Three SCC-based Emptiness Checks for Generalized Büchi Automata

LPAR’19


Thursday, December 19th
Transition-based Generalized Büchi Automata

Runs are accepting iff they visit each acceptance set infinitely often.

An emptiness check looks for accepting runs.
Runs are accepting iff they visit each acceptance set infinitely often.
Transition-based Generalized Büchi Automata

Runs are accepting iff they visit each acceptance set infinitely often.

An emptiness check looks for accepting runs.
Existing explicit emptiness checks

- **NDFS-based**: look for accepting runs of the automaton using a second interleaved DFS,

- **SCC-based**: compute SCCs of the automaton and maintains acceptance sets for each SCCs using one DFS.

<table>
<thead>
<tr>
<th></th>
<th>NDFS-based</th>
<th>SCC-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-the-Fly</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bit state hashing</td>
<td>all states but DFS</td>
<td>only dead SCCs</td>
</tr>
<tr>
<td>State space caching</td>
<td>all states but DFS</td>
<td>only dead SCCs</td>
</tr>
<tr>
<td>Max memory req. for BA</td>
<td>2 bits per state</td>
<td>1 int per state</td>
</tr>
<tr>
<td>Generalization</td>
<td>difficult</td>
<td>trivial</td>
</tr>
<tr>
<td>Earlier CE detection</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>
This Talk!

Is there a best explicit SCC computation algorithm?

How to transform SCCs computation algorithms into generalized emptiness checks?

What is the cost of adding the emptiness check to an SCC computation algorithm?
Terminology

\[ s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_8 \rightarrow s_9 \rightarrow s_{10} \rightarrow s_{12} \rightarrow s_{13} \]

\[ s_4 \leftarrow s_3 \rightarrow s_6 \rightarrow s_7 \rightarrow s_{11} \]

- Current state: DFS stack
- LIVE state: LIVE stack
- DEAD state: Partial root backedge
- Compressed root stack
Terminology

Current state
Terminology

- **DFS stack**
- **Current state**
Terminology

DFS stack

Current state

LIVE state
Terminology

- **Current state**
- **DFS stack**
- **DEAD state**
- **LIVE state**
Terminology

- **DFS stack**
- **Current state**
- **LIVE state**
- **DEAD state**
- **LIVE number**

E. Renault

LPAR - 2013

Thursday, December 19th 5 / 14
Terminology

DFS stack
Current state
LIVE state

LIVE stack

DFS stack
Current state
LIVE state

LIVE stack

<table>
<thead>
<tr>
<th>s₁</th>
<th>s₂</th>
<th>s₃</th>
<th>s₄</th>
<th>s₅</th>
<th>s₆</th>
<th>s₇</th>
<th>s₈</th>
<th>s₉</th>
<th>s₁₀</th>
<th>s₁₁</th>
<th>s₁₂</th>
<th>s₁₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>
Terminology

DFS stack

Current state

DEAD state

LIVE state

LIVE stack

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
<th>s6</th>
<th>s7</th>
<th>s8</th>
<th>s9</th>
<th>s10</th>
<th>s11</th>
<th>s12</th>
<th>s13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LIVE number

Partial root
Terminology

DFS stack

Current state

LIVE state

DFS stack

DEAD state

LIVE number

partial root

Backedge

LIVE stack

\[
\begin{array}{cccccccccccc}
  s_1 & s_2 & s_3 & s_4 & s_5 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} \\
  1   & 2   & 3   & 4   & 5   & 6   & 7   & 8   & 9   & 10  & 11  \\
\end{array}
\]
Tarjan [1972]

Associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink:

LIVE stack size() + 1;

For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004]

Büchi Automaton; one lowlink per LIVE state; an extra stack for DFS position of accepting states.

LPAR'19

Generalized Büchi Automaton; one lowlink per state on the DFS stack; a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973]

Associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Couvreur [1999]

Generalized Büchi Automaton; rediscovers Dijkstra [1973] starting from Tarjan [1972]; a hybrid algorithm between SCC-based and NDFS-based; an acceptance set per element in the root stack.

Couvreur et al. [2005]

Restores the SCC-based aspect of the algorithm by storing states in the same SCC; two new heuristics using characteristics of Dijkstra's algorithm; counterexamples extraction.

Geldenhuys and Valmari [2005]

Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; more efficient data structure; counterexamples extraction.

Cheriyan and Mehlhorn [1996]

Optimisation for dense explicit graph; theoretical complexity analysis; Gabow [2000]

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; Alur et al. [2005]

Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton; Hansen and Geldenhuys [2008]

Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; extraction of small counterexamples.

