Beyond Big-Oh analysis in automata theory

Javier Esparza

Foundations of Software Reliability Group
Technische Universität München

A bit of satire . . .

Theoretical computer scientists as classifiers.

A bit of satire . . .

Theoretical computer scientists as classifiers.

Definition

- A theoretical computer scientist (TCS) is a (possibly non-terminating) algorithm that gets a problem P as input and outputs a lower bound $\Omega(LB)$ and an upper bound $O(UB)$.

A bit of satire . . .

Theoretical computer scientists as classifiers.

Definition

- A theoretical computer scientist (TCS) is a (possibly non-terminating) algorithm that gets a problem P as input and outputs a lower bound $\Omega(LB)$ and an upper bound $O(UB)$.
- A TCS is sober if $LB \leq UB$, otherwise is drunk.

A bit of satire . . .

Theoretical computer scientists as classifiers.

Definition

- A theoretical computer scientist (TCS) is a (possibly non-terminating) algorithm that gets a problem P as input and outputs a lower bound $\Omega(LB)$ and an upper bound $O(UB)$.
- A TCS is sober if $LB \leq UB$, otherwise is drunk.
- A TCS is good iff it writes papers that deserve publishing.

A bit of satire . . .

Theoretical computer scientists as classifiers.

Definition

- A theoretical computer scientist (TCS) is a (possibly non-terminating) algorithm that gets a problem P as input and outputs a lower bound $\Omega(LB)$ and an upper bound $O(UB)$.
- A TCS is sober if $LB \leq UB$, otherwise is drunk.
- A TCS is good iff it writes papers that deserve publishing.
- A paper deserves publishing iff it provides new or better bounds.

The classifier's world view

- Once matching upper and lower bounds up to a multiplicative constant have been found, going beyond is tedious and uninteresting.

The classifier's world view

- Once matching upper and lower bounds up to a multiplicative constant have been found, going beyond is tedious and uninteresting.

- Therefore, going beyond Big-Oh analysis is left to another class of computer scientists called

The classifier's world view

- Once matching upper and lower bounds up to a multiplicative constant have been found, going beyond is tedious and uninteresting.

- Therefore, going beyond Big-Oh analysis is left to another class of computer scientists called masochists.

The classifier's world view

- Once matching upper and lower bounds up to a multiplicative constant have been found, going beyond is tedious and uninteresting.

- Therefore, going beyond Big-Oh analysis is left to another class of computer scientists called masochists.

- Implementing algorithms is a mechanical task. It brings a theoretician neither new insights nor "scientific glory".

The classifier's world view

- Once matching upper and lower bounds up to a multiplicative constant have been found, going beyond is tedious and uninteresting.

- Therefore, going beyond Big-Oh analysis is left to another class of computer scientists called masochists.

- Implementing algorithms is a mechanical task. It brings a theoretician neither new insights nor "scientific glory".

- However, implementations are sometimes needed to please reviewers and research councils. Fortunately, they can be left to another class of human beings:

The classifier's world view

- Once matching upper and lower bounds up to a multiplicative constant have been found, going beyond is tedious and uninteresting.

- Therefore, going beyond Big-Oh analysis is left to another class of computer scientists called masochists.

- Implementing algorithms is a mechanical task. It brings a theoretician neither new insights nor "scientific glory".

- However, implementations are sometimes needed to please reviewers and research councils. Fortunately, they can be left to another class of human beings: students.

A few alternative thesis

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.
- Classifications usually help, are but a first step.

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.
- Classifications usually help, are but a first step.
- An efficient algorithm is not the same as an algorithm with $O(f(n))$ runtime for a slowly growing f:

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.
- Classifications usually help, are but a first step.
- An efficient algorithm is not the same as an algorithm with $O(f(n))$ runtime for a slowly growing f:
  - constants may matter,

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.
- Classifications usually help, are but a first step.
- An efficient algorithm is not the same as an algorithm with $O(f(n))$ runtime for a slowly growing f:
  - constants may matter,
  - runtime is not the only important parameter.

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.
- Classifications usually help, are but a first step.
- An efficient algorithm is not the same as an algorithm with $O(f(n))$ runtime for a slowly growing f:
  - constants may matter,
  - runtime is not the only important parameter.
- Implementations very much help to reveal the problems of seemingly efficient algorithms. They lead to better theory.

A few alternative thesis

- Theoretical computer scientists should provide efficient algorithms for problems, not just classify them.
- Classifications usually help, are but a first step.
- An efficient algorithm is not the same as an algorithm with $O(f(n))$ runtime for a slowly growing f:
  - constants may matter,
  - runtime is not the only important parameter.
- Implementations very much help to reveal the problems of seemingly efficient algorithms. They lead to better theory.
- Automata theory for verification very much profits from "beyond Big-Oh" analysis and prototype implementations.

Today's menu

Today's menu

- Appetizer: Universality of finite automata

Today's menu

- Appetizer: Universality of finite automata
- Main course: Emptiness of Büchi automata

Today’s menu

- Appetizer: Universality of finite automata
- Main course: Emptiness of Büchi automata
- Dessert: Universal search

Universality of finite automata

The problem

Given: a NFA A over alphabet Σ.
Decide: is $L(A) = \Sigma^*$?

