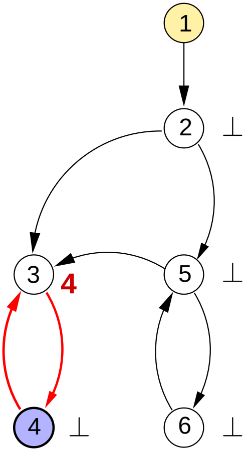
$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

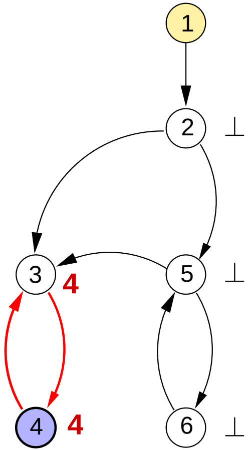
$2 > 4 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

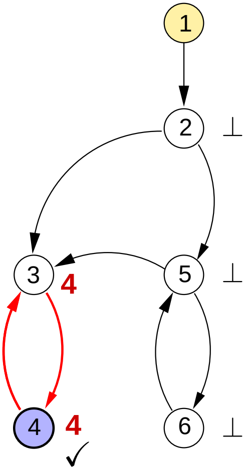
$2 > 4 > \bot$

Algorithm

while $A \neq \emptyset$ **do**
  compute *map*; { max. accepting predecessors }
  if $(\exists u \in A : map(u) = u)$
  then return CYCLE
  else $G = delacc(G)$; { unmark acc. predecessors }
  fi
od
return NO CYCLE

Maximal Accepting Predecessors

[Brim, Černá, Moravec, Šimša – FMCAD 2004, PDMC 2005]



Ordering

$2 > 4 > \perp$

Algorithm

while $A \neq \emptyset$ **do**
compute *map*; { max. accepting predecessors }
if $(\exists u \in A : map(u) = u)$
then return CYCLE
else $G = delacc(G)$; { unmark acc. predecessors }
fi
od
return NO CYCLE

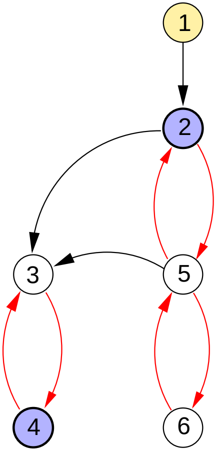
Maximal Accepting Predecessors

Comments

- An accepting cycle in G can be formed from vertices with the same maximal accepting predecessor only.
- A graph induced by the set of vertices having the same maximal accepting predecessor is called predecessor subgraph.
- Every cycle in the graph is completely included in one of the predecessor subgraphs.
- Re-computing the MAP function can be done in parallel for every predecessor subgraph.
- DFS gives optimal ordering – heuristics for “good” ordering.

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]

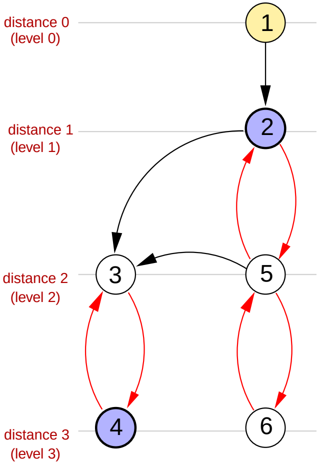


Back-Level Edge

Destination state has no greater distance from source vertex than its source state.

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]

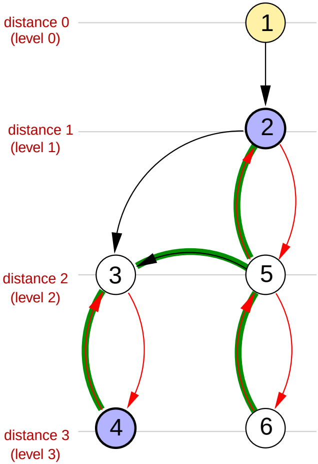


Back-Level Edge

Destination state has no greater distance from source vertex than its source state.

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]

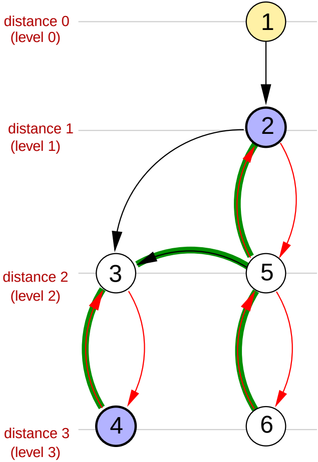


Back-Level Edge

Destination state has no greater distance from source vertex than its source state.

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]

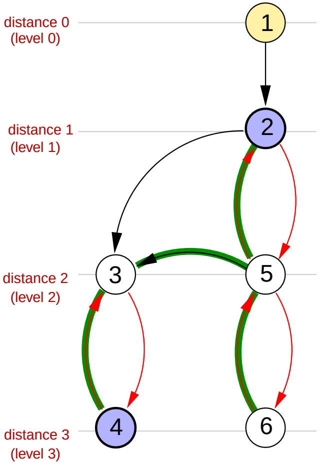


Idea

Each cycle must contain a back-level edge.