Gaiser and Schwoon [2009]

Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton.

LPAR'19

Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; compressed root stack; rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; suggests the use of a Union-Find to perform the SCC computation.

Nuutila and Soisalon-Soininen [1994]

Pearce [2005]
Tarjan [1972]

- Associates an identifier (*lowlink*) to each state on the DFS stack;
- These *lowlinks* are stored in a *lowlink stack*
- Every new state pushed on the DFS stack has for *lowlink* : LIVE stack size() + 1;
- For every backtrack, the *lowlink* at the top of the *lowlink stack* will be affected to a smaller or equal value;
- If a state that has a *lowlink* equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Dijkstra [1973]

- Associates an identifier (*DFS Position*) to each state on the DFS stack;
- These *DFS Positions* are stored in a *root stack*
- When a backedge is found, the *root stack* is updated until the top of this stack is lesser or equal to the *DFS Position* of the destination;
- If a state that has a *DFS position* equal to the top of the *root stack* it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.
Tarjan [1972] Associates an identifier ('lowlink') to each state on the DFS stack; These lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink:

\[ \text{LIVE stack size() + 1}; \]

For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value; If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] Büchi Automaton; One lowlink per LIVE state; An extra stack for DFS position of accepting states;

Dijkstra [1973] Associates an identifier ('DFS Position') to each state on the DFS stack; These DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination; If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.


Couvreur et al. [2005] Restores the SCC-based aspect of the algorithm by storing states in the same SCC; Two new heuristics using characteristic of Dijkstra's algorithm; Counterexamples extraction;

Geldenhuys and Valmari [2005] Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; More efficient data structure; Counterexamples extraction;


Gaiser and Schwoon [2009] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; LPAR'19 Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; Compressed root stack; Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; Suggests the use of a Union-Find to perform the SCC computation; LPAR'19 Use a Union-Find data strucure to avoid the cost of marking dead an SCC; Compatible root stack compression; Nuutila and Soisalon-Soininen [1994] E. Renault LPAR - 2013 Thursday, December 19th 6 / 14

- Büchi Automaton;
- One lowlink per LIVE state;
- An extra stack for DFS position of accepting states;

Dijkstra [1973] →

- Associates an identifier (lowlink) to each state on the DFS stack;
- These lowlinks are stored in a lowlink stack;
- Every new state pushed on the DFS stack has for lowlink: LIVE stack size() + 1;
- For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value;
- If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Couvreur [1999]

- Generalized Büchi Automaton;
- Rediscovers Dijkstra [1973] starting from Tarjan [1972];
- Hybrid algorithm between SCC-based and NDFS-based;
- An acceptance set per element in the lowlink stack;

Couvreur et al. [2005]

- Restores the SCC-based aspect of the algorithm by storing states in the same SCC;
- Two new heuristics using characteristic of Dijkstra's algorithm;
- Counterexamples extraction;

Geldenhuys and Valmari [2004] → Couvreur [1999];

- More efficient data structure;
- Counterexamples extraction;

Cheriyan and Mehlhorn [1996]

- Optimisation for dense explicit graph;
- Theoretical complexity analysis;

Gabow [2000]

- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];

Alur et al. [2005]

- Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008]

- Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;
- Extraction of small counterexamples;

Gaiser and Schwoon [2009]

- Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

LPAR'19

- Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed root stack;

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
- Suggests the use of a Union-Find to perform the SCC computation;

LPAR'19

- Use a Union-Find data structure to avoid the cost of marking dead an SCC;
- Compatible root stack compression;

Nuutila and Soisalon-Soininen [1994]

E. Renault

Thursday, December 19th 6 / 14

- Büchi Automaton;
- One lowlink per LIVE state;
- An extra stack for DFS position of accepting states;

Dijkstra [1973]
Tarjan [1972] 

Geldenhuys and Valmari [2004] 

LPAR’19

- Generalized Büchi Automaton;
- One \textit{lowlink} per state on the DFS stack;
- A set of acceptance sets per element in the \textit{lowlink stack};

- Büchi Automaton;
- One \textit{lowlink} per LIVE state;
- An extra stack for DFS position of accepting states;

Dijkstra [1973]
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has a lowlink of LIVE stack size + 1. For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number, it's a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.

Geldenhuys and Valmari [2004] introduce a Büchi Automaton; one lowlink per LIVE state; an extra stack for DFS position of accepting states.

Generalized Büchi Automaton; one lowlink per state on the DFS stack; a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.


Couvreur et al. [2005] restores the SCC-based aspect of the algorithm by storing states in the same SCC; two new heuristics using characteristic of Dijkstra's algorithm; counterexamples extraction.