The problem

Given: a NFA A over alphabet Σ.
Decide: is $L(A) = \Sigma^*$?

Theorem:

Universality is PSPACE-complete.

The problem

Given: a NFA A over alphabet Σ.
Decide: is $L(A) = \Sigma^*$?

Theorem:

Universality is PSPACE-complete.

Deterministic algorithm:

Determinize $\rightarrow$ complement $\rightarrow$ check for emptiness.

The problem

Given: a NFA A over alphabet Σ.
Decide: is $L(A) = \Sigma^*$?

Theorem:

Universality is PSPACE-complete.

Deterministic algorithm:

Determinize $\rightarrow$ complement $\rightarrow$ check for emptiness.

Complexity:

$O(2^{|A|})$ time and space, and $\Theta(2^{|A|})$ for some family.

End of the story? No!

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

- Let $B = Pow(A)$ (only reachable states).

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

- Let $B = Pow(A)$ (only reachable states).
- Recall: $L_B(Q) = \bigcup_{q \in Q} L_A(q)$ for every state Q of B.

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

- Let $B = Pow(A)$ (only reachable states).
- Recall: $L_B(Q) = \bigcup_{q \in Q} L_A(q)$ for every state Q of B.
- Recall: A universal iff $L_B(Q) = \Sigma^*$ for every state Q of B.

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

- Let $B = Pow(A)$ (only reachable states).
- Recall: $L_B(Q) = \bigcup_{q \in Q} L_A(q)$ for every state Q of B.
- Recall: A universal iff $L_B(Q) = \Sigma^*$ for every state Q of B.
- Assume $Q_1 \subseteq Q_2$. We have $L_B(Q_1) \subseteq L_B(Q_2)$ and if B universal then $L_B(Q_1) = L_B(Q_2)$.

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

- Let $B = Pow(A)$ (only reachable states).
- Recall: $L_B(Q) = \bigcup_{q \in Q} L_A(q)$ for every state Q of B.
- Recall: A universal iff $L_B(Q) = \Sigma^*$ for every state Q of B.
- Assume $Q_1 \subseteq Q_2$. We have $L_B(Q_1) \subseteq L_B(Q_2)$ and if B universal then $L_B(Q_1) = L_B(Q_2)$.
- Let B' be the result of the operation. Then $L_{B'} \subseteq L_B$ and if B universal then $L_{B'} = L_B$.

End of the story? No!

Subsumption check [DeWDHR06]:

If the powerset construction generates states $Q_1 \subseteq Q_2$, redirect Q_2's incoming arcs to Q_1 and remove Q_2.

Correctness

- Let $B = Pow(A)$ (only reachable states).
- Recall: $L_B(Q) = \bigcup_{q \in Q} L_A(q)$ for every state Q of B.
- Recall: A universal iff $L_B(Q) = \Sigma^*$ for every state Q of B.
- Assume $Q_1 \subseteq Q_2$. We have $L_B(Q_1) \subseteq L_B(Q_2)$ and if B universal then $L_B(Q_1) = L_B(Q_2)$.
- Let B' be the result of the operation. Then $L_{B'} \subseteq L_B$ and if B universal then $L_{B'} = L_B$.
- So B' universal iff B universal iff A universal.

Potential application to verification

Typical scenario

- System: communicating automata $A_1, A_2, \ldots, A_n$.
- Specification (allowed behaviour): automaton B.
- System's behaviour: automaton $A = A_1 \otimes A_2 \otimes \ldots \otimes A_n$.
- System correct if $L(A) \subseteq L(B)$

Potential application to verification

Typical scenario

- System: communicating automata $A_1, A_2, \ldots, A_n$.
- Specification (allowed behaviour): automaton B.
- System's behaviour: automaton $A = A_1 \otimes A_2 \otimes \ldots \otimes A_n$.
- System correct if $L(A) \subseteq L(B)$

Usual approach: $L(A) \subseteq L(B)$ iff $L(A) \cap \overline{L(B)}) = \emptyset$

- Compute $A = A_1 \otimes \ldots \otimes A_n$. Possible blowup!
- Check emptiness of $A \times \overline{B}$.

Potential application to verification

Typical scenario

- System: communicating automata $A_1, A_2, \ldots, A_n$.
- Specification (allowed behaviour): automaton B.
- System's behaviour: automaton $A = A_1 \otimes A_2 \otimes \ldots \otimes A_n$.
- System correct if $L(A) \subseteq L(B)$

Usual approach: $L(A) \subseteq L(B)$ iff $L(A) \cap \overline{L(B)}) = \emptyset$

- Compute $A = A_1 \otimes \ldots \otimes A_n$. Possible blowup!
- Check emptiness of $A \times \overline{B}$.

Alternative approach: $L(A) \subseteq L(B)$ iff $\overline{L(A)} \cup L(B) = \Sigma^*$

- Compute $\overline{A} = \overline{A}_1 \oplus \ldots \oplus \overline{A}_n$.
- Check universality of $A + \overline{B}$. Possible blowup!

