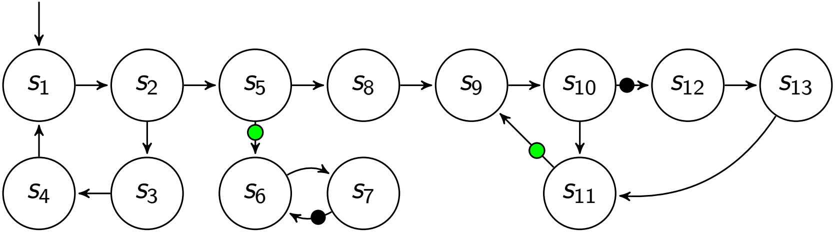
Three SCC-based Emptiness Checks for Generalized Büchi Automata

LPAR'19

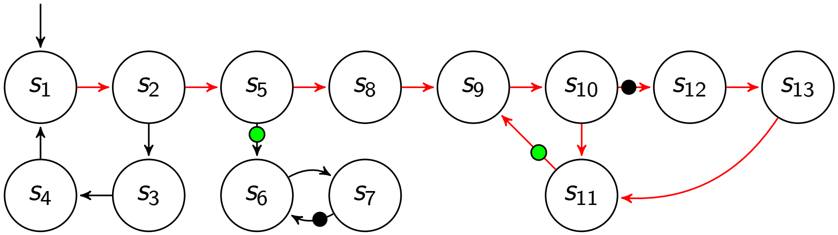
E. Renault, A. Duret-Lutz, F. Kordon, D. Poitrenaud

Thursday, December 19th

Transition-based Generalized Büchi Automata

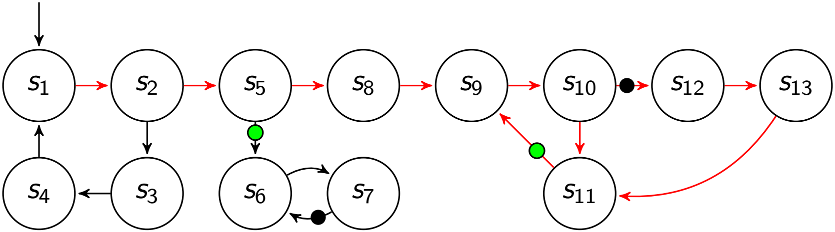


Transition-based Generalized Büchi Automata



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.

	NDFS-based	SCC-based
On-the-Fly	✓	✓
Bit state hashing	all states but DFS	only dead SCCs
State space caching	all states but DFS	only dead SCCs
Max memory req. for BA	2 bits per state	1 int per state
Generalization	difficult	trivial
Earlier CE detection	–	✓

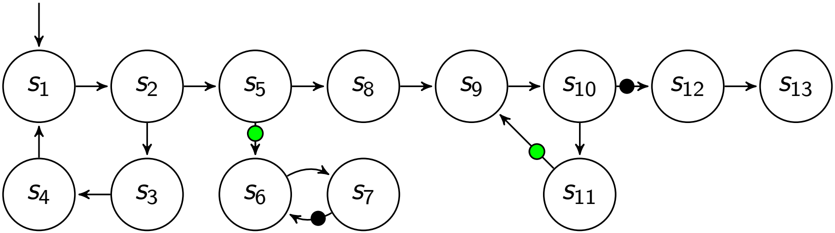
This Talk!

Is there a best explicit SCC computation algorithm?

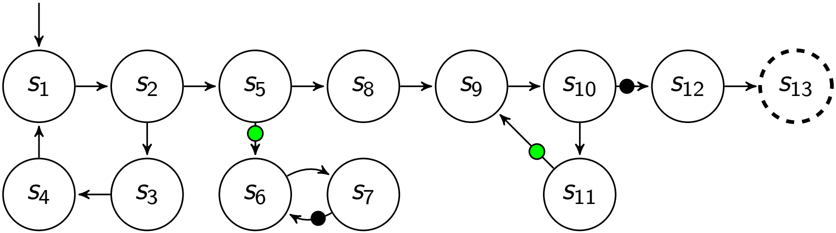
How to transform SCCs computation algorithms into generalized emptiness checks?

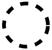
What is the cost of adding the emptiness check to an SCC computation algorithm?