Geldenhuys and Valmari [2005] combine Geldenhuys and Valmari [2004] and Couvreur [1999]; a more efficient data structure; counterexamples extraction.

Cheriyan and Mehlhorn [1996] optimize for dense explicit graph; theoretical complexity analysis.


Gaiser and Schwoon [2009] propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; LPAR’19 mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; compressed root stack; rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; suggests the use of a Union-Find to perform the SCC computation; LPAR’19 uses a Union-Find data structure to avoid the cost of marking dead an SCC; compatible root stack compression.

Nuutila and Soisalon-Soininen [1994] and Pearce [2005] contribute to the emptiness check of Büchi Automaton; extraction of small counterexamples; LPAR’19 mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; compressed root stack; suggests the use of a Union-Find to perform the SCC computation.
Associates an identifier (\textit{DFS Position}) to each state on the DFS stack;

These \textit{DFS Position} are stored in a \textit{root stack}

When a backedge is found, the \textit{root stack} is updated until the top of this stack is lesser or equal to the \textit{DFS Position} of the destination;

If a state that has a \textit{DFS position} equal to the top of the \textit{root stack} it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.
Tarjan [1972] 

Geldenhuys and Valmari [2004] 

LPAR’19 

Dijkstra [1973] 

Couvreur [1999] 

Couvreur et al. [2005] 

Couvreur [1999] 

Hybrid algorithm between SCC-based and NDFS-based; 

An acceptance set per element in the root stack; 

Couvreur et al. [2005] 

Restores the SCC-based aspect of the algorithm by storing states in the same SCC; 

Two new heuristics using characteristic of Dijkstra’s algorithm; 

Counterexamples extraction; 

Geldenhuys and Valmari [2005] 

Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; 

More efficient data structure; 

Counterexamples extraction; 

Gaiser and Schwoon [2009] 

Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; 

Nuutila and Soisalon-Soininen [1994] 

Pearce [2005] 

E. Renault LPAR - 2013 Thursday, December 19th 7 / 14

- Generalized Büchi Automaton;
- RedisCOVERs Dijkstra [1973] starting from Tarjan [1972];
- Hybrid algorithm between SCC-based and NDFS-based;
- An acceptance set per element in the root stack;

- Use a Union-Find data structure to avoid the cost of marking an SCC;
- Compatible root stack compression;
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has a lowlink:

\[ \text{LIVE stack size} + 1; \]

For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.

Geldenhuys and Valmari [2004] associate one lowlink per LIVE state; an extra stack for DFS position of accepting states.

Generalized Büchi Automaton; one lowlink per state on the DFS stack; a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.


Couvreur et al. [2005] restores the SCC-based aspect of the algorithm by storing states in the same SCC; two new heuristics using characteristics of Dijkstra's algorithm; counterexamples extraction.

Geldenhuys and Valmari [2005] combines Geldenhuys and Valmari [2004] and Couvreur [1999]; more efficient data structure; counterexamples extraction.

Cheriyan and Mehlhorn [1996] optimizes for dense explicit graph; theoretical complexity analysis.


Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; extraction of small counterexamples.

Gaiser and Schwoon [2009] propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; LPAR’19 mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; compressed root stack.

Cheriyan and Mehlhorn [1996] optimized for dense explicit graph; theoretical complexity analysis.

Gabow [2000] rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; suggests the use of a Union-Find structure to perform the SCC computation; LPAR’19 uses a Union-Find data structure to avoid the cost of marking an SCC; compatible root stack compression.
Restores the SCC-based aspect of the algorithm by storing states in the same SCC;

Two new heuristics using characteristic of Dijkstra’s algorithm;

Counterexamples extraction;
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink: \( \text{LIVE stack size} + 1 \). For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] uses a Büchi Automaton; one lowlink per LIVE state; an extra stack for DFS position of accepting states.

Dijkstra [1973] associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.


Couvreur et al. [2005] restores the SCC-based aspect of the algorithm by storing states in the same SCC; two new heuristics using characteristic of Dijkstra's algorithm; counterexamples extraction.

Geldenhuys and Valmari [2005] combines Geldenhuys and Valmari [2004] and Couvreur [1999]; more efficient data structure; counterexamples extraction.