Emptiness of Büchi automata

The problem

Given: a Büchi automaton A.
Decide: is $L(A) = \emptyset$?

Lassos

A is nonempty iff it contains an accepting lasso: a path leading from some initial state to some accepting state, followed by a cycle.

A trivial quadratic algorithm

The algorithm

(1) Compute all reachable final states.

(2) For every final state q:
 - check if q is reachable from itself.
 - if so, stop and answer “nonempty”.
 Answer “empty”.

A trivial quadratic algorithm

The algorithm

(1) Compute all reachable final states.

(2) For every final state q:
 - check if q is reachable from itself.
 - if so, stop and answer “nonempty”.
Answer “empty”.

Complexity

- (1) takes $O(|A|)$ time.

A trivial quadratic algorithm

The algorithm

(1) Compute all reachable final states.

(2) For every final state q:
 - check if q is reachable from itself.
 - if so, stop and answer “nonempty”.
 Answer “empty”.

Complexity

- (1) takes $O(|A|)$ time.
- (2) takes $O(|A|^2)$ time, and there is a family of graphs for which it takes $\Theta(|A|^2)$.

A first linear algorithm: double-DFS [CVWY91]

(1) Use DFS to compute a list $\alpha_1, \alpha_2, \ldots, \alpha_k$ of all reachable accepting states

(2) For $i = 1$ to k:
- use DFS to check if α_i is reachable from itself
- if so, stop and answer "nonempty".

Answer "empty".

A first linear algorithm: double-DFS [CVWY91]

(1) Use DFS to compute a list $\alpha_1, \alpha_2, \ldots, \alpha_k$ of all reachable accepting states sorted in postorder.
(a state is added to list when **backtracking** from it)

(2) For $i = 1$ to k:
- use a modified DFS to check if α_i is reachable from itself without visiting any state reachable from $\alpha_1, \ldots, \alpha_{i-1}$.
- if so, stop and answer "nonempty".

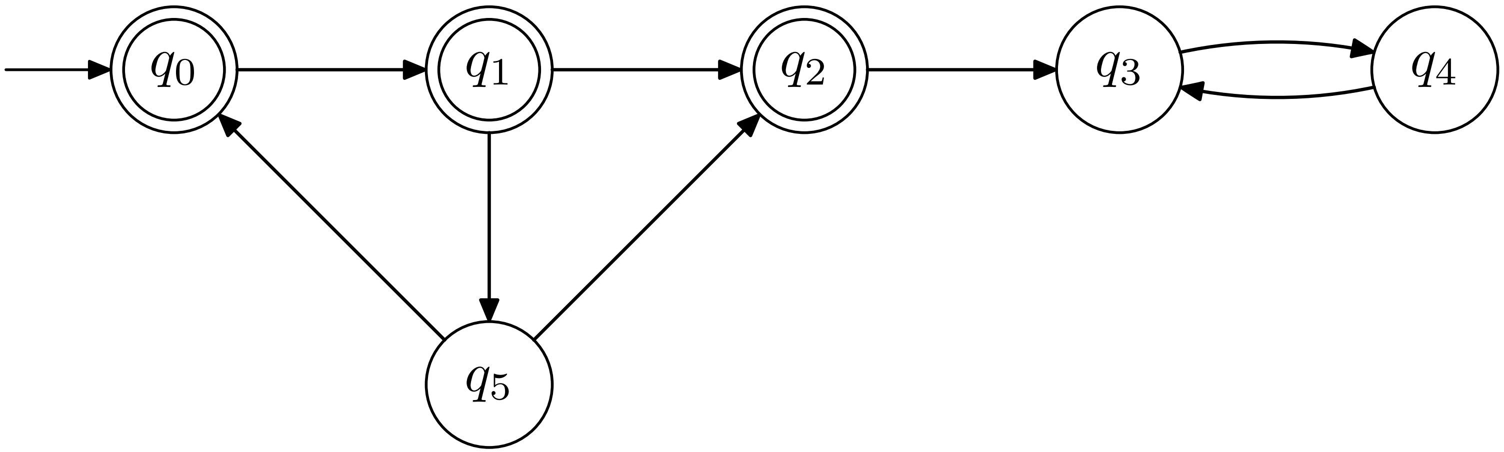
Answer "empty".

A first linear algorithm: double-DFS [CVWY91]

(1) Use DFS to compute a list $\alpha_1, \alpha_2, \ldots, \alpha_k$ of all reachable accepting states sorted in postorder.
(a state is added to list when **backtracking** from it)

(2) For $i = 1$ to k:
- use a modified DFS to check if α_i is reachable from itself without visiting any state reachable from $\alpha_1, \ldots, \alpha_{i-1}$.
- if so, stop and answer "nonempty".

Answer "empty".



Complexity

Time complexity

- Phase (1) takes $O(|A|)$ time.

Complexity

Time complexity

- Phase (1) takes $O(|A|)$ time.
- Phase (2) takes $O(|A|)$ time.
  In the DFS for α_i we backtrack whenever hitting states visited during the former DFSs, and so every transition is explored at most once.