Terminology

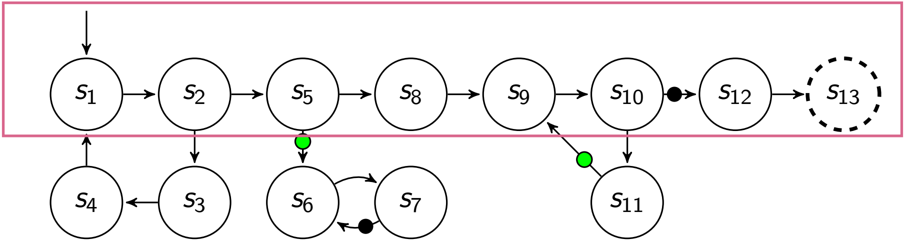


Terminology

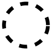


 Current state

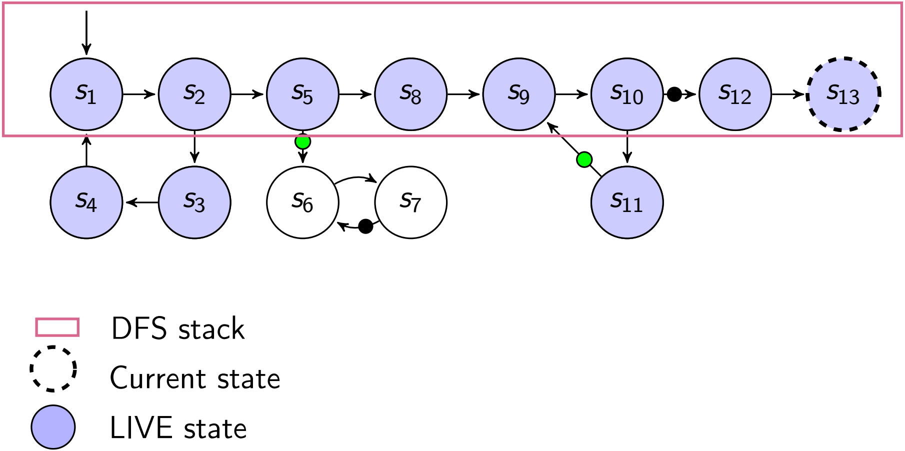
Terminology



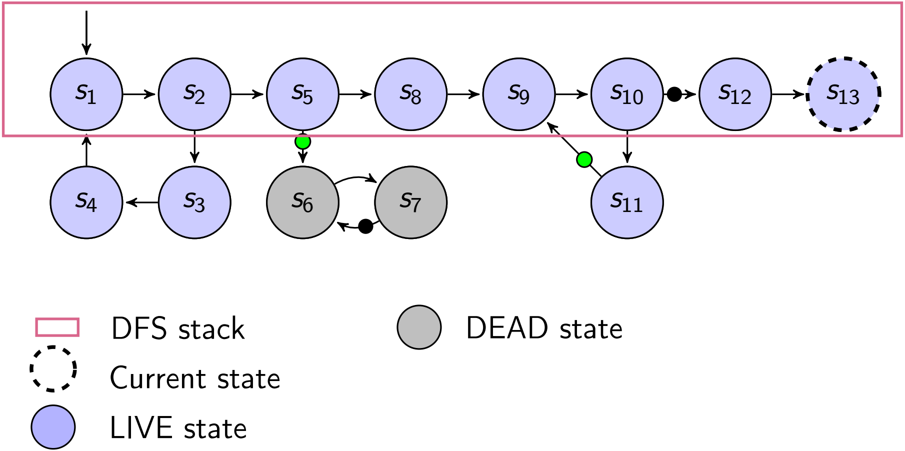

DFS stack


Current state

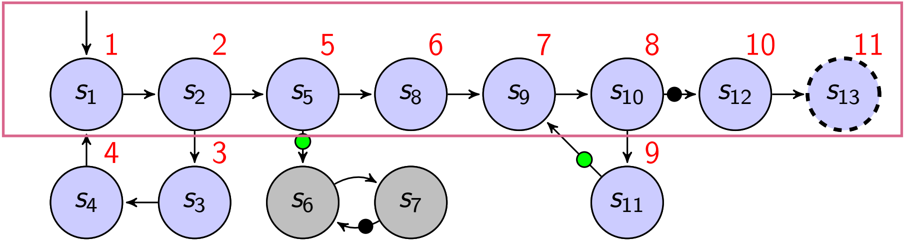
Terminology

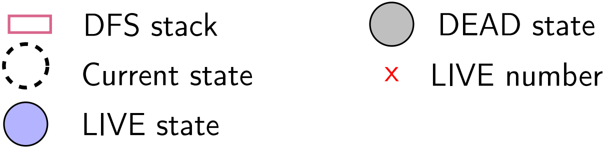


Terminology

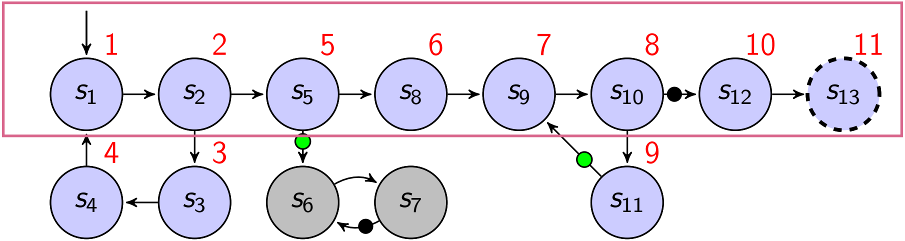


Terminology





Terminology



DFS stack

Current state

LIVE state

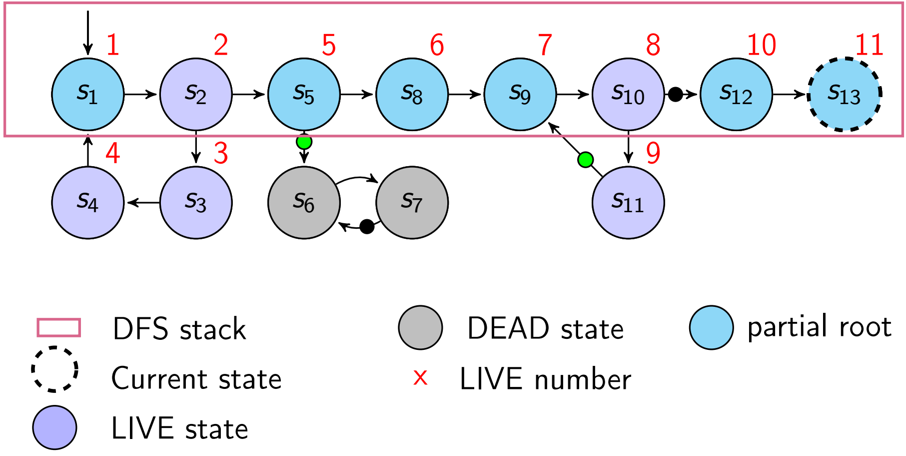
DEAD state

× LIVE number

LIVE stack

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

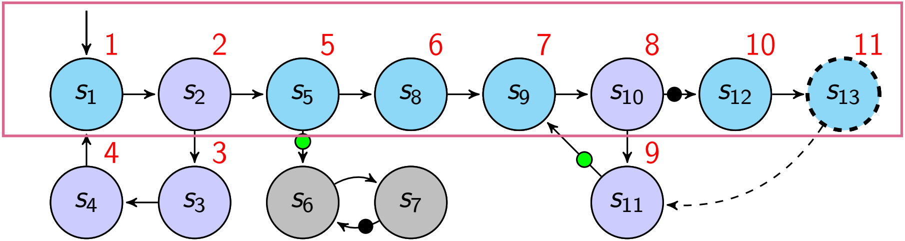
Terminology



LIVE stack

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

Terminology



DFS stack
Current state
LIVE state
DEAD state
x LIVE number
partial root
Backedge

LIVE stack	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

Tarjan [1972]

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 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]

Tarjan [1972] → Geldenhuys and Valmari [2004]

Dijkstra [1973]

Tarjan [1972] → Geldenhuys and Valmari [2004]

- 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

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

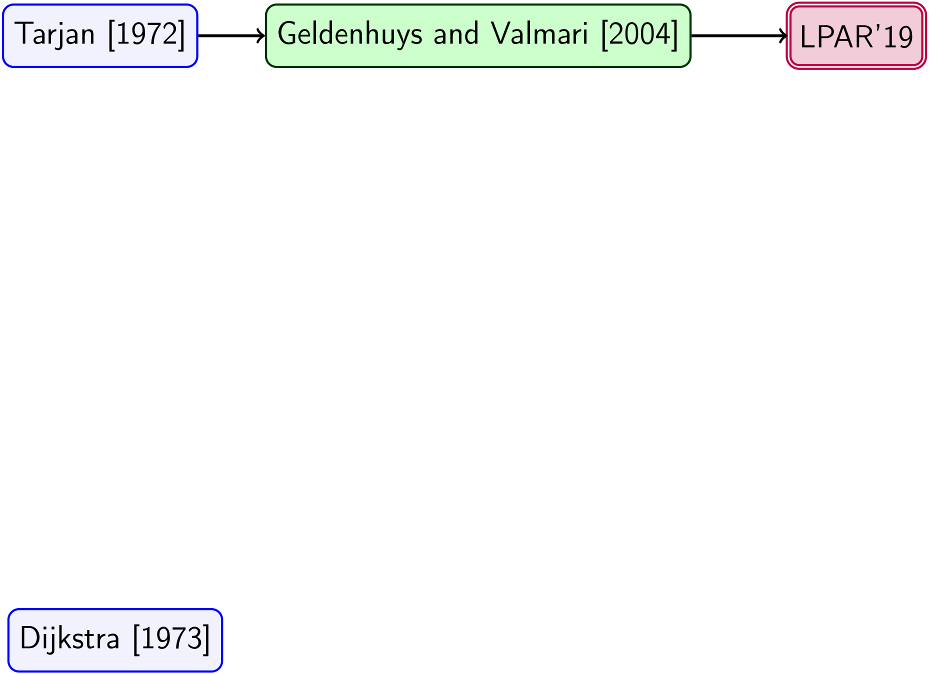
Dijkstra [1973]

Tarjan [1972] → Geldenhuys and Valmari [2004] → LPAR'19

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

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

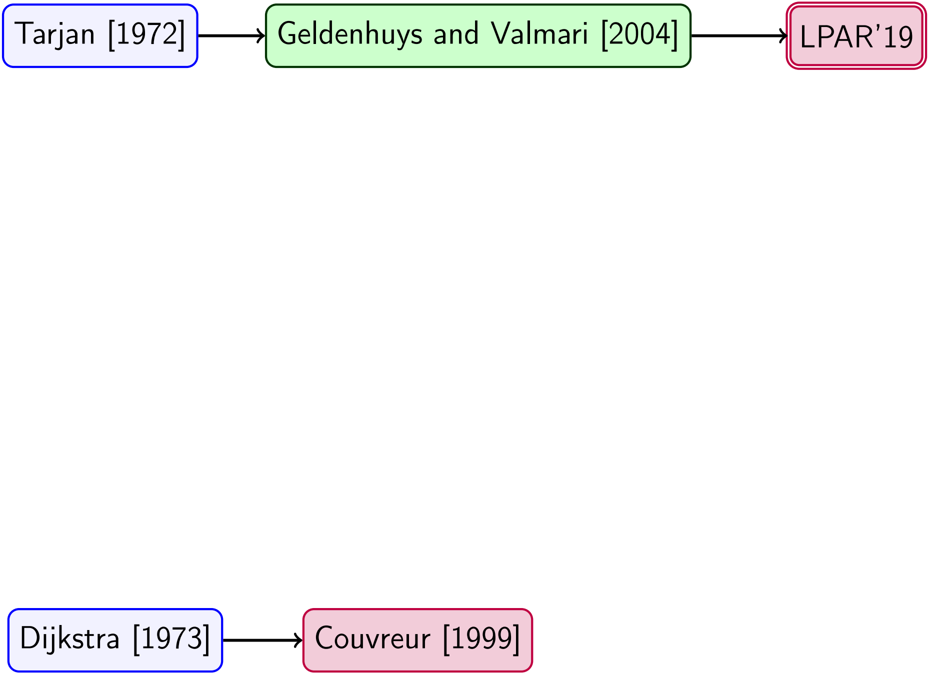
Dijkstra [1973]

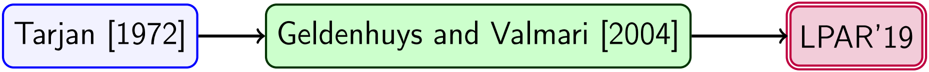

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Dijkstra [1973]

Tarjan [1972] → Geldenhuys and Valmari [2004] → LPAR'19

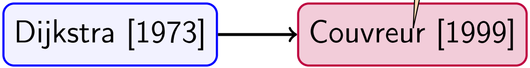
- Associates an identifier (*DFS Position*) to each state on the DFS stack;
- These *DFS Position* are stored in a *root stack*
- When a backedge is found, the *root stack* is updated until the top of this stack is lesser or equal to the *DFS Position* of the destination;
- If a state that has a *DFS 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.