Cheriyan and Mehlhorn [1996] optimize for dense explicit graphs; theoretical complexity analysis; Gabow [2000] reDISCOVERS Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; Alur et al. [2005] propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton; Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; extraction of small counterexamples; Gaiser and Schwoon [2009] propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; LPAR'19 mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; compressed root stack; reDISCOVERS Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; suggests the use of a Union-Find to perform the SCC computation; LPAR'19 uses a Union-Find data structure to avoid the cost of marking dead an SCC; compatible root stack compression; Nuutila and Soisalon-Soininen [1994].
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink: LIVE stack size() + 1. For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] introduce a Büchi Automaton; one lowlink per LIVE state; an extra stack for DFS position of accepting states.

Generalized Büchi Automaton; one lowlink per state on the DFS stack; a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Couvreur [1999] re-disCOVERS the lowlink aspect of Tarjan's algorithm, starting from Dijkstra's. It introduces a hybrid algorithm between SCC-based and NDFS-based algorithms, with an acceptance set per element in the root stack.

Couvreur et al. [2005] further develop this hybrid algorithm by restoring the SCC-based aspect of the algorithm by storing states in the same SCC. They propose two new heuristics using characteristics of Dijkstra's algorithm: Counterexamples extraction.

Geldenhuys and Valmari [2005] COMBINE Geldenhuys and Valmari's work with Couvreur's to produce a more efficient data structure, introducing counterexamples extraction.

Cheriyan and Mehlhorn [1996] OPTIMIZE their algorithm for dense explicit graphs, focusing on theoretical complexity analysis.

Gabow [2000] re-Discover Cheriyan and Mehlhorn's idea, starting from Tarjan's work, introducing emptiness checks similar to Couvreur et al.'s.

Alur et al. [2005] propose an emptiness check similar to Couvreur et al. for Büchi Automata.

Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. for Büchi Automata, focusing on extraction of small counterexamples.

Gaiser and Schwoon [2009] propose an emptiness check similar to Couvreur et al. for Generalized Büchi Automata.

LPAR'19 mixES all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching, utilizing compressed root stack.

Alur et al. [2005] and Hansen and Geldenhuys [2008] suggest the use of a Union-Find data structure to perform the SCC computation.

- Combines Geldenhuys and Valmari [2004] and Couvreur [1999];
- More efficient data structure;
- Counterexamples extraction;


- Optimisation for dense explicit graph;
- Theoretical complexity analysis;
- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
- Suggests the use of a Union-Find to perform the SCC computation;
- Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed root stack;
- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
- Use a Union-Find data structure to avoid the cost of marking dead an SCC;
- Compatible root stack compression;

E. Renault

Thursday, December 19th

LPAR - 2013

E. Renault
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has its lowlink:

\[
\text{LIVE stack size} + 1;
\]

For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.

Geldenhuys and Valmari [2004] combine Geldenhuys and Valmari [2004] and Couvreur [1999]; they introduce an extra stack for DFS position of accepting states.

LPAR'19 generalizes the Büchi Automaton by associating one lowlink per state on the DFS stack and introducing a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.


Couvreur et al. [2005] restore the SCC-based aspect of the algorithm by storing states in the same SCC. They propose two new heuristics using characteristics of Dijkstra's algorithm: the first one is for optimisation for dense explicit graph while the second is theoretical complexity analysis.


Alur et al. [2005] propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton.

Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; they suggest the use of a Union-Find to perform the SCC computation.


Cheriyan and Mehlhorn [1996] optimise for dense explicit graph; they also perform theoretical complexity analysis.


Nuutila and Soisalon-Soininen [1994] and Pearce [2005] suggest the use of a Union-Find data structure to avoid the cost of marking dead SCC; they also suggest the use of a Union-Find data structure to perform the SCC computation.

Alur et al. [2005] propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton; they suggest the use of a Union-Find to perform the SCC computation.

Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; they suggest the use of a Union-Find to perform the SCC computation.

Gaiser and Schwoon [2009] propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; they suggest the use of a Union-Find to perform the SCC computation.

Cheriyan and Mehlhorn [1996]

- Optimisation for dense explicit graph;
- Theoretical complexity analysis;


Geldenhuys and Valmari [2005]

- More efficient data structure;
- Counterexamples extraction;

Couvreur et al. [2005]

- Restores the SCC-based aspect of the algorithm by storing states in the same SCC;
- Two new heuristics using characteristic of Dijkstra’s algorithm;
- Counterexamples extraction;

Gabow [2000]

- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];

Alur et al. [2005]

- Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008]

- Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;
- Extraction of small counterexamples;

Gaiser and Schwoon [2009]

- Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

LPAR’19

- Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed root stack;

Rediscovers Cheriyan and Mehlhorn [1996]

- Suggests the use of a Union-Find to perform the SCC computation;

Nuutila and Soisalon-Soininen [1994]

Pearce [2005]
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has its lowlink set to the LIVE stack size + 1. For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.