Complexity

Time complexity

- Phase (1) takes $O(|A|)$ time.
- Phase (2) takes $O(|A|)$ time.
  In the DFS for α_i we backtrack whenever hitting states visited during the former DFSs, and so every transition is explored at most once.
- Together: 2 post ops per (reachable) state.

Complexity

Time complexity

- Phase (1) takes $O(|A|)$ time.
- Phase (2) takes $O(|A|)$ time.
  In the DFS for α_i we backtrack whenever hitting states visited during the former DFSs, and so every transition is explored at most once.
- Together: 2 post ops per (reachable) state.

Space complexity

Complexity

Time complexity

- Phase (1) takes $O(|A|)$ time.
- Phase (2) takes $O(|A|)$ time.
  In the DFS for α_i we backtrack whenever hitting states visited during the former DFSs, and so every transition is explored at most once.
- Together: 2 post ops per (reachable) state.

Space complexity

- For each state we have three possible situations:
  - not yet discovered by the first phase;
  - discovered by the first, but not yet by the second;
  - discovered by both phases.

Complexity

Time complexity

- Phase (1) takes $O(|A|)$ time.
- Phase (2) takes $O(|A|)$ time.
  In the DFS for α_i we backtrack whenever hitting states visited during the former DFSs, and so every transition is explored at most once.
- Together: 2 post ops per (reachable) state.

Space complexity

- For each state we have three possible situations:
  - not yet discovered by the first phase;
  - discovered by the first, but not yet by the second;
  - discovered by both phases.
- 2 additional bits per (reachable) state.

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness II

If A is nonempty, then the algorithm answers “nonempty”.

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness II

If A is nonempty, then the algorithm answers “nonempty”.

Proof:

- Consider the case $k = 2$ (two final states α_1, α_2).

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness II

If A is nonempty, then the algorithm answers “nonempty”.

Proof:

- Consider the case $k = 2$ (two final states α_1, α_2).
- If some cycle contains α_1, the algorithm will detect it.

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness II

If A is nonempty, then the algorithm answers “nonempty”.

Proof:

- Consider the case $k = 2$ (two final states α_1, α_2).
- If some cycle contains α_1, the algorithm will detect it.
- If some cycle contains α_2, and no transition of the cycle is reachable from α_1, the algorithm will detect it.

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness II

If A is nonempty, then the algorithm answers “nonempty”.

Proof:

- Consider the case $k = 2$ (two final states α_1, α_2).
- If some cycle contains α_1, the algorithm will detect it.
- If some cycle contains α_2, and no transition of the cycle is reachable from α_1, the algorithm will detect it.
- Potential problem: some cycle contains α_2, some transition of the cycle is reachable from α_1.

Correctness

Correctness I

If the algorithm answers “nonempty”, then A is nonempty. Easy.

Correctness II

If A is nonempty, then the algorithm answers “nonempty”.

Proof:

- Consider the case $k = 2$ (two final states α_1, α_2).
- If some cycle contains α_1, the algorithm will detect it.
- If some cycle contains α_2, and no transition of the cycle is reachable from α_1, the algorithm will detect it.
- Potential problem: some cycle contains α_2, some transition of the cycle is reachable from α_1.
- Call these cycles blocked.

- Solution: guarantee that if there are blocked cycles, then some cycle contains α_1.
  Because cycles containing α_1 are always detected!

- Solution: guarantee that if there are blocked cycles, then some cycle contains α_1.
  Because cycles containing α_1 are always detected!

- If there is a blocked cycle, then $\alpha_1 \rightsquigarrow \alpha_2$.
- If $(\alpha_1 \rightsquigarrow \alpha_2 \wedge \alpha_2 \rightsquigarrow \alpha_1)$ then some cycle contains α_1.
- So it suffices to guarantee: if $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \rightsquigarrow \alpha_1$.
- We show that postorder implies this.

- Solution: guarantee that if there are blocked cycles, then some cycle contains α_1.
  Because cycles containing α_1 are always detected!

- If there is a blocked cycle, then $\alpha_1 \rightsquigarrow \alpha_2$.
- If $(\alpha_1 \rightsquigarrow \alpha_2 \wedge \alpha_2 \rightsquigarrow \alpha_1)$ then some cycle contains α_1.
- So it suffices to guarantee: if $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \rightsquigarrow \alpha_1$.
- We show that postorder implies this.

- Look at DFS as a recursive procedure *dfs*(q).
- Let *ca*(q) denote the time at which *dfs*(q) is called.
- Let *ret*(q) denote the time at which *dfs*(q) returns.
  (The search backtracks from q.)

- Solution: guarantee that if there are blocked cycles, then some cycle contains α_1.
  Because cycles containing α_1 are always detected!

- If there is a blocked cycle, then $\alpha_1 \rightsquigarrow \alpha_2$.
- If $(\alpha_1 \rightsquigarrow \alpha_2 \wedge \alpha_2 \rightsquigarrow \alpha_1)$ then some cycle contains α_1.
- So it suffices to guarantee: if $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \rightsquigarrow \alpha_1$.
- We show that postorder implies this.