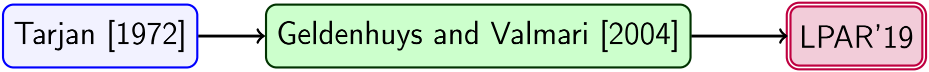
Dijkstra [1973]

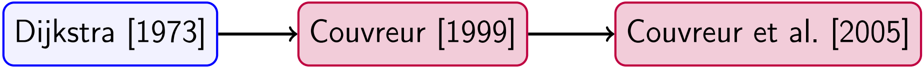

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Dijkstra [1973]
Couvreur [1999]



- 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*;



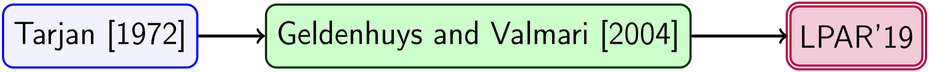

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19

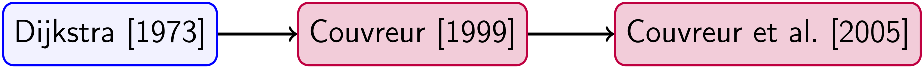

Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

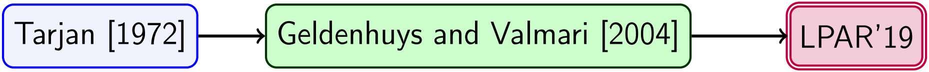
Tarjan [1972] → Geldenhuys and Valmari [2004] → LPAR'19

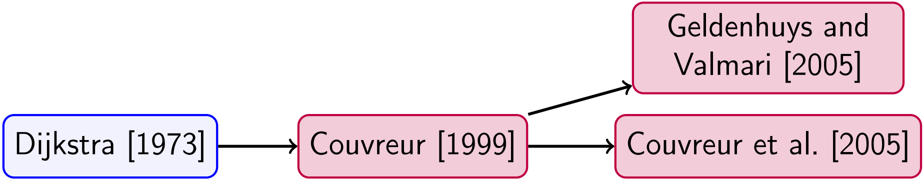
- Restores the SCC-based aspect of the algorithm by storing states in the same SCC;
- Two new heuristics using characteristic of Dijkstra's algorithm;
- Counterexamples extraction;

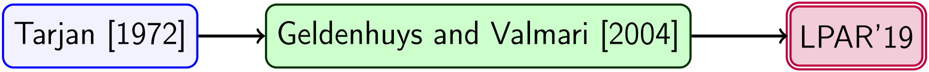
Dijkstra [1973] → Couvreur [1999] → Couvreur et al. [2005]


Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19

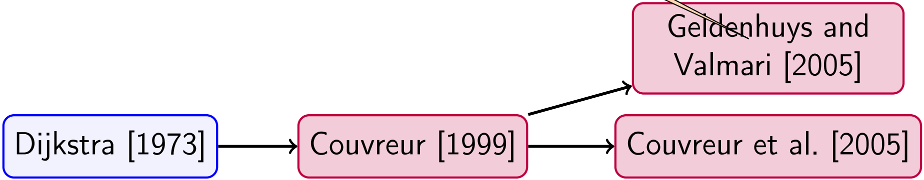

Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

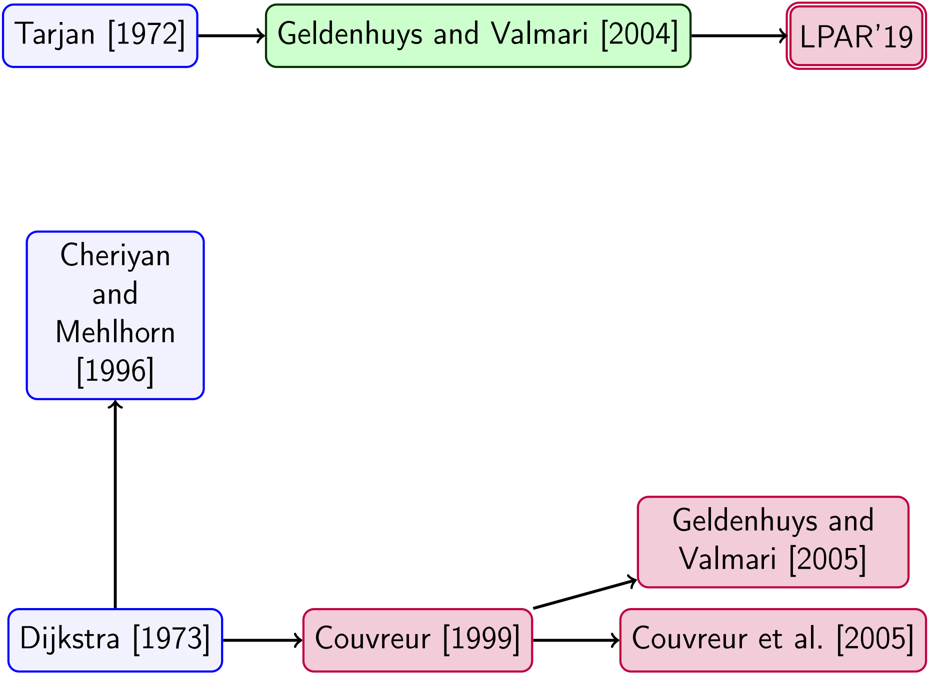

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19

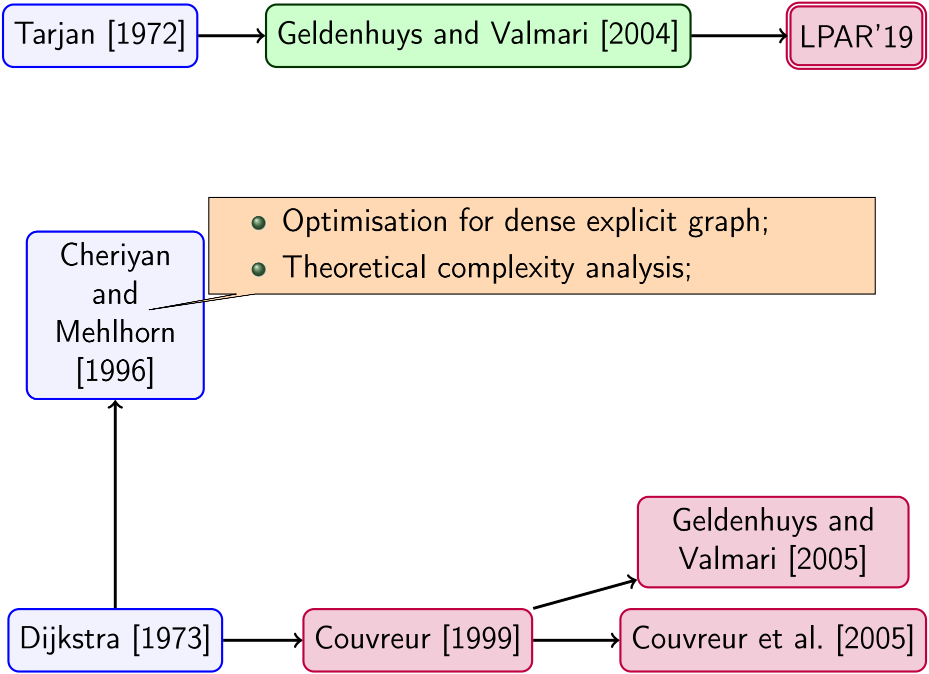

Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]



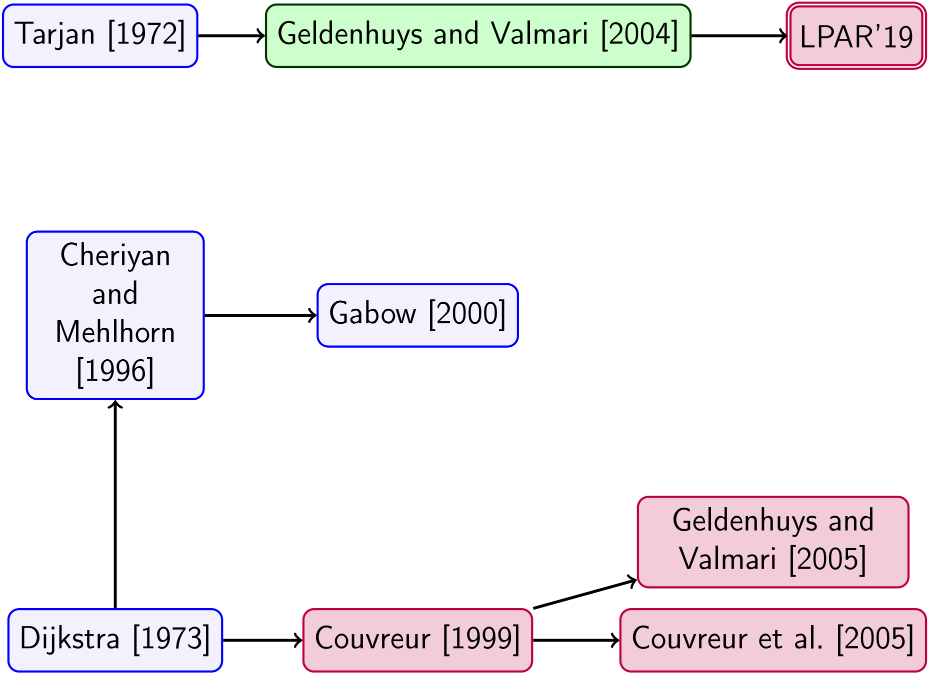
- Combines Geldenhuys and Valmari [2004] and Couvreur [1999];
- More efficient data strucutre;
- Counterexamples extraction;