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



Idea

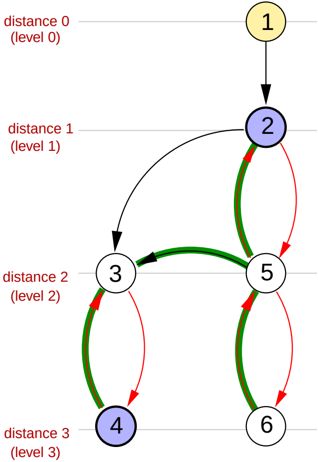
Each cycle must contain a back-level edge.

Algorithm

- Discover all back-level edges – level synchronized BFS.
- Check if there is an edge that is part of a cycle (nested procedure).

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



Idea

Each cycle must contain a back-level edge.

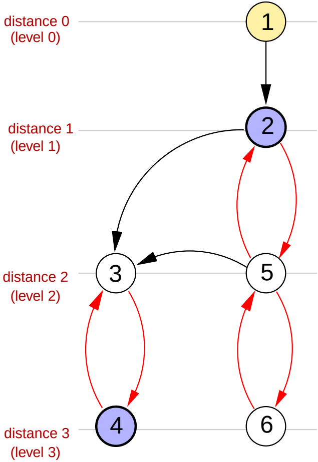
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$| L |$)
  od
od

proc test_cycle(s,t,$| L |$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $> | L |$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



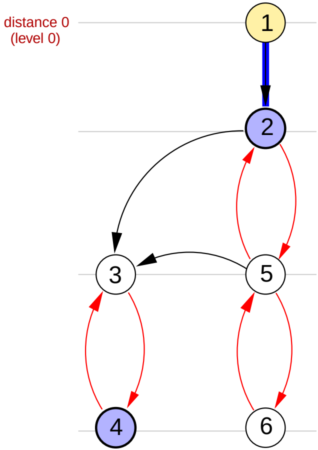
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



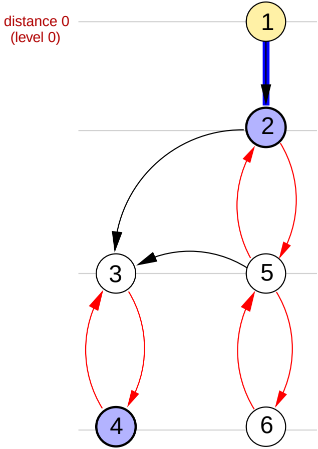
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|\ L\ |$)
  od
od

proc test_cycle(s,t,$|\ L\ |$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|\ L\ |$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



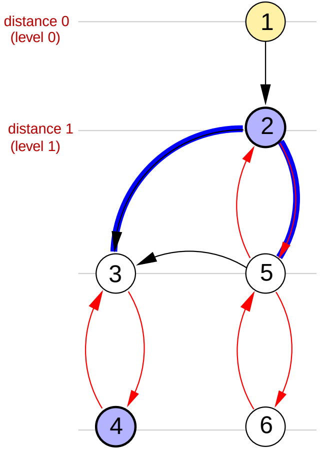
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s, t) \in L$ **do in parallel**
    test_cycle(s,t,$|$ L $|$)
  od
od

proc test_cycle(s,t,$|$ L $|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|$ L $|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



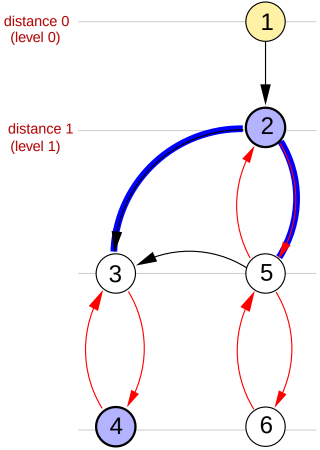
Algorithm

for level = 0 **to** . . . **do**
L = all current BL edges
forall $(s, t) \in L$ **do in parallel**
test_cycle(s,t,| L |)
od
od

proc test_cycle(s,t,| L |)
propagate s
if s propagated to itself **then** ✓
else if current BL passed $>| L |$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



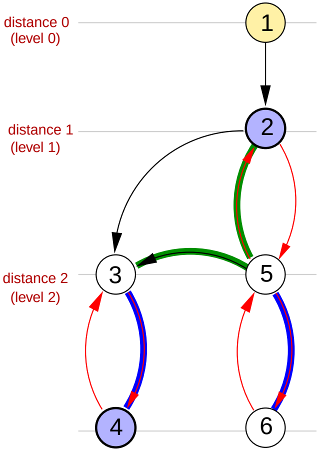
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s, t) \in L$ **do in parallel**
    test_cycle(s,t,| L |)
  od
od

proc test_cycle(s,t,| L |)
  propagate s
  if s propagated to itself **then** ✓
  else if current BL passed $>$| L | **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$L = \{(5,2),(5,3)\}$

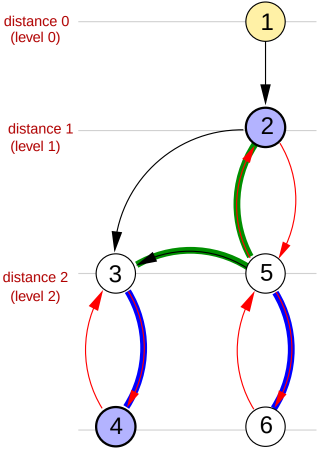
Algorithm

for level = 0 **to** ... **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$$L = \{(5,2),(5,3)\}$$