- Look at DFS as a recursive procedure *dfs*(q).
- Let *ca*(q) denote the time at which *dfs*(q) is called.
- Let *ret*(q) denote the time at which *dfs*(q) returns.
  (The search backtracks from q.)
- Postorder assumption: *ret*$(\alpha_1) <$ *ret*(α_2).

Lemma

Assume $\mathit{ret}(\alpha_1) < \mathit{ret}(\alpha_2)$. If $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \rightsquigarrow \alpha_1$.

Lemma

Assume $\mathit{ret}(\alpha_1) < \mathit{ret}(\alpha_2)$. If $\alpha_1 \leadsto \alpha_2$ then $\alpha_2 \leadsto \alpha_1$.

Proof:

- By proper nesting of calls we have either:
  - $\mathit{ca}(\alpha_1) < \mathit{ret}(\alpha_1) < \mathit{ca}(\alpha_2) < \mathit{ret}(\alpha_2)$ or
  - $\mathit{ca}(\alpha_2) < \mathit{ca}(\alpha_1) < \mathit{ret}(\alpha_1) < \mathit{ret}(\alpha_2)$

Lemma

Assume $ret(\alpha_1) < ret(\alpha_2)$. If $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \rightsquigarrow \alpha_1$.

Proof:

- By proper nesting of calls we have either:
  - $ca(\alpha_1) < ret(\alpha_1) < ca(\alpha_2) < ret(\alpha_2)$ or
  - $ca(\alpha_2) < ca(\alpha_1) < ret(\alpha_1) < ret(\alpha_2)$
- Case 1: $ca(\alpha_1) < ret(\alpha_1) < ca(\alpha_2) < ret(\alpha_2)$.
  Then $\alpha_1 \not\rightsquigarrow \alpha_2$.

Lemma

Assume $ret(\alpha_1) < ret(\alpha_2)$. If $\alpha_1 \rightsquigarrow \alpha_2$ then $\alpha_2 \rightsquigarrow \alpha_1$.

Proof:

- By proper nesting of calls we have either:
  - $ca(\alpha_1) < ret(\alpha_1) < ca(\alpha_2) < ret(\alpha_2)$ or
  - $ca(\alpha_2) < ca(\alpha_1) < ret(\alpha_1) < ret(\alpha_2)$
- Case 1: $ca(\alpha_1) < ret(\alpha_1) < ca(\alpha_2) < ret(\alpha_2)$.
  Then $\alpha_1 \not\rightsquigarrow \alpha_2$.
- Case 2: $ca(\alpha_2) < ca(\alpha_1) < ret(\alpha_1) < ret(\alpha_2)$.
  Then $\alpha_2 \rightsquigarrow \alpha_1$.

End of the story? No!

End of the story? No!

- Double-DFS requires to explore every transition at least once.
  (Cannot terminate before the end of the first search!)

End of the story? No!

- Double-DFS requires to explore every transition at least once.
  (Cannot terminate before the end of the first search!)
- Double-DFS inadequate for producing counterexamples:

End of the story? No!

- Double-DFS requires to explore every transition at least once.
  (Cannot terminate before the end of the first search!)
- Double-DFS inadequate for producing counterexamples:
  Counterexample: path to accepting state α_i + cycle.

End of the story? No!

- Double-DFS requires to explore every transition at least once.
  (Cannot terminate before the end of the first search!)
- Double-DFS inadequate for producing counterexamples:
  Counterexample: path to accepting state α_i + cycle.
  Double-DFS requires to store paths for all accepting states.

Solution: nested-DFS [CVWY91]

- Interleave the two phases.

Solution: nested-DFS [CVWY91]

- Interleave the two phases.
- At time *ret*(α_i) interrupt the first search and launch the second search for α_i.

Solution: nested-DFS [CVWY91]

- Interleave the two phases.
- At time *ret*(α_i) interrupt the first search and launch the second search for α_i.
- When the algorithm finds a cycle the call stack contains
  - a path to the current final state α_i, plus
  - a path leading from α_i to itself.

Solution: nested-DFS [CVWY91]

- Interleave the two phases.
- At time *ret*(α_i) interrupt the first search and launch the second search for α_i.
- When the algorithm finds a cycle the call stack contains
  - a path to the current final state α_i, plus
  - a path leading from α_i to itself.
- Counterexample: just pop the call stack!

Solution: nested-DFS [CVWY91]

- Interleave the two phases.
- At time *ret*(α_i) interrupt the first search and launch the second search for α_i.
- When the algorithm finds a cycle the call stack contains
  - a path to the current final state α_i, plus
  - a path leading from α_i to itself.
- Counterexample: just pop the call stack!
- Correctness: Easy. The second searches are exactly as in the double-DFS algorithm.

End of the story? No!

End of the story? No!

Definition

A search algorithm for Büchi emptiness is optimal if it terminates immediately after the set of transitions it has explored contains an accepting lasso.

End of the story? No!

Definition

A search algorithm for Büchi emptiness is optimal if it terminates immediately after the set of transitions it has explored contains an accepting lasso.

The nested-DFS algorithm is not optimal!

Minor improvements

[Holzmann, Peled, Yannakakis 96]

If the second search finds a state that is currently in the call stack of the first search, answer “nonempty”.