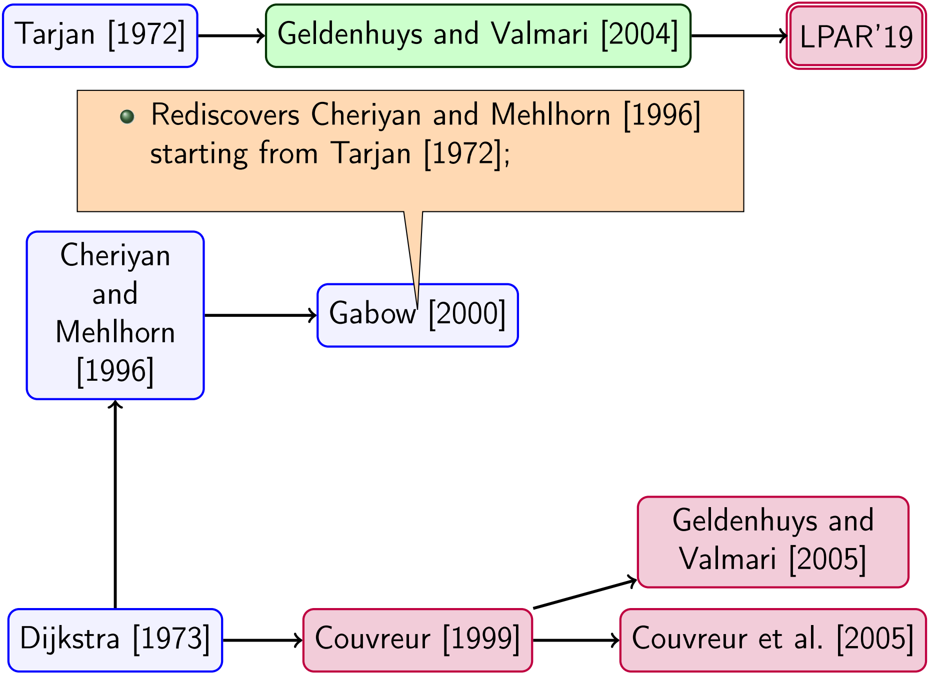



Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Cheriyan and Mehlhorn [1996]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

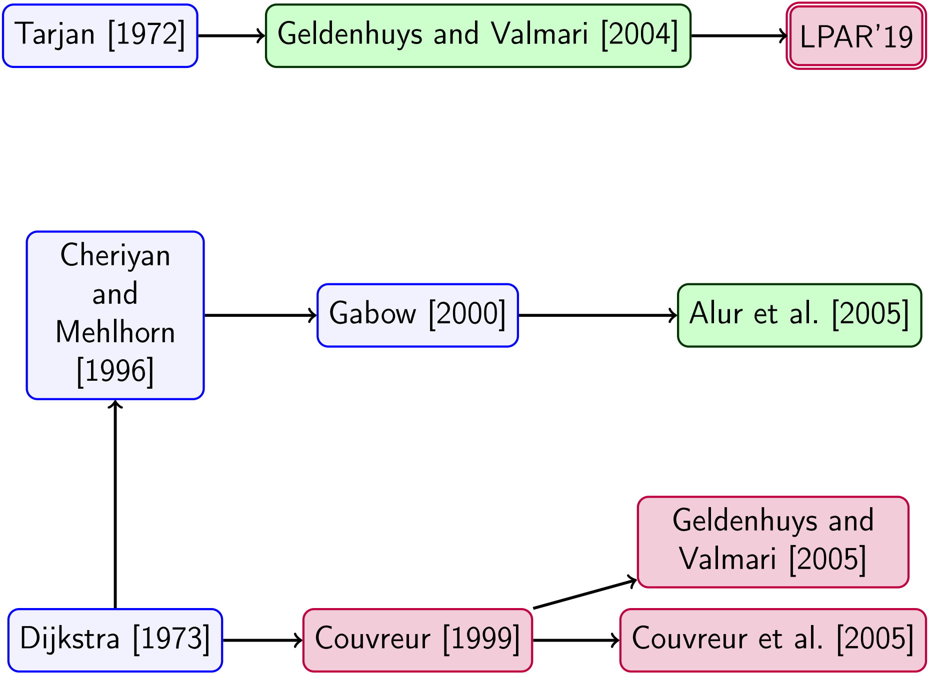


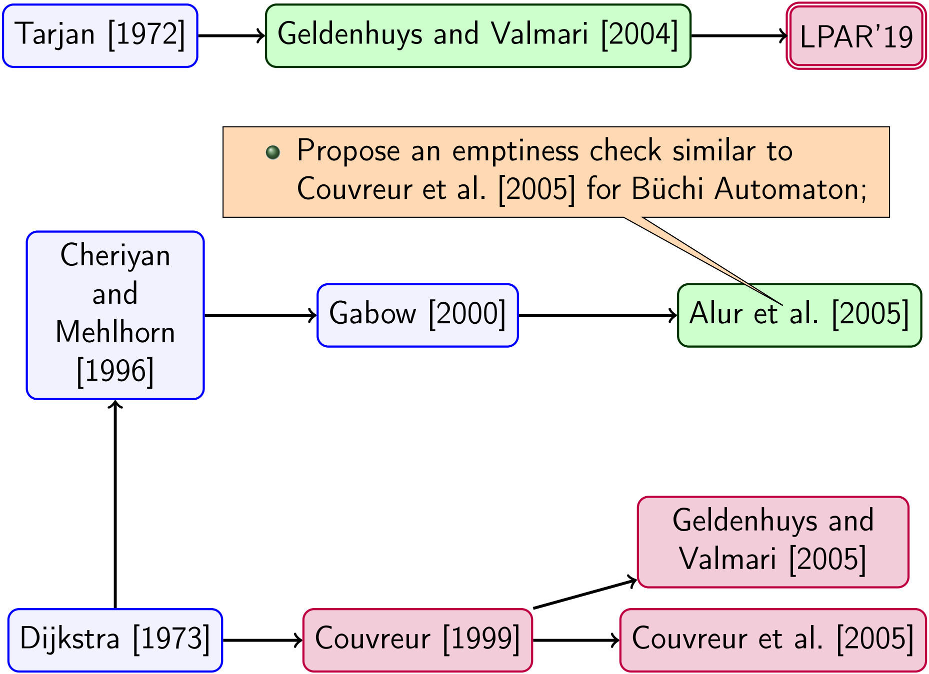
- Optimisation for dense explicit graph;
- Theoretical complexity analysis;