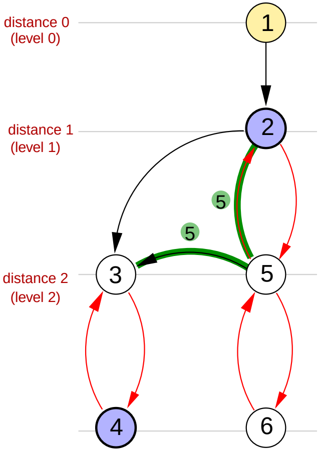
Algorithm

for level = 0 **to** ... **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate s
  if s propagated to itself **then** ✓
  else if current BL passed $> |L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$L = \{(5,2),(5,3)\}$

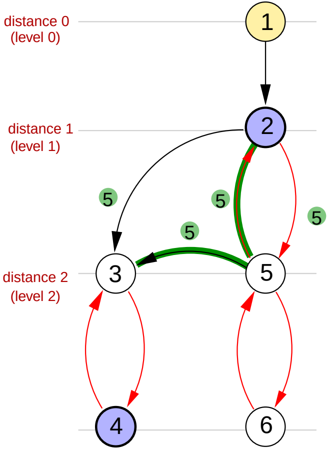
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $> |L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$L = \{(5,2),(5,3)\}$

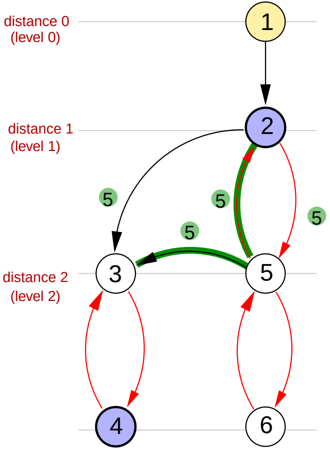
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $> |L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$$L = \{(5,2),(5,3)\}$$

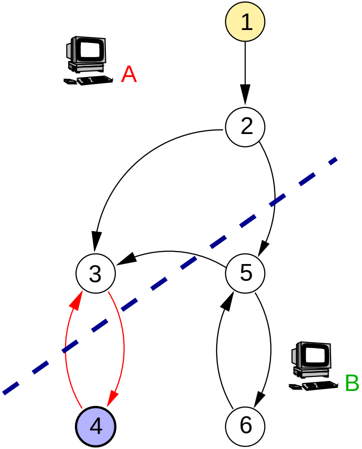
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|L|$ **then** ✓

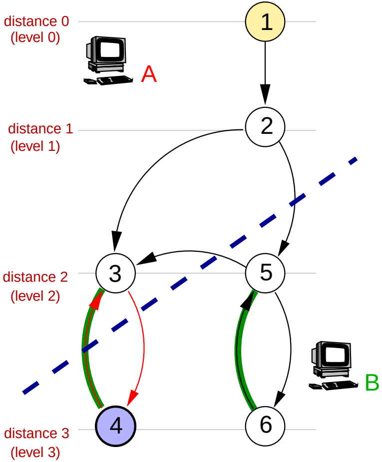
Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$L = \{(4,3),(6,5)\}$

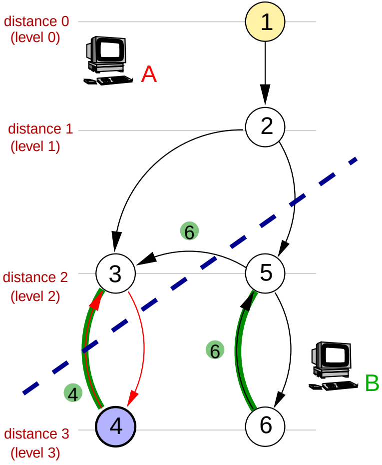
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $> |L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$$L = \{(4,3),(6,5)\}$$

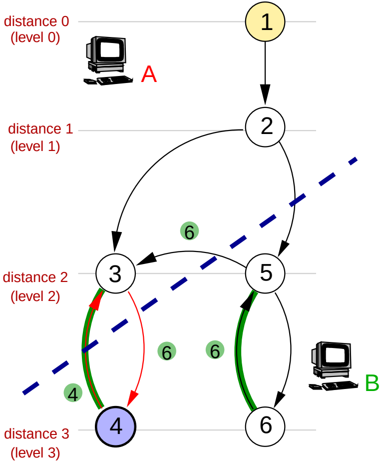
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $> |L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$$L = \{(4,3),(6,5)\}$$

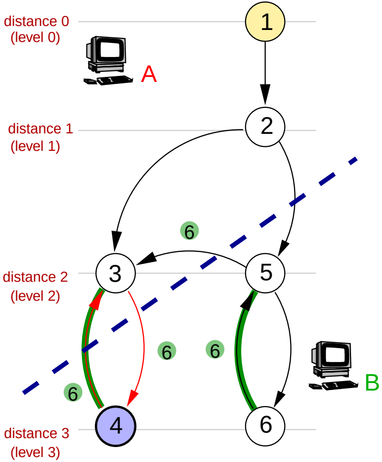
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



$$L = \{(4,3),(6,5)\}$$

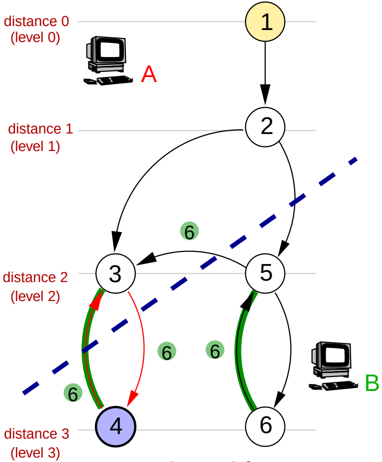
Algorithm

for level = 0 **to** . . . **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|L|$ **then** ✓