Minor improvements

[Holzmann, Peled, Yannakakis 96]

If the second search finds a state that is currently in the call stack of the first search, answer “nonempty”.

[Gastin, Moro, Zeitoun 04]

If the first search finds an accepting state that is currently in the call stack, answer “nonempty”.

Minor improvements

[Holzmann, Peled, Yannakakis 96]

If the second search finds a state that is currently in the call stack of the first search, answer "nonempty".

[Gastin, Moro, Zeitoun 04]

If the first search finds an accepting state that is currently in the call stack, answer "nonempty".

[Schwoon, E. 05]

These two improvements still require only 2 additional bits per state: four-colour algorithm.

But: the four-colour algorithm is still not optimal.

But: the four-colour algorithm is still not optimal.

Question

Are there optimal (linear-time) algorithms?

SCC-based algorithms

Approach

- Identify the reachable (nontrivial) SCCs of A.
- Check if some of them contains an accepting state.

Tarjan's algorithm for computing SCCs

Basic notions

- Automaton $A \Rightarrow$ dag of SCCs.

Tarjan's algorithm for computing SCCs

Basic notions

- Automaton $A \Rightarrow$ dag of SCCs.
- Root of a SCC: the first node of the SCC discovered by the DFS.
  (The definition of root refers to a particular, fixed DFS-run!)

Tarjan's algorithm for computing SCCs

Basic notions

- Automaton $A \Rightarrow$ dag of SCCs.
- Root of a SCC: the first node of the SCC discovered by the DFS.
  (The definition of root refers to a particular, fixed DFS-run!)
- If ρ is a root, then at time $ret(\rho)$ the DFS has discovered all nodes of ρ's SCC and its descendants in the dag.

Tarjan's algorithm for computing SCCs

Basic notions

- Automaton $A \Rightarrow$ dag of SCCs.
- Root of a SCC: the first node of the SCC discovered by the DFS.
  (The definition of root refers to a particular, fixed DFS-run!)
- If ρ is a root, then at time $ret(\rho)$ the DFS has discovered all nodes of ρ's SCC and its descendants in the dag.

First idea

- Push all discovered nodes in a new stack (Tarjan's stack).
- For every root ρ: at time $ret(\rho)$, pop from Tarjan's stack until ρ is popped; the popped nodes constitute ρ's SCC.

Tarjan and GOD's algorithm

GOD's contribution: Oracle

For a given state q oracle decides if q is a root.

T(q)
    push(q, $Stack$);
    for each transition $q \to r$
        if r not yet explored then T(r)
    if q is a root then
        repeat $s := pop(Stack)$ until $s = q$

Implementing the oracle

Problem

The algorithm must identify the roots of the SCCs.
But the SCCs are what we want to compute!

Implementing the oracle

Problem

The algorithm must identify the roots of the SCCs.
But the SCCs are what we want to compute!

Second idea

- Annotate each state q with $ca(q)$ and a lowlink-number $lowlink(q)$.
  (Order induced by call numbers is all that matters)

Implementing the oracle

Problem

The algorithm must identify the roots of the SCCs.
But the SCCs are what we want to compute!

Second idea

- Annotate each state q with $ca(q)$ and a lowlink-number $lowlink(q)$.
  (Order induced by call numbers is all that matters)
- $lowlink(q)$: lowest $ca(r)$ of states r satisfying
  - q and r lie in the same SCC, and
  - r reachable from q through states not yet discovered at time $ca(q)$.

Implementing the oracle

Problem

The algorithm must identify the roots of the SCCs.
But the SCCs are what we want to compute!

Second idea

- Annotate each state q with $ca(q)$ and a lowlink-number $lowlink(q)$.
  (Order induced by call numbers is all that matters)
- $lowlink(q)$: lowest $ca(r)$ of states r satisfying
  - q and r lie in the same SCC, and
  - r reachable from q through states not yet discovered at time $ca(q)$.
- $lowlink(q) \leq ca(q)$ for every state q.

Implementing the oracle

Problem

The algorithm must identify the roots of the SCCs.
But the SCCs are what we want to compute!

Second idea

- Annotate each state q with $ca(q)$ and a lowlink-number $lowlink(q)$.
  (Order induced by call numbers is all that matters)
- $lowlink(q)$: lowest $ca(r)$ of states r satisfying
  - q and r lie in the same SCC, and
  - r reachable from q through states not yet discovered at time $ca(q)$.
- $lowlink(q) \leq ca(q)$ for every state q.
- Fact: $lowlink(q) = ca(q)$ if and only if q is a root.

Tarjan's algorithm

Miracle

lowlink(q) can be easily determined at time *ret*(q).

Tarjan's algorithm

Miracle

lowlink(q) can be easily determined at time *ret*(q).

```
T(q)
    push(q, Stack);
    for each transition q → r
        if r not yet explored then
            T(r);
            r.lowlink := min(q.lowlink, r.lowlink)
        else if r ∈ Stack then
            r.lowlink := min(q.lowlink, r.ca)
    if q.lowlink = q.ca then
        repeat s := pop(Stack) until s = q
```

Geldenhuys and Valmari's algorithm [GV04]

- A direct modification of Tarjan's algorithm for emptiness checking is non-optimal.