Geldenhuys and Valmari [2004] apply a Büchi Automaton with one lowlink per LIVE state. An extra stack for DFS position of accepting states is used. As a result, a Generalized Büchi Automaton is obtained with one lowlink per state on the DFS stack and a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] associates a DFS Position identifier to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination. If a state that has a DFS position equal to the top of the root stack is a root: when this state will be popped, all states with a greater LIVE number will be removed from the LIVE stack.


Couvreur et al. [2005] restore the SCC-based aspect of the algorithm by storing states in the same SCC. Two new heuristics using characteristics of Dijkstra's algorithm are proposed. Counterexamples extraction is also considered.

Geldenhuys and Valmari [2005] combine Geldenhuys and Valmari [2004] and Couvreur [1999], leading to a more efficient structure. Counterexamples extraction is still considered.

Cheriyan and Mehlhorn [1996] focus on optimizing their algorithm for dense explicit graphs. Theoretical complexity analysis is also presented.


Alur et al. [2005] propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton.

Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton with the extraction of small counterexamples.


LPAR’19 mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching, using compressed root stack and suggesting the use of a Union-Find data structure to perform the SCC computation.

- RedisCOVERS Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];


Dijkstra [1973] → Couvreur [1999]

Geldenhuys and Valmari [2005]

Couvreur et al. [2005]

Gaiser and Schwoon [2009] → LPAR’19

Use a Union-Find data structure to avoid the cost of marking a SCC; Compatible root stack compression;


Optimisation for dense explicit graph; Theoretical complexity analysis;


E. Renault LPAR - 2013 Thursday, December 19th 7 / 14

- Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;


- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
- Suggests the use of a Union-Find to perform the SCC computation;


- Restores the SCC-based aspect of the algorithm by storing states in the same SCC;
- Two new heuristics using characteristic of Dijkstra’s algorithm;
- Counterexamples extraction;

Gabow [2000] → Alur et al. [2005]

- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];

Gaiser and Schwoon [2009] → LPAR’19

- Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Nuutila and Soisalon-Soininen [1994] → LPAR’19

- Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed root stack;

E. Renault

LPAR - 2013

Thursday, December 19th
Tarjan [1972] Associates an identifier (lowlink) to each state on the DFS stack; These lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink: \( \text{LIVE stack size} + 1 \); For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value; If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] Büchi Automaton; One lowlink per LIVE state; An extra stack for DFS position of accepting states.

LPAR'19 Generalized Büchi Automaton; One lowlink per state on the DFS stack; A set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] Associates an identifier (DFS Position) to each state on the DFS stack; These DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination; If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.


Couvreur et al. [2005] Restores the SCC-based aspect of the algorithm by storing states in the same SCC; Two new heuristics using characteristic of Dijkstra's algorithm; Counterexamples extraction.

Geldenhuys and Valmari [2005] Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; More efficient data structure; Counterexamples extraction.


Hansen and Geldenhuys [2008] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; Gaiser and Schwoon [2009] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; LPAR’19 Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; Compressed root stack.


- Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;
- Extraction of small counterexamples;

Geldenhuys and Valmari [2004]

- Combines Geldenhuys and Valmari [2004] and Couvreur [1999];
- More efficient data structure;
- Counterexamples extraction;

Couvreur [1999]

- Rediscovers Dijkstra [1973] starting from Tarjan [1972];
- Hybrid algorithm between SCC-based and NDFS-based;
- An acceptance set per element in the root stack;

Couvreur et al. [2005]

- Restores the SCC-based aspect of the algorithm by storing states in the same SCC;
- Two new heuristics using characteristic of Dijkstra’s algorithm;
- Counterexamples extraction;

Gabow [2000]

- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
- Theoretical complexity analysis;

Cheriyan and Mehlhorn [1996]

- Optimisation for dense explicit graph;
- Alur et al. [2005] Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Gabow [2000]

- Alur et al. [2005] Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;
- Hansen and Valmari [2005] Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;

Hansen and Valmari [2005]

- Extraction of small counterexamples;

Couvreur [1999]

- Compressed root stack;
- LPAR’19 Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;

Hansen and Valmari [2005]

- LPAR’19 Use a Union-Find data structure to avoid the cost of marking dead an SCC; Compatible root stack compression;

Nuutila and Soisalon-Soininen [1994]

- Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;


- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];


- Alur et al. [2005] Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;


- Hansen and Geldenhuys [2008] Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;


- Gaiser and Schwoon [2009] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Gabow [2000] → Alur et al. [2005]


- Hansen and Geldenhuys [2008] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Couvreur [1999] → Alur et al. [2005]

- Couvreur [1999] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Couvreur et al. [2005] → Alur et al. [2005]

- Couvreur et al. [2005] Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Gaiser and Schwoon [2009] → Couvreur et al. [2005]

- Gaiser and Schwoon [2009] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;
Tarjan [1972] Associates an identifier (lowlink) to each state on the DFS stack; These lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink: LIVE stack size + 1; For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value; If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] Büchi Automaton; One lowlink per LIVE state; An extra stack for DFS position of accepting states.