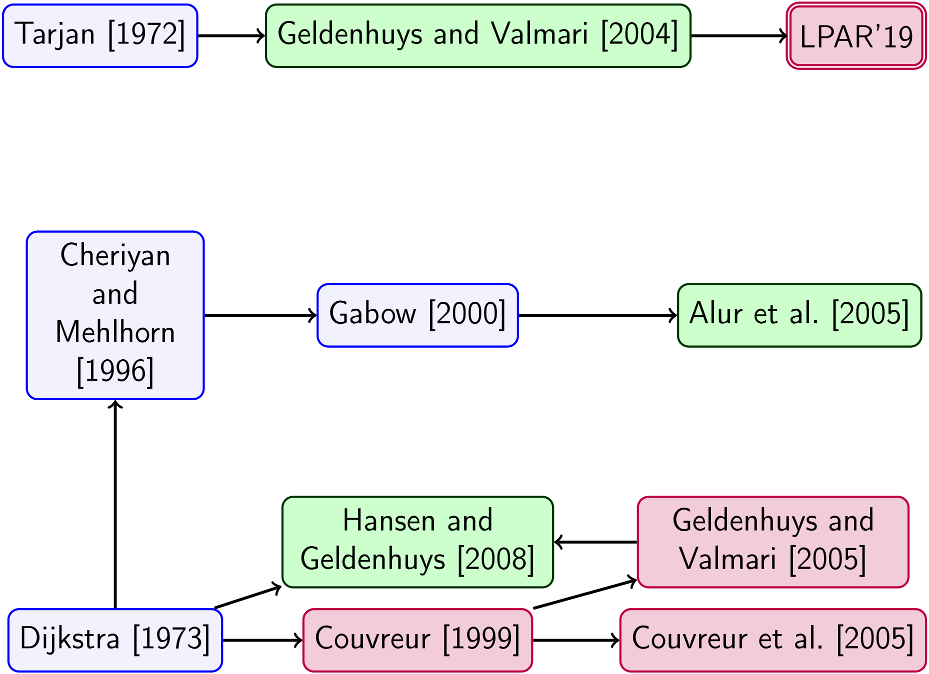

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

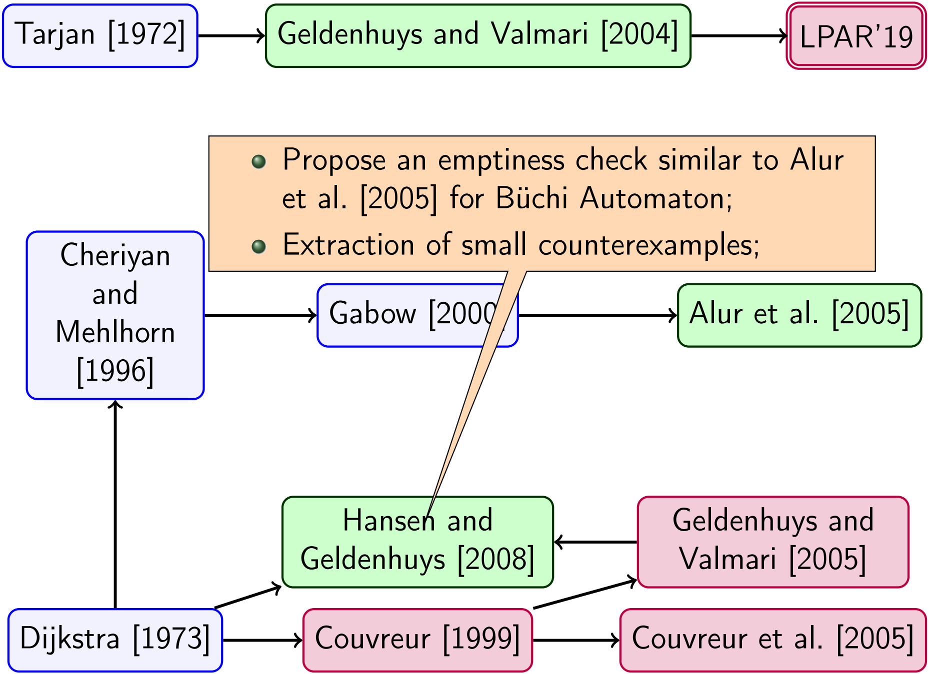


Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

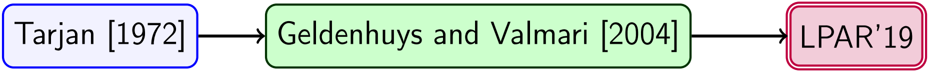

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

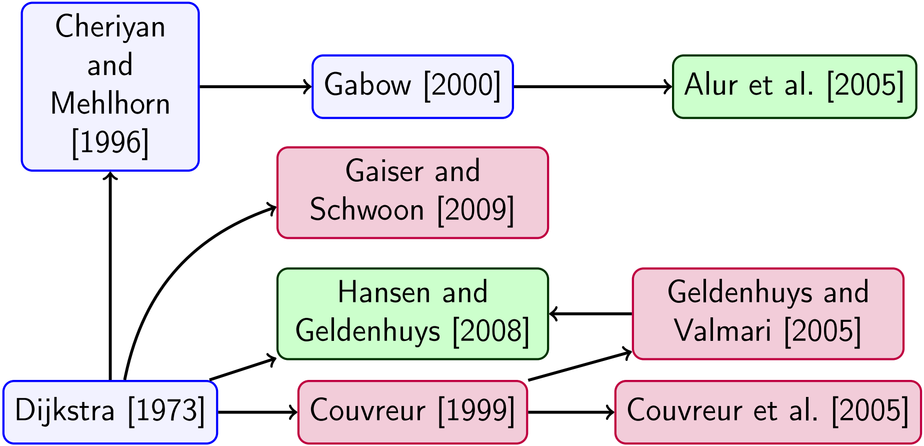

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Propose an emptiness check similar to Couvreur et al. [2005] for Büchi Automaton;
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

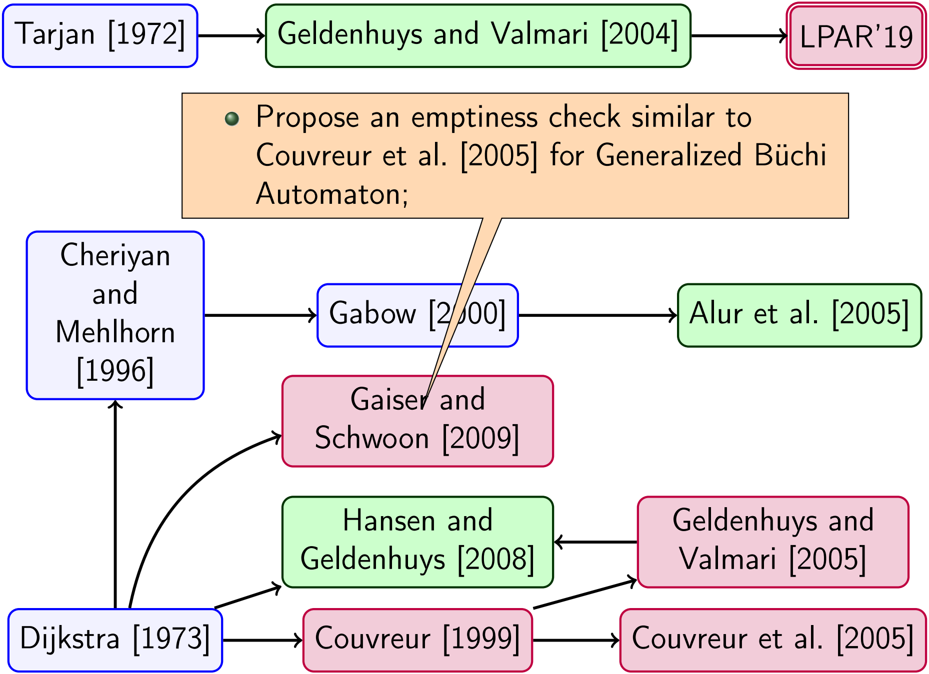

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

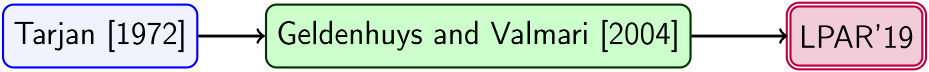


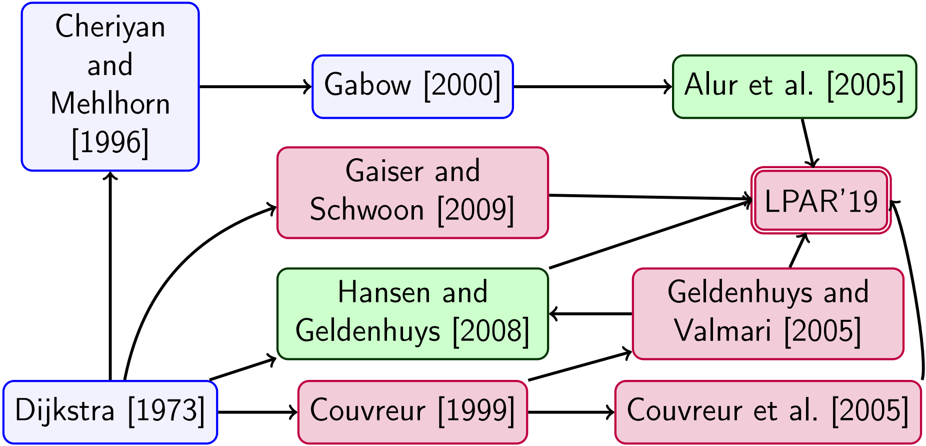
- Propose an emptiness check similar to Alur et al. [2005] for Büchi Automaton;
- Extraction of small counterexamples;