Back-Level Edges Algorithm

[Barnat, Brim, Chaloupka – ASE 2003]



stops after #bl edges

$$L = \{(4,3),(6,5)\}$$

Algorithm

for level = 0 **to** ... **do**
  L = all current BL edges
  forall $(s,t) \in L$ **do in parallel**
    test_cycle(s,t,$|L|$)
  od
od

proc test_cycle(s,t,$|L|$)
  propagate *s*
  if *s* propagated to itself **then** ✓
  else if current BL passed $>|L|$ **then** ✓

Back-Level Edges Algorithm

Comments

- Accepting cycle detection (additional bit required)
- Partial Order Reduction

Partial Order Reduction

- Exploring subsets of successors of states (ample sets)
- Conditions ensuring correctness: C0 – C3
- C3-DFS: at least one fully explored state on each cycle
- C3-BFS: **Full expansion of source states of back-level edges**

Back-Level Edges Algorithm

Comments

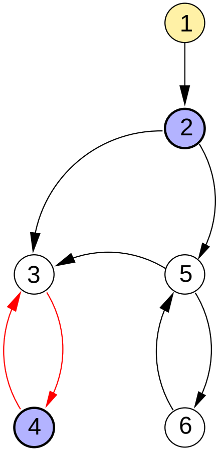
- Accepting cycle detection (additional bit required)
- Partial Order Reduction

Partial Order Reduction

- Exploring subsets of successors of states (ample sets)
- Conditions ensuring correctness: C0 – C3
- C3-DFS: at least one fully explored state on each cycle
- C3-BFS: **Full expansion of source states of back-level edges**

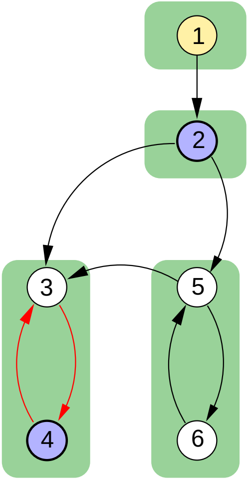
SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



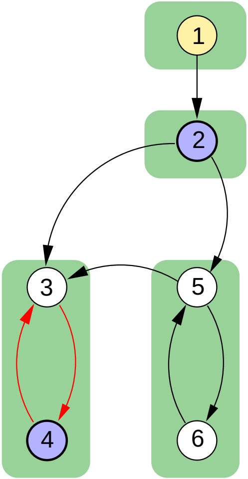
SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]

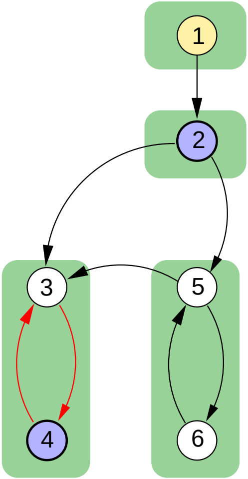


Idea

Each reachable accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



Idea

Each reachable accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

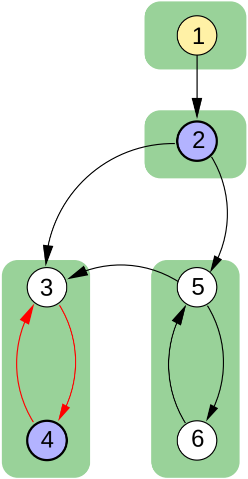
Algorithm

Remove all SCCs **without** required properties.

- remove trivial SCCs
- remove SCCs which are not reachable from the source
- remove SCCs which do not contain accepting vertices

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



Idea

Each reachable accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Algorithm

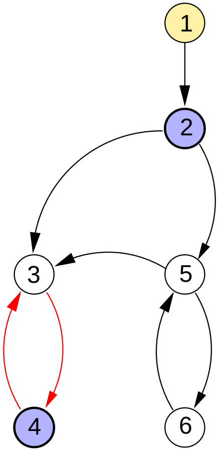
Remove all SCCs **without** required properties.