Dijkstra [1973] Associates an identifier (DFS Position) to each state on the DFS stack; These DFS Position are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination; If a state that has a DFS Position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Couvreur [1999] Generalized Büchi Automaton; Rediscovers Dijkstra [1973] starting from Tarjan [1972]; Hybrid algorithm between SCC-based and NDFS-based; An acceptance set per element in the root stack; Couvreur et al. [2005] Restores the SCC-based aspect of the algorithm by storing states in the same SCC; Two new heuristics using characteristic of Dijkstra's algorithm; Counterexamples extraction.

Couvreur et al. [2005] Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; More efficient data structure; Counterexamples extraction.


Hansen and Geldenhuys [2008] Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; E. Renault LPAR - 2013 Thursday, December 19th 7 / 14
Tarjan [1972] 

Geldenhuys and Valmari [2004] 

LPAR’19 

- Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed root stack;

Couvreur et al. [2005] 

Restores the SCC-based aspect of the algorithm by storing states in the same SCC;
- Two new heuristics using characteristic of Dijkstra’s algorithm;
- Counterexamples extraction;

Couvreur [1999] 

Generalized Büchi Automaton;
- Rediscovers Dijkstra [1973] starting from Tarjan [1972];
- Hybrid algorithm between SCC-based and NDFS-based;
- An acceptance set per element in the root stack;

Cheriyan and Mehlhorn [1996] 

Optimisation for dense explicit graph;
- Theoretical complexity analysis;

Dijkstra [1973] 

Associates an identifier (DFS Position) to each state on the DFS stack;
- These DFS Positions are stored in a root stack;
- When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination;
- If a state that has a DFS position equal to the top of the root stack it’s a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Gelderhuys and Valmari [2004] 

Büchi Automaton;
- One lowlink per LIVE state;
- An extra stack for DFS position of accepting states;

Gabow [2000] 

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];

Alur et al. [2005] 

Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008] 

LPAR’19 

Use a Union-Find data structure to avoid the cost of marking dead an SCC;
- Compatible root stack compression;

Hansen and Geldenhuys [2008] 

Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;

Gabow [2000] 

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];

Alur et al. [2005] 

Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Gaiser and Schwoon [2009] 

Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Gaiser and Schwoon [2009] 

Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008] 

Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Gaiser and Schwoon [2009] 

Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed root stack;
Tarjan [1972] Associates an identifier (lowlink) to each state on the DFS stack; These lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink: \( \text{LIVE stack size} + 1 \); For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value; If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] Büchi Automaton; One lowlink per LIVE state; An extra stack for DFS position of accepting states.

LPAR'19 Generalized Büchi Automaton; One lowlink per state on the DFS stack; A set of acceptance sets per element in the lowlink stack.


Couvreur et al. [2005] Restores the SCC-based aspect of the algorithm by storing states in the same SCC; Two new heuristics using characteristic of Dijkstra's algorithm; Counterexamples extraction.

Geldenhuys and Valmari [2005] Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; More efficient data structure; Counterexamples extraction.


Alur et al. [2005] Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008] Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; Extraction of small counterexamples.

Gaiser and Schwoon [2009] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Nuutila and Soisalon-Soininen [1994] LPAR'19 Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; Compressed root stack; Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; Suggests the use of a Union-Find to perform the SCC computation;

Gabow [2000] LPAR'19 Use a Union-Find data structure to avoid the cost of marking dead an SCC; Compatible root stack compression;
Tarjan [1972]

Associates an identifier (lowlink) to each state on the DFS stack; These lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has its lowlink set to LIVE stack size() + 1. For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value. If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004]

Büchi Automaton; One lowlink per LIVE state; An extra stack for DFS position of accepting states;

Dijkstra [1973]

Associates an identifier (DFS Position) to each state on the DFS stack; These DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination; If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Couvreur [1999]

Generalized Büchi Automaton; Rediscovers Dijkstra [1973] starting from Tarjan [1972]; Hybrid algorithm between SCC-based and NDFS-based; An acceptance set per element in the root stack;