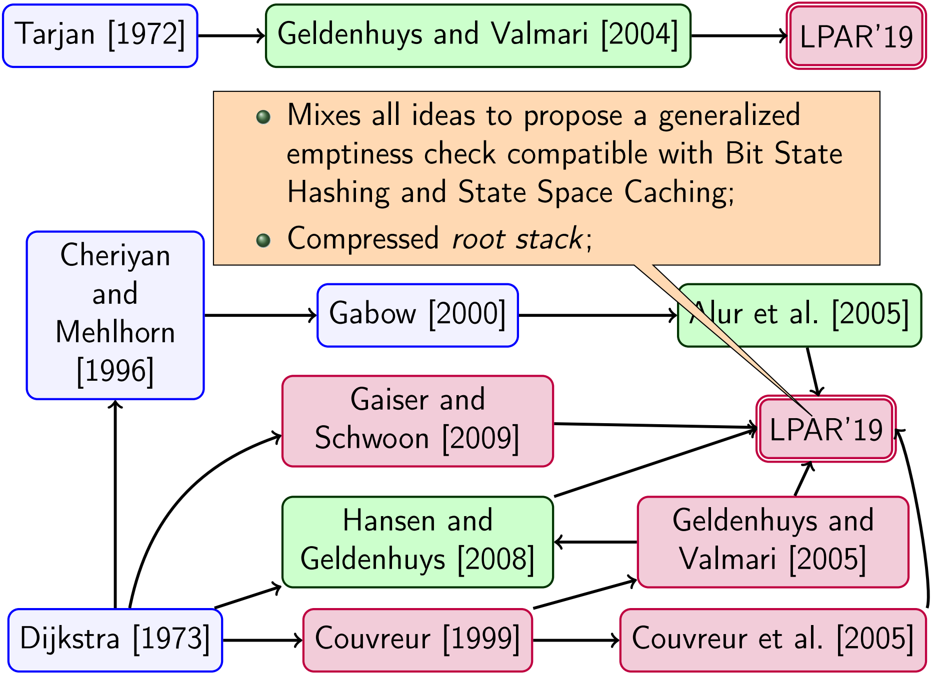

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19


Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

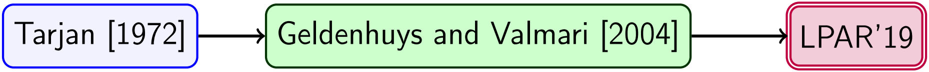

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Propose an emptiness check similar to Couvreur et al. [2005] for Generalized Büchi Automaton;
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

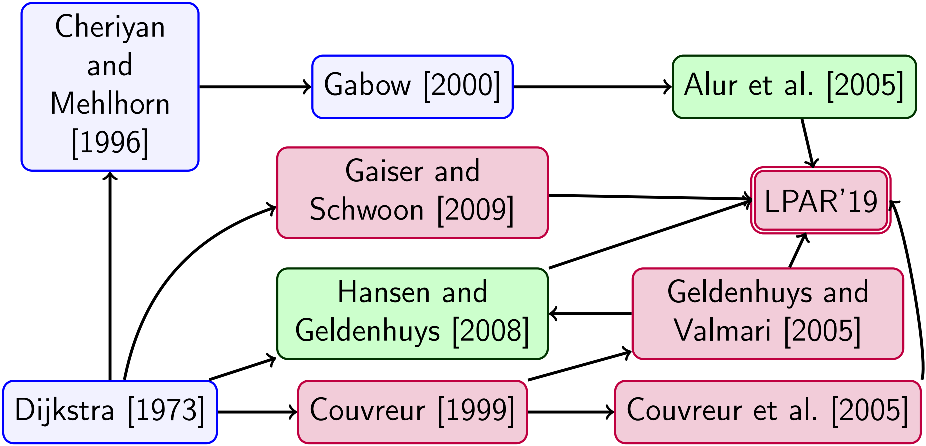

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19


Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
LPAR'19
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

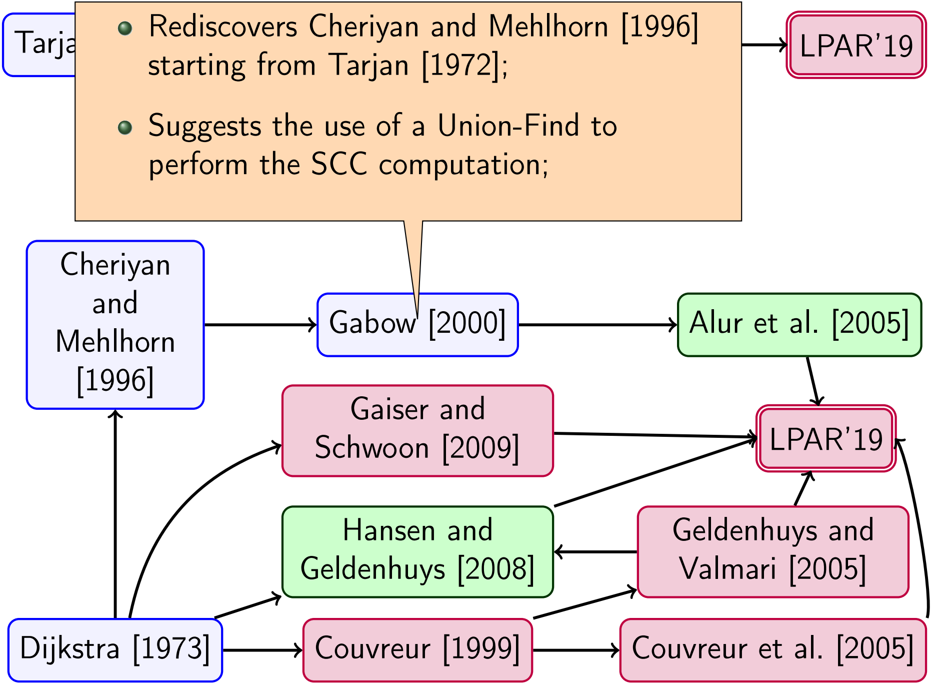


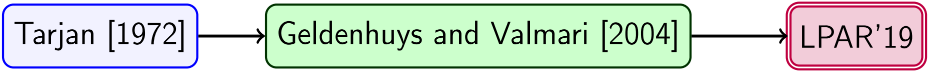
- Mixes all ideas to propose a generalized emptiness check compatible with Bit State Hashing and State Space Caching;
- Compressed *root stack*;

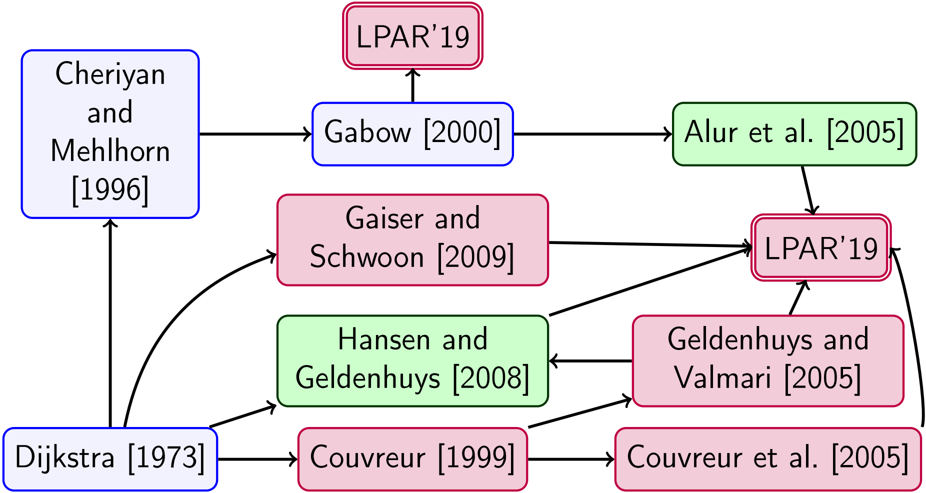

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19


Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
LPAR'19
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

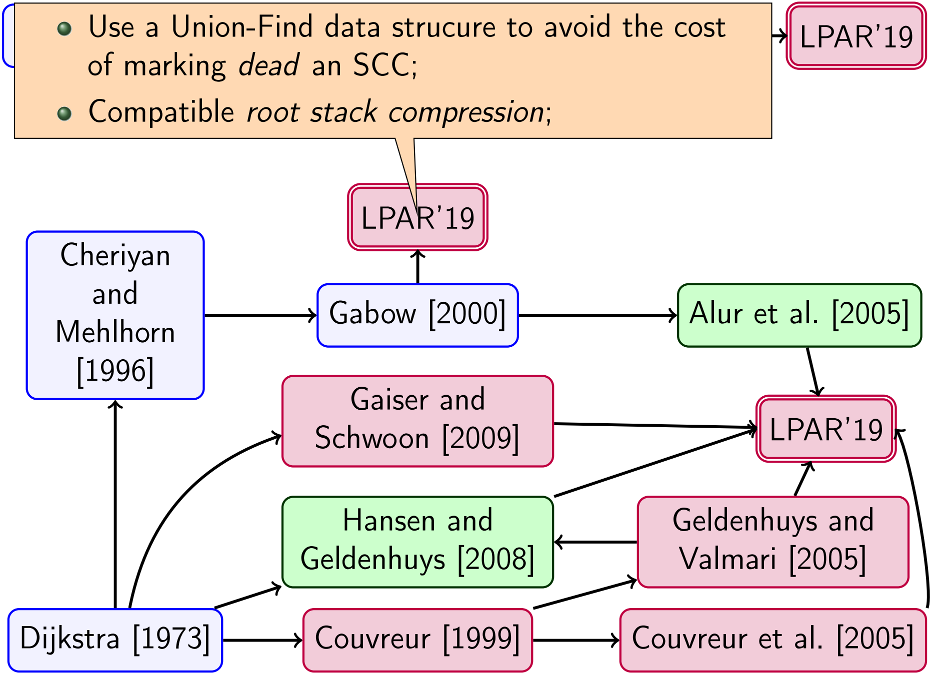
- Rediscovers Cheriyan and Mehlhorn [1996] starting from Tarjan [1972];
- Suggests the use of a Union-Find to perform the SCC computation;