Geldenhuys and Valmari's algorithm [GV04]

- A direct modification of Tarjan's algorithm for emptiness checking is non-optimal.
- Requires to completely explore an SCC before it is popped from the stack.

Geldenhuys and Valmari's algorithm [GV04]

- A direct modification of Tarjan's algorithm for emptiness checking is non-optimal.
- Requires to completely explore an SCC before it is popped from the stack.

Main observation of [GV04]:

α belongs to a cycle iff $T(\alpha)$ reaches some state r satisfying two conditions:

- $r \in \mathit{Stack}$, and
- $\mathit{lowlink}(r) < \mathit{ca}(\alpha)$.

Geldenhuys and Valmari's algorithm [GV04]

Add a new parameter to the procedure to keep track of the last visited accepting state.

```
GV(q, α)
    push(q, Stack);
    for each transition q → r
        if r not yet explored then
            if r accepting then GV(r, r) else GV(r, α);
            r.lowlink := min(q.lowlink, r.lowlink)
        else if r ∈ Stack then
            if r.lowlink < α.ca then report "nonempty";
            r.lowlink := min(q.lowlink, r.ca)
    if q.lowlink = q.ca then
        repeat s := pop(Stack) until s = q
```

End of the story? No!

End of the story? No!

Time complexity

[GV04] requires only one post op per state.

End of the story? No!

Time complexity

[GV04] requires only one post op per state.

Space complexity

- [GV04] requires to store two numbers per state plus a third number for Tarjan's stack ($3 \cdot \log n$ bits per state).

End of the story? No!

Time complexity

[GV04] requires only one post op per state.

Space complexity

- [GV04] requires to store two numbers per state plus a third number for Tarjan's stack ($3 \cdot \log n$ bits per state).
- Compare with 2 bits per state of nested-DFS or the four-colour algorithm.

End of the story? No!

Time complexity

[GV04] requires only one post op per state.

Space complexity

- [GV04] requires to store two numbers per state plus a third number for Tarjan's stack ($3 \cdot \log n$ bits per state).
- Compare with 2 bits per state of nested-DFS or the four-colour algorithm.

Generalized Büchi automata

- LTL $\rightarrow$ Büchi translations yield generalized BA.

End of the story? No!

Time complexity

[GV04] requires only one post op per state.

Space complexity

- [GV04] requires to store two numbers per state plus a third number for Tarjan's stack ($3 \cdot \log n$ bits per state).
- Compare with 2 bits per state of nested-DFS or the four-colour algorithm.

Generalized Büchi automata

- LTL $\rightarrow$ Büchi translations yield generalized BA.
- GBA with n states and k acceptings sets $\rightarrow$ BA with $n \cdot k$ states. Expensive!

End of the story? No!

Time complexity

[GV04] requires only one post op per state.

Space complexity

- [GV04] requires to store two numbers per state plus a third number for Tarjan's stack ($3 \cdot \log n$ bits per state).
- Compare with 2 bits per state of nested-DFS or the four-colour algorithm.

Generalized Büchi automata

- LTL $\rightarrow$ Büchi translations yield generalized BA.
- GBA with n states and k acceptings sets $\rightarrow$ BA with $n \cdot k$ states. Expensive!
- Neither nested-DFS nor GV can be extended to GBA.

Question

Do optimal algorithms exist that

Question

Do optimal algorithms exist that

- require less memory, and

Question

Do optimal algorithms exist that

- require less memory, and
- can be easily extended to GBAs?

Couvreur and Gabow's algorithm [C99,G00]

First idea

Partition Stack into Roots and Nonroots, keeping the following invariant:

Couvreur and Gabow's algorithm [C99,G00]

First idea

Partition Stack into Roots and Nonroots, keeping the following invariant:

- Roots contains all nodes that are roots of the part of the graph explored so far .
- Nonroots: contains all nodes that are non-roots of the part of the graph explored so far .

Couvreur and Gabow's algorithm [C99,G00]

First idea

Partition Stack into Roots and Nonroots, keeping the following invariant:

- Roots contains all nodes that are roots of the part of the graph explored so far .
- Nonroots: contains all nodes that are non-roots of the part of the graph explored so far .

- Key insight: q is a root iff it is a root of the part of the graph explored at time $ret(q)$.

Couvreur and Gabow's algorithm [C99,G00]

First idea

Partition Stack into Roots and Nonroots, keeping the following invariant:

- Roots contains all nodes that are roots of the part of the graph explored so far .
- Nonroots: contains all nodes that are non-roots of the part of the graph explored so far .

- Key insight: q is a root iff it is a root of the part of the graph explored at time $ret(q)$.
- So we can check if q is a root by checking $q = top(Roots)$ at time $ret(q)$.

Couvreur and Gabow's algorithm [C99,G00]

First idea

Partition Stack into Roots and Nonroots, keeping the following invariant:

- Roots contains all nodes that are roots of the part of the graph explored so far .
- Nonroots: contains all nodes that are non-roots of the part of the graph explored so far .