Couvreur et al. [2005]

Restores the SCC-based aspect of the algorithm by storing states in the same SCC; Two new heuristics using characteristic of Dijkstra’s algorithm; Counterexamples extraction;

C metre and Mehlhorn [1996]

Optimisation for dense explicit graph; Theoretical complexity analysis;

Gabow [2000]

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; Suggests the use of a Union-Find to perform the SCC computation;

Hansen and Geldenhuys [2008]

Suggests the use of a Union-Find to perform the SCC computation;

Gaiser and Schwoon [2009]

LPAR’19

Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; Compressed root stack;

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; Suggests the use of a Union-Find to perform the SCC computation;

LPAR’19

Use a Union-Find data structure to avoid the cost of marking dead an SCC; Compatible root stack compression;

Pearce [2005]


Gaiser and Schwoon [2009] → LPAR’19

Gelderhuys and Valmari [2005] → LPAR’19

Couvreur et al. [2005] → LPAR’19


Alur et al. [2005] Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008] Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;

Couvreur et al. [2005] Restores the SCC-based aspect of the algorithm by storing states in the same SCC;

Two new heuristics using characteristic of Dijkstra’s algorithm;

Counterexamples extraction;

Gaiser and Schwoon [2009] Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Suggests the use of a Union-Find to perform the SCC computation;

LPAR’19 Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;

Compressed root stack;

Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];

Suggests the use of a Union-Find to perform the SCC computation;

Tarjan [1972] - Associates an identifier (lowlink) to each state on the DFS stack; These lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink: LIVE stack size() + 1; For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value; If a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] - Büchi Automaton; One lowlink per LIVE state; An extra stack for DFS position of accepting states.

LPAR'19 - Generalized Büchi Automaton; One lowlink per state on the DFS stack; A set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] - Associates an identifier (DFS Position) to each state on the DFS stack; These DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination; If a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.


Couvreur et al. [2005] - Restores the SCC-based aspect of the algorithm by storing states in the same SCC; Two new heuristics using characteristic of Dijkstra's algorithm; Counterexamples extraction.

Geldenhuys and Valmari [2005] - Combines Geldenhuys and Valmari [2004] and Couvreur [1999]; More efficient data structure; Counterexamples extraction.


Alur et al. [2005] - Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;

Hansen and Geldenhuys [2008] - Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; Extraction of small counterexamples.

Gaiser and Schwoon [2009] - Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

Nuutila and Soisalon-Soininen [1994] - Suggests the use of a Union-Find to perform the SCC computation;


Hansen and Geldenhuys [2008] - Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;

LPAR'19 - Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; Compressed root stack.

Use a Union-Find data structure to avoid the cost of marking dead an SCC; Compatible root stack compression;
Gabow – Back to the example

Current state
DFS stack

LIVE state

DEAD state

Root stack

<table>
<thead>
<tr>
<th></th>
<th>s₁</th>
<th>s₅</th>
<th>s₈</th>
<th>s₉</th>
<th>s₁₂</th>
<th>s₁₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>✓</td>
<td>∅</td>
</tr>
</tbody>
</table>

UF

DEAD

s₁ s₂ s₃ s₄ s₅ s₆ s₇ s₈ s₉ s₁₀ s₁₁ s₁₂ s₁₃
Gabow – Back to the example

Current state

LIVE state

DEAD state

Root stack

UF

Root stack:

<table>
<thead>
<tr>
<th>s1</th>
<th>s5</th>
<th>s8</th>
<th>s9</th>
<th>s12</th>
<th>s13</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>🔵</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

UF:

DEAD:

s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13
Gabow – Back to the example

Current state
DFS stack

LIVE state

DEAD state

LIVE number
LIVE stack

Root stack

Root stack

Root stack

UF

UF

UF

UF
Let’s benchmark!

- Models from the BEEM benchmark
- 448 empty products where the emptiness check takes at least 10 seconds on an Intel 64-bit Xeon @ 2.00 GHz
- 412 non-empty products
- Union-Find uses common optimizations:
  - Link by Rank
  - Immediate Parent Check
  - Memory Smart
  - Path Compression
Comparisons of emptiness checks

The three algorithms are comparable.

Dijkstra-based emptiness check is the best memory efficient and can benefit from a compressed stack!