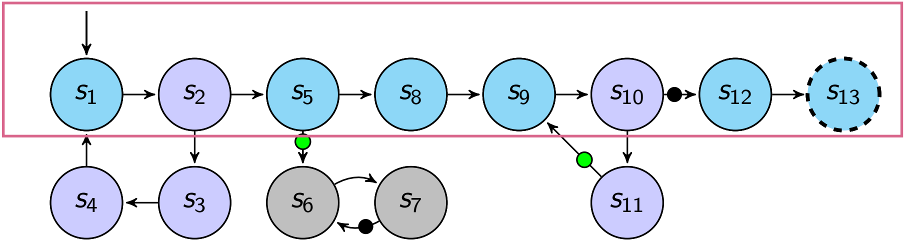

Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19


LPAR'19
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
LPAR'19
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]

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



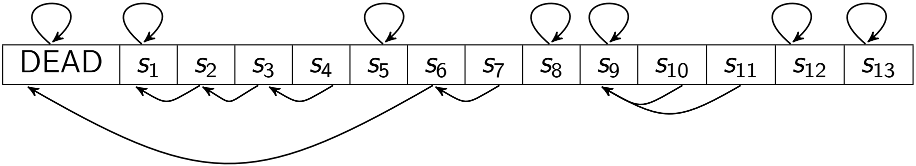
Gabow – Back to the example



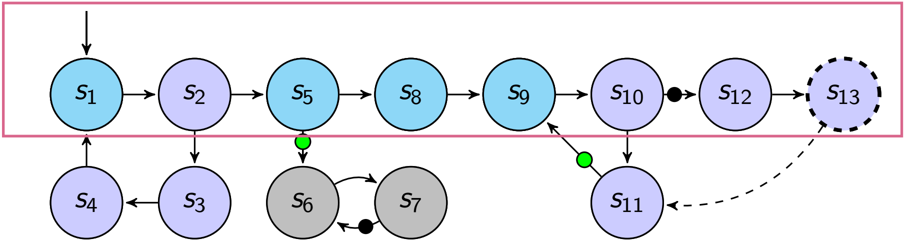
Root stack

s_1	s_5	s_8	s_9	s_{12}	s_{13}
$\emptyset$	$\emptyset$	$\emptyset$	●	$\emptyset$	$\emptyset$

UF



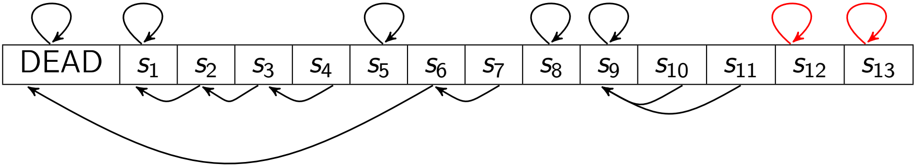
Gabow – Back to the example



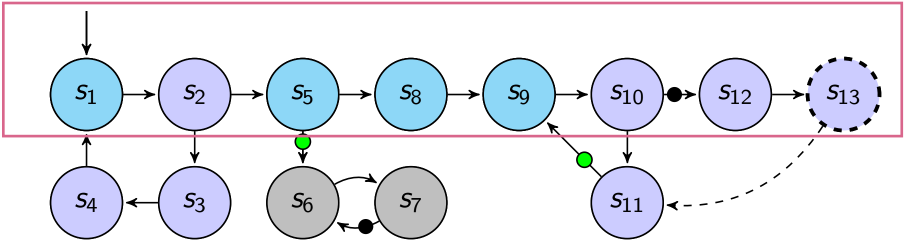
Root stack

s_1	s_5	s_8	s_9	s_{12}	s_{13}
$\emptyset$	$\emptyset$	$\emptyset$	●	$\emptyset$	$\emptyset$

UF



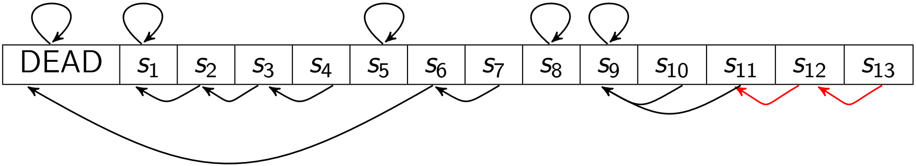
Gabow – Back to the example



Root stack

s_1	s_5	s_8	s_9
$\emptyset$	$\emptyset$	$\emptyset$	

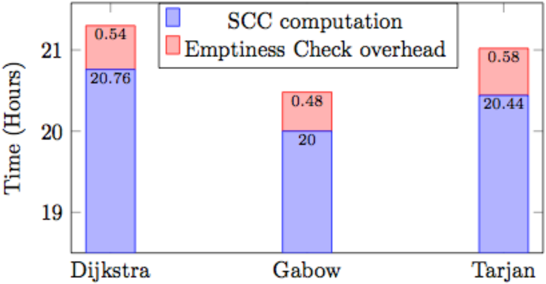
UF



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



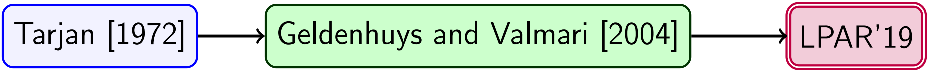
The three algorithms are comparable.

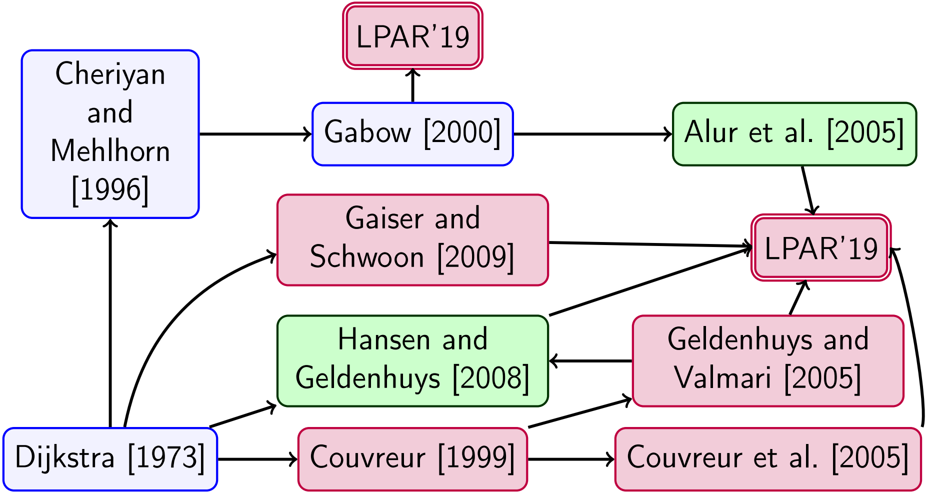
Dijkstra-based emptiness check is the best memory efficient and can benefit from a compressed stack!

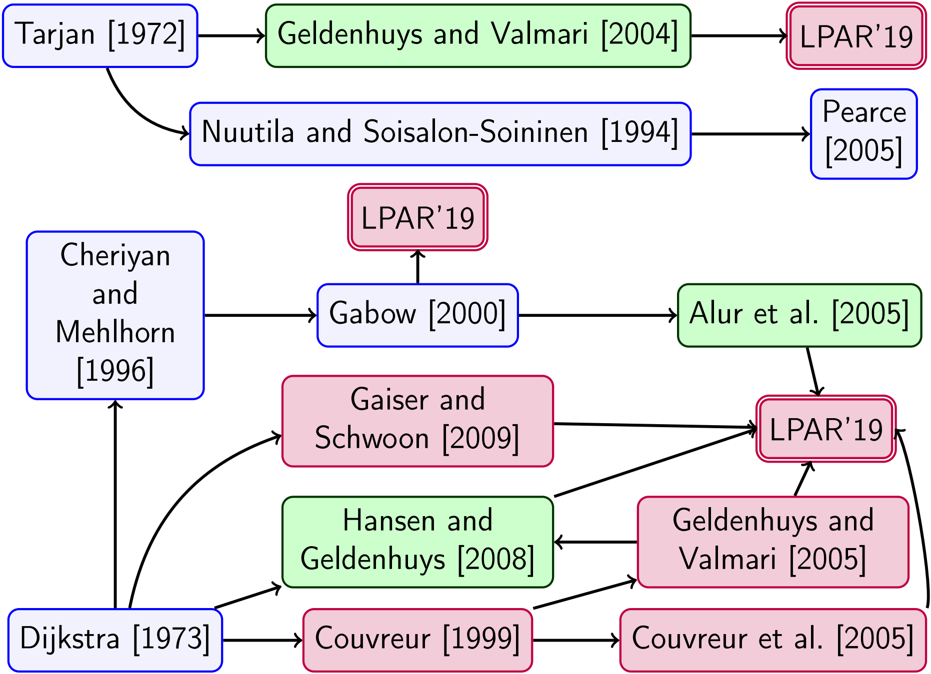
Tarjan-based is the faster when bit state hashing and state space caching are not used!