- remove trivial SCCs
- remove SCCs which are not reachable from the source
- remove SCCs which do not contain accepting vertices

ParaDiSe Parallel & Distributed Systems Laboratory

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



Idea

Each reachable accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

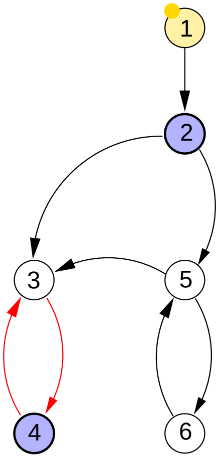
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



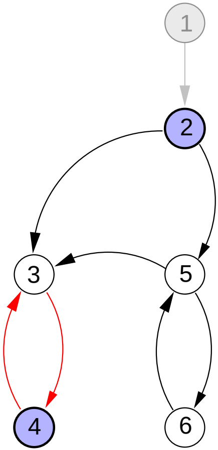
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



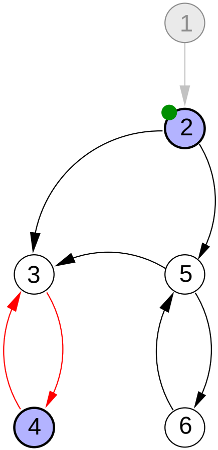
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



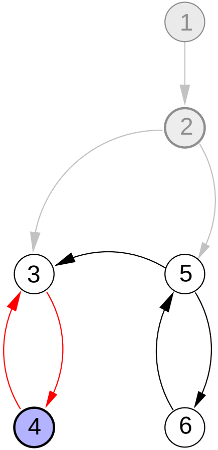
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



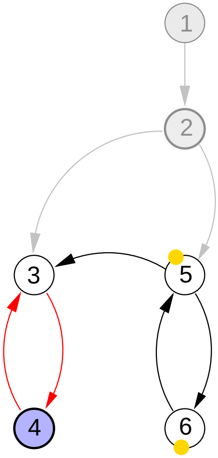
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



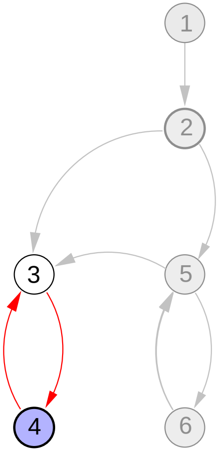
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



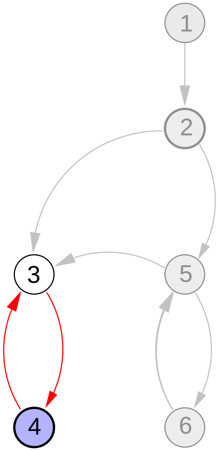
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm

[Černá, Pelánek – SPIN 2003]



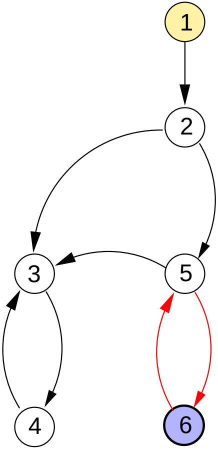
Algorithm on vertices

while not finished **do**

- remove vertices which are not reachable from accepting vertices
- remove vertices which are not contained in any cycle (have in-degree 0)

SCC-Based Algorithm – Reversed Version

[Barnat 2005]



Idea

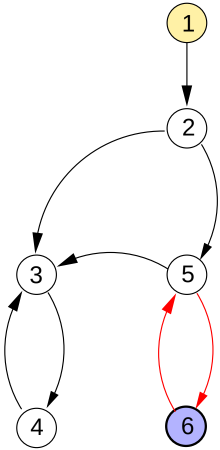
Main idea is the same: Each accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Computing successors may be expensive.

Store edges and check symmetric conditions using predecessors.

SCC-Based Algorithm – Reversed Version

[Barnat 2005]



Idea

Main idea is the same: Each accepting cycle is contained in a nontrivial strongly connected component which is reachable from the source vertex and contains an accepting vertex.

Computing successors may be expensive.

Store edges and check symmetric conditions using predecessors.

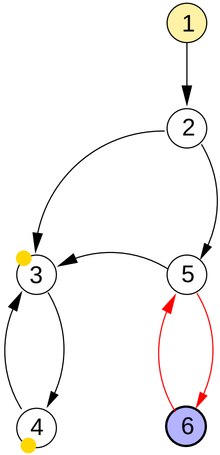
Algorithm on vertices

while not finished **do**

- remove vertices from which no accepting vertex is reachable
- remove vertices with out-degree 0

SCC-Based Algorithm – Reversed Version

[Barnat 2005]



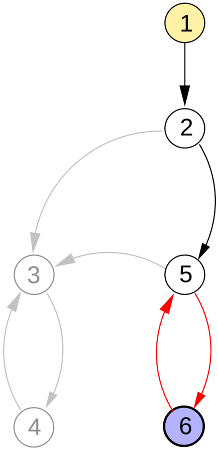
Algorithm on vertices

while not finished **do**

- remove vertices from which no accepting vertex is reachable
- remove vertices with out-degree 0

SCC-Based Algorithm – Reversed Version

[Barnat 2005]



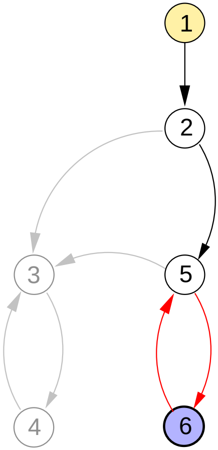
Algorithm on vertices

while not finished **do**

- remove vertices from which no accepting vertex is reachable
- remove vertices with out-degree 0

SCC-Based Algorithm – Reversed Version

[Barnat 2005]



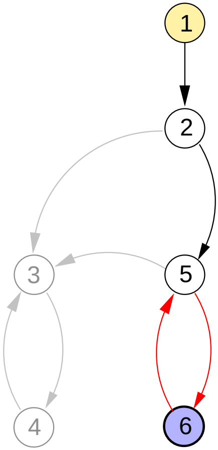
Algorithm on vertices

while not finished **do**

- remove vertices from which no accepting vertex is reachable
- remove vertices with out-degree 0