Tarjan-based is the faster when bit state hashing and state space caching are not used!
Conclusion

- Comparision of generalized emptiness checks for the automata theoretic approach to model checking;
- Improve Dijkstra SCC computation algorithm;
- First emptiness check based on a Union-Find data structure;
- Memory comparison.
Tarjan [1972] associates an identifier (lowlink) to each state on the DFS stack; these lowlinks are stored in a lowlink stack. Every new state pushed on the DFS stack has for lowlink:

\[ \text{LIVE stack size} + 1; \]

For every backtrack, the lowlink at the top of the lowlink stack will be affected to a smaller or equal value; if a state that has a lowlink equal to its LIVE number it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.

Geldenhuys and Valmari [2004] introduce the Büchi Automaton; one lowlink per LIVE state; an extra stack for DFS position of accepting states; Generalized Büchi Automaton; one lowlink per state on the DFS stack; a set of acceptance sets per element in the lowlink stack.

Dijkstra [1973] associates an identifier (DFS Position) to each state on the DFS stack; these DFS Positions are stored in a root stack. When a backedge is found, the root stack is updated until the top of this stack is lesser or equal to the DFS Position of the destination; if a state that has a DFS position equal to the top of the root stack it's a root: when this state will be popped, all states with a greater LIVE number will be removed from LIVE stack.


Couvreur et al. [2005] restores the SCC-based aspect of the algorithm by storing states in the same SCC; two new heuristics using characteristic of Dijkstra's algorithm; counterexamples extraction.

Geldenhuys and Valmari [2005] combines Geldenhuys and Valmari [2004] and Couvreur [1999]; more efficient data structure; counterexamples extraction.

Cheriyan and Mehlhorn [1996] optimize for dense explicit graph; theoretical complexity analysis.


Alur et al. [2005] propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton.

Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton; extraction of small counterexamples.


Hansen and Geldenhuys [2008] propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton.

Couvreur et al. [2005] propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton; counterexamples extraction.

Nuutila and Soisalon-Soininen [1994] suggest the use of a Union-Find to perform the SCC computation.

Gabow [2000] suggests the use of a Union-Find to perform the SCC computation; compatible root stack compression.

LPAR’19 mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching; compressed root stack; rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972]; suggests the use of a Union-Find to perform the SCC computation.

Nuutila and Soisalon-Soininen [1994]

Couvrleur [1999] → Couvrleur et al. [2005]


Gaiser and Schwoon [2009] → LPAR’19


Gabow [2000]

Gaiser and Schwoon [2009]


Couvreur [1999] → Couvrleur et al. [2005]


Couvreur [1999] → Couvrleur et al. [2005]


Future work...

- Integrate Nuutila’s optimisation in all algorithms.
- Compressed stack for Tarjan’s algorithm.
- Build a Tarjan-based algorithm with a Union-Find data structure.
- Explore parallel set-ups for these algorithms.
Future work...

- Integrate Nuutila’s optimisation in all algorithms.
- Compressed stack for Tarjan’s algorithm.
- Build a Tarjan-based algorithm with a Union-Find data structure.
- Explore parallel set-ups for these algorithms.

Questions?


Bibliography II


Tarjan – Back to the example

Current state

DFS stack: s_1, s_2, s_5, s_8, s_9, s_10, s_12

LIVE state

LIVE stack: s_1, s_2, s_3, s_4, s_5, s_8, s_9, s_10, s_11, s_12, s_13

DEAD state

Lowlink stack

Lowlink stack:

Compressed Root Stack

1 1 5 6 7 7 10 11
∅ ∅ ∅ ∅ ∅ ∅ ∅ ∅
Tarjan – Back to the example

Current state

DFS stack

LIVE state

DEAD state

LIVE number

Lowlink stack

LIVE stack

Compressed Root Stack

E. Renault
Dijkstra – Back to the example

Root stack

<table>
<thead>
<tr>
<th>State</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIVE</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>×</td>
<td>∅</td>
</tr>
</tbody>
</table>

LIVE stack

<table>
<thead>
<tr>
<th>State</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Dijkstra – Back to the example

Current state
DFS stack

LIVE state

DEAD state

LIVE number

Root stack

Compressed Root Stack

LIVE stack
Dijkstra – Back to the example

Current state
DFS stack
LIVE state
DEAD state
LIVE number
Root stack
Compressed Root Stack
LIVE stack

\[
\begin{array}{cccc}
\{s_1\} & \{s_5\} & \{s_8\} & \{s_9\} \\
\emptyset & \emptyset & \emptyset & \bullet \cdot \\
\end{array}
\]

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccccccccccc}
\{s_1\} & \{s_2\} & \{s_3\} & \{s_4\} & \{s_5\} & \{s_8\} & \{s_9\} & \{s_{10}\} & \{s_{11}\} & \{s_{12}\} & \{s_{13}\} \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]