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]
Geldenhuys and Valmari [2004]
LPAR'19


LPAR'19
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
LPAR'19
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur et al. [2005]


Tarjan [1972]
Geldenhuys and Valmari [2004]
LPAR'19
Nuutila and Soisalon-Soininen [1994]
Pearce [2005]
LPAR'19
Cheriyan and Mehlhorn [1996]
Gabow [2000]
Alur et al. [2005]
Gaiser and Schwoon [2009]
LPAR'19
Hansen and Geldenhuys [2008]
Geldenhuys and Valmari [2005]
Dijkstra [1973]
Couvreur [1999]
Couvreur 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 I

Alur, R., Chaudhuri, S., Etessami, K., and Madhusudan, P. (2005). On-the-fly reachability and cycle detection for recursive state machines. In Halbwachs, N. and Zuck, L., editors, Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), volume 3440 of Lecture Notes in Computer Science, pages 61–76. Springer Berlin Heidelberg.

Cheriyan, J. and Mehlhorn, K. (1996). Algorithms for dense graphs and networks on the random access computer. Algorithmica, 15(6):521–549.

Couvreur, J.-M. (1999). On-the-fly verification of temporal logic. In Wing, J. M., Woodcock, J., and Davies, J., editors, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), volume 1708 of Lecture Notes in Computer Science, pages 253–271, Toulouse, France. Springer-Verlag.

Couvreur, J.-M., Duret-Lutz, A., and Poitrenaud, D. (2005). On-the-fly emptiness checks for generalized Büchi automata. In Godefroid, P., editor, Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05), volume 3639 of Lecture Notes in Computer Science, pages 143–158. Springer.

Dijkstra, E. W. (1973). EWD 376: Finding the maximum strong components in a directed graph. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF`.

Gabow, H. N. (2000). Path-based depth-first search for strong and biconnected components. Information Processing Letters, 74(3-4):107–114.

Bibliography II

Gaiser, A. and Schwoon, S. (2009). Comparison of algorithms for checking emptiness on Büchi automata. In Hlinený, P., Matyás, V., and Vojnar, T., editors, Procedings of Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09), volume 13 of OASICS. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Germany.

Geldenhuys, J. and Valmari, A. (2004). Tarjan's algorithm makes on-the-fly LTL verification more efficient. In Jensen, K. and Podelski, A., editors, Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), volume 2988 of Lecture Notes in Computer Science, pages 205–219. Springer.

Geldenhuys, J. and Valmari, A. (2005). More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoretical Computer Science, 345(1):60–82.

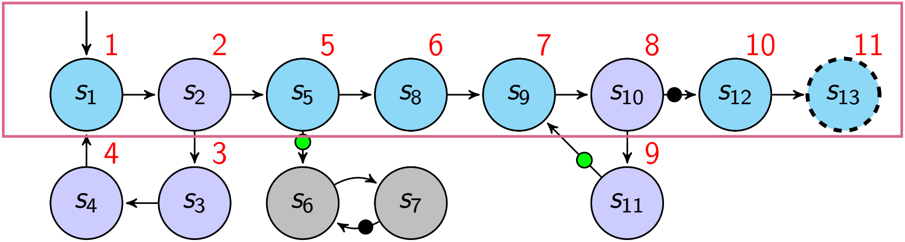
Hansen, H. and Geldenhuys, J. (2008). Cheap and small counterexamples. In Cerone, A. and Gruner, S., editors, Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM'08), pages 53–62. IEEE Computer Society.

Nuutila, E. and Soisalon-Soininen, E. (1994). On finding the strongly connected components in a directed graph. Information Processing Letters, 49(1):9–14.

Pearce, D. J. (2005). An improved algorithm for finding the strongly connected components of a directed graph.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160.

Tarjan – Back to the example



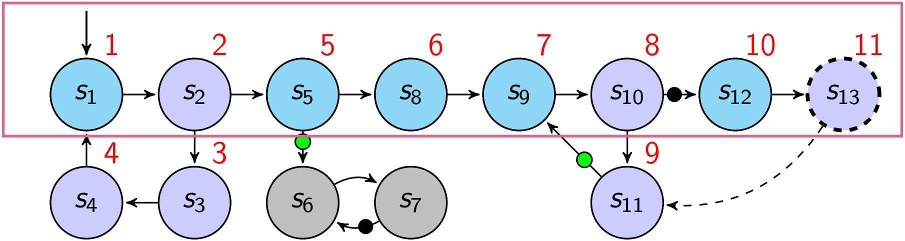
Lowlink stack

1	1	5	6	7	7	10	11
∅	∅	∅	∅	∅	●	∅	∅

LIVE stack

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

Tarjan – Back to the example



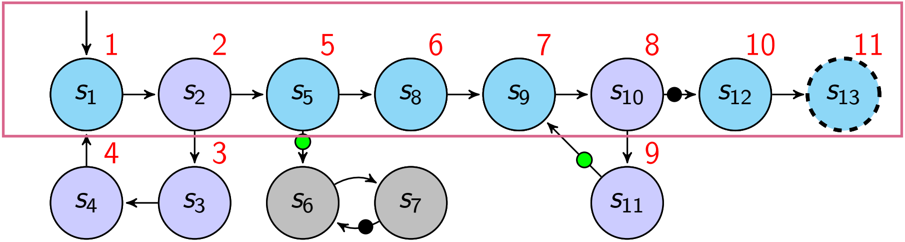
Lowlink stack

1	1	5	6	7	7	10	9
∅	∅	∅	∅	∅	●	∅	∅

LIVE stack

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

Dijkstra – Back to the example



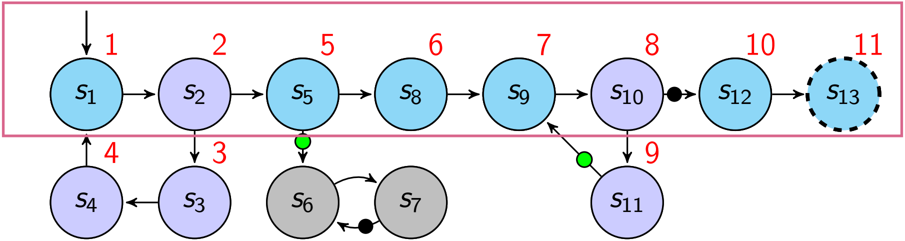
Root stack

s_1	s_5	s_8	s_9	s_{12}	s_{13}
$\emptyset$	$\emptyset$	$\emptyset$	●	$\emptyset$	$\emptyset$

LIVE stack

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

Dijkstra – Back to the example



Root stack

s_1	s_5	s_8	s_9	s_{12}	s_{13}
$\emptyset$	$\emptyset$	$\emptyset$	●	$\emptyset$	$\emptyset$

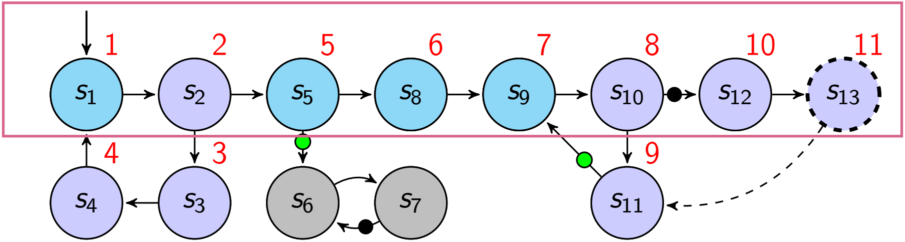
Compressed Root Stack

2	4	6	8
$\emptyset$	$\emptyset$	●	$\emptyset$
×	✓	×	✓

LIVE stack

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

Dijkstra – Back to the example



Root stack

s_1	s_5	s_8	s_9
$\emptyset$	$\emptyset$	$\emptyset$	●●

Compressed Root Stack

2	4	8
$\emptyset$	$\emptyset$	●●
$\times$	✓	$\times$

LIVE stack

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