SCC-Based Algorithm – Reversed Version

[Barnat 2005]



Algorithm on vertices

while not finished **do**

- remove vertices from which no accepting vertex is reachable
- remove vertices with out-degree 0

SCC-Based Algorithms

Comments

- Time complexity is $O(h.(n+m))$
  - n - number of vertices
  - m - number of edges
  - h - height of SCC quotient graph
- Almost linear complexity
- Only one external iteration for weak BA graphs
- Algorithm does not work on-the-fly

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]

Idea

- Reduce BA emptiness problem to another one which can be distributed more easily.
- **Detecting negative cycles in the SSSP problem.**

Negative cycles coincide with accepting cycles.

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]

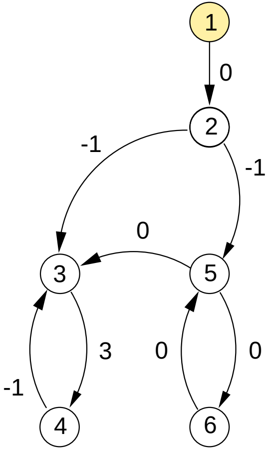
Idea

- Reduce BA emptiness problem to another one which can be distributed more easily.
- **Detecting negative cycles in the SSSP problem.**

Negative cycles coincide with accepting cycles.

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]

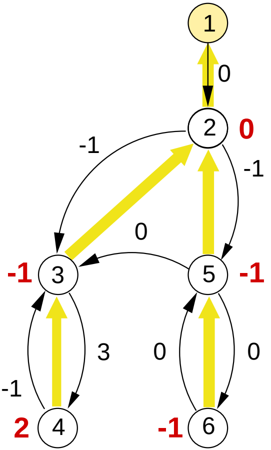


SSSP

For each vertex compute the smallest distance from source a build the parent graph (tree)

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



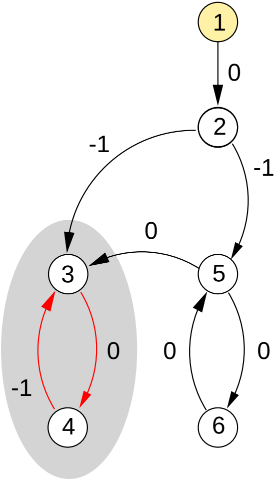
SSSP

For each vertex compute the smallest distance from source a build the parent graph (tree)

ParaDiSe Parallel & Distributed Systems Laboratory

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



SSSP

For each vertex compute the smallest distance from source a build the parent graph (tree)

Negative length cycles

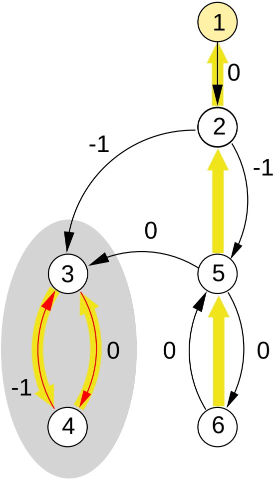
There is no shortest path to the source for vertices on negative cycles.

The parent graph has a cycle.

Detect negative cycles via cycles in the parent graph

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



SSSP

For each vertex compute the smallest distance from source a build the parent graph (tree)

Negative length cycles

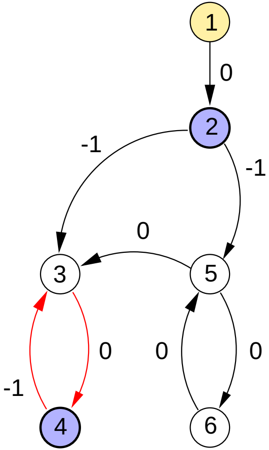
There is no shortest path to the source for vertices on negative cycles.

The parent graph has a cycle.

Detect negative cycles via cycles in the parent graph

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]

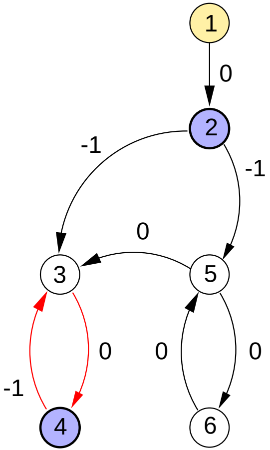


Idea

Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

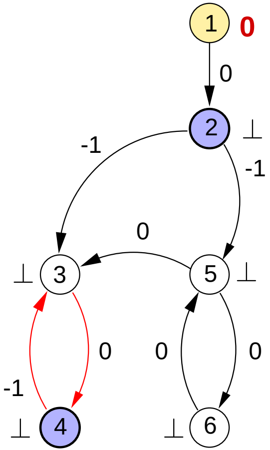
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

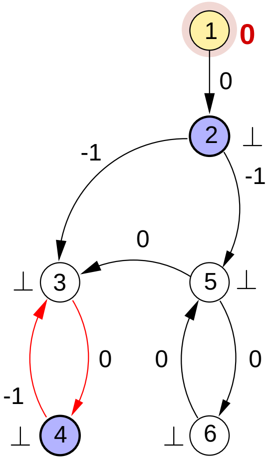
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

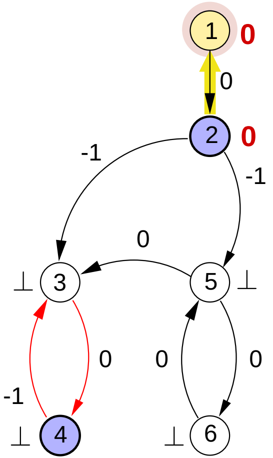
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

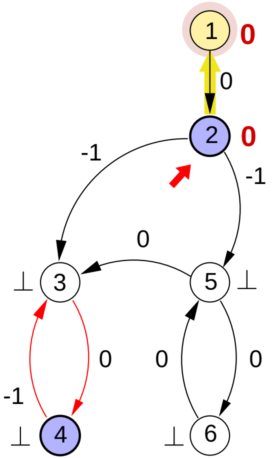
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

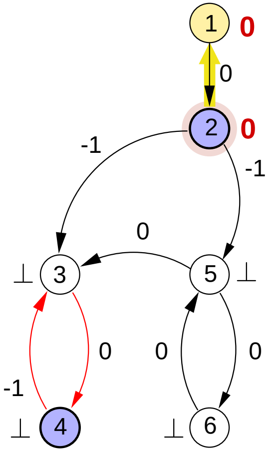
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

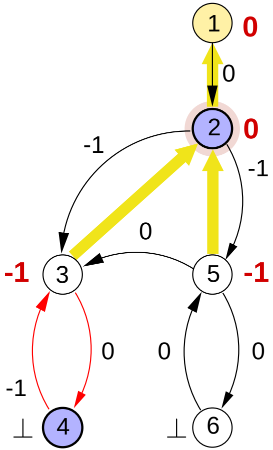
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

```
initialize
while not finished do
   scan vertices
   if successor vertex is accepting then
      run walk to root (WTR)
      if WTR reaches source
      then continue
      else CYCLE
      fi
   fi
od
```

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

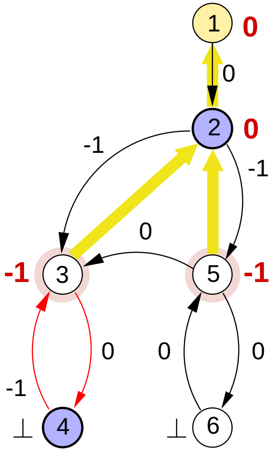
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

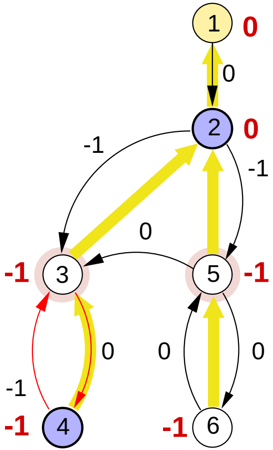
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

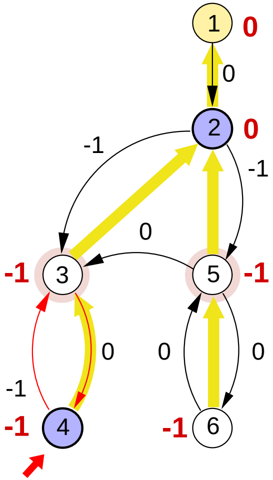
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

```
initialize
while not finished do
   scan vertices
   if successor vertex is accepting then
      run walk to root (WTR)
      if WTR reaches source
      then continue
      else CYCLE
      fi
   fi
od
```

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

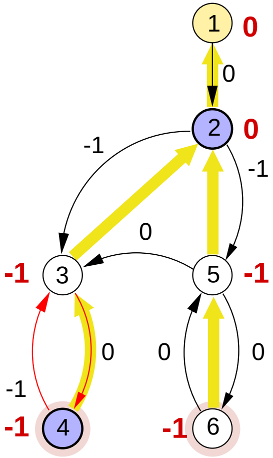
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

```
initialize
while not finished do
   scan vertices
   if successor vertex is accepting then
      run walk to root (WTR)
      if WTR reaches source
      then continue
      else CYCLE
      fi
   fi
od
```

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

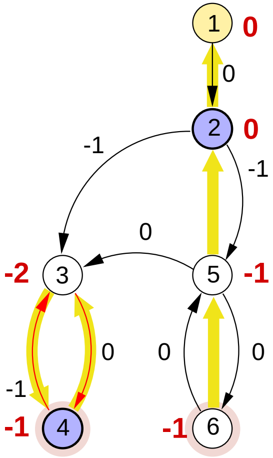
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



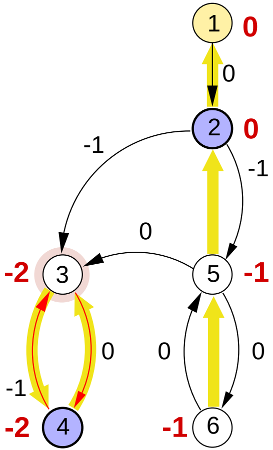
Idea

Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

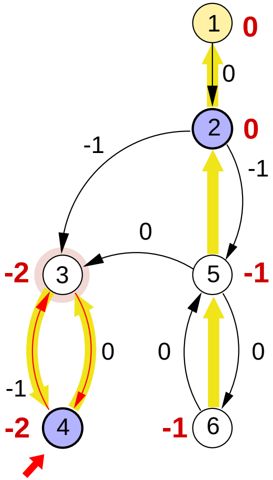
Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

[Brim, Černá, Krčál, Pelánek – FSTTCS 2001]



Idea

Reduction: Assign **-1** to out-going edges of accepting vertices, otherwise assign **0**.

Algorithm for detecting NC

initialize
while not finished **do**
  scan vertices
  if successor vertex is accepting **then**
    run walk to root (WTR)
    if WTR reaches source
    then continue
    else CYCLE
    fi
  fi
od

Negative Cycles Algorithm

Comments

- Strategies to detect presence of a negative cycle
  - time out
  - **walk to root (WTR)**
  - subtree traversal
  - **amortized search**

– time complexity is $\mathcal{O}(n^3/P)$
P - number of processors, n number of vertices

\+ algorithm is comparable with nested-DFS algorithm on all graphs

\+ algorithm is significantly better on graphs **without accepting cycles**

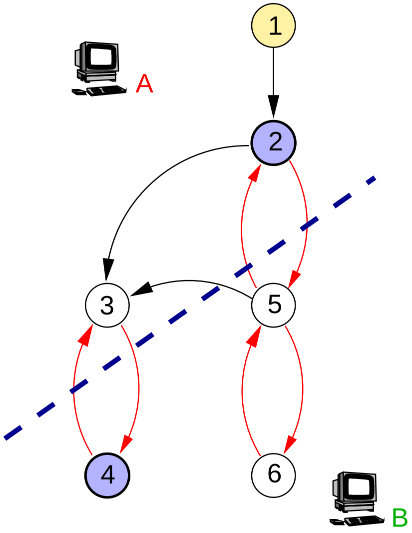
Dependency Structure Algorithm

[Barnat, Brim, Stříbrná - SPIN 2001]

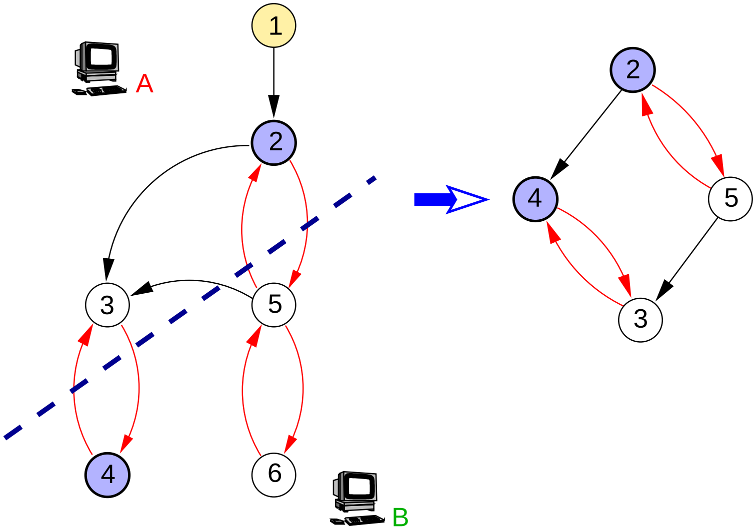
Idea

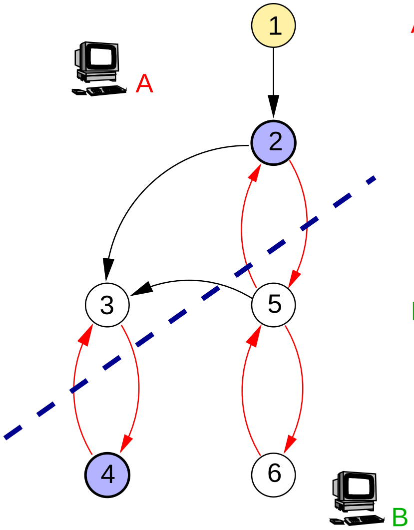
- Replace the graph by another smaller one:
  - Border vertices and accepting accepting only
  - Edges represent reachability (dependency) among these vertices.
- There is an accepting cycle in dependency graph iff there is a splitted accepting cycle in the original graph.
- Dependency graph is on-the-fly and is distributed as well.

Dependency Structure Algorithm – Example



Dependency Structure Algorithm – Example





A

2 → 4

2 → 5

3 → 4

B

4 → 3

5 → 3

5 → 2

Dependency Structure Algorithm

Comments

- Dependency structure:
  - Each workstation maintains its own local dependency structure.
  - Dynamic – vertices are added and removed.
- Additional memory required:
  ($O(n.r)$ on average, where r is the maximal out-degree and n is the number of states)
- Any distributed cycle detection algorithm can be used.

Conclusion

- Core problem of automata based LTL model-checking is the detection of reachable accepting cycles in the state space.
- Alternative approaches to distributing LTL Model-Checking presented.
- All algortihms implemented in DiVinE.
- Parallelization is used because parallel systems are complex and their development is difficult – development of parallel algorithms for their analyzis is mentally and technically challenging as well.
- Work in progress:
    - Extension to GBA, RA, SA.
    - LTL MC of probabilistic and real-time systems.
    - Cost analysis.