- Key insight: q is a root iff it is a root of the part of the graph explored at time $ret(q)$.
- So we can check if q is a root by checking $q = top(Roots)$ at time $ret(q)$.
- New problem: to keep the invariant.

Couvreur, Gabow, and GOD's algorithm

GOD's contribution: oracle to keep the invariant

- For $q \rightarrow r$, the oracle decides if q reachable from r: $r \rightsquigarrow q$.

Couvreur, Gabow, and GOD's algorithm

GOD's contribution: oracle to keep the invariant

- For $q \rightarrow r$, the oracle decides if q reachable from r: $r \rightsquigarrow q$.

Couvreur, Gabow, and GOD's algorithm

GOD's contribution: oracle to keep the invariant

- For $q \longrightarrow r$, the oracle decides if q reachable from r: $r \rightsquigarrow q$.
- Observe: if $r \rightsquigarrow q$ then r belongs to a cycle.

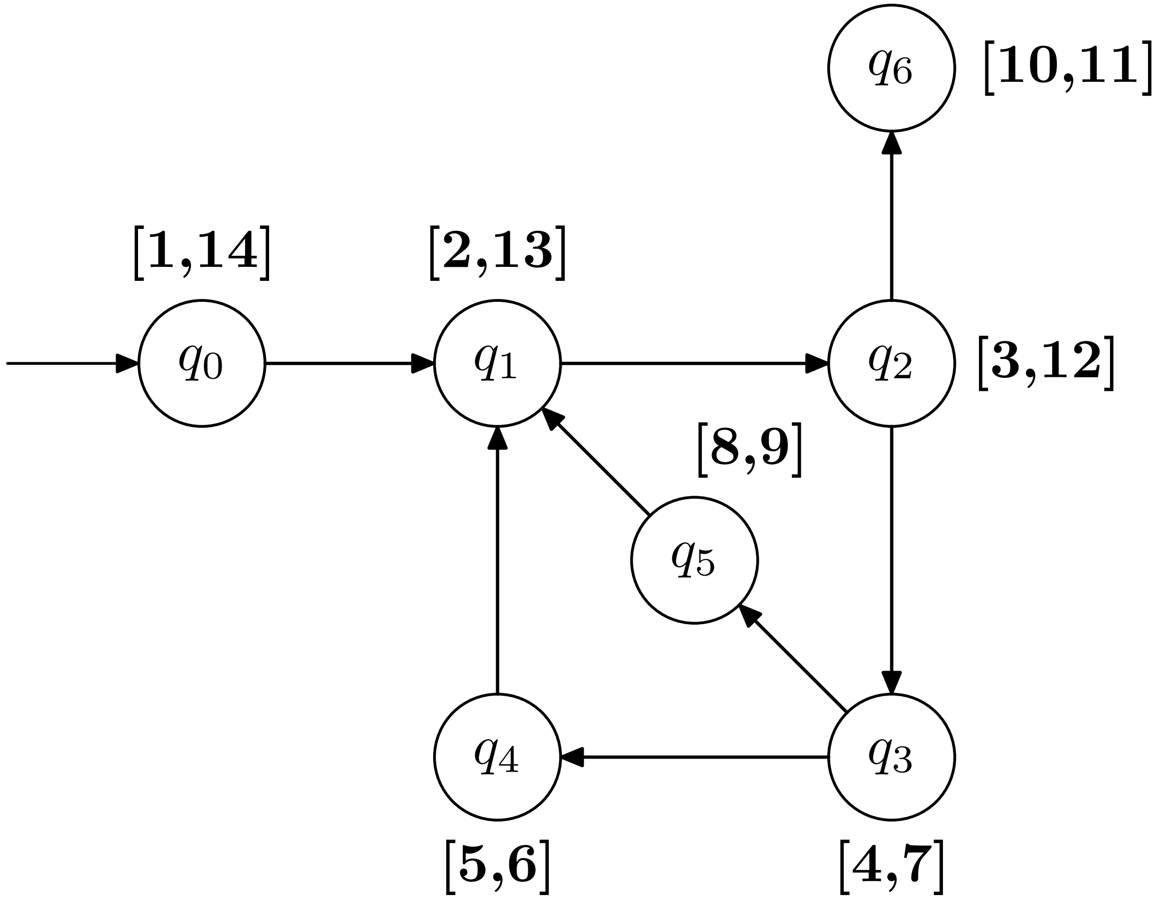
Couvreur, Gabow, and GOD's algorithm

GOD's contribution: oracle to keep the invariant

- For $q \rightarrow r$, the oracle decides if q reachable from r: $r \rightsquigarrow q$.
- Observe: if $r \rightsquigarrow q$ then r belongs to a cycle.
- We show: no node in Roots discovered after r can be a root.

```
GCG(q)
    push(q, Roots);
    for each transition q → r
        if r not yet explored then GCG(r)
        elseif r ⇝ q then
            repeat
                s :=pop(Roots); push(Nonroots);
                if s is accepting report "nonempty"
            until ca(s) ≤ ca(r);
            push(s, Roots); pop(Nonroots)
    if top(Roots) = q then
        pop(Roots);
        while ca(top(Nonroots)) > ca(q)
            pop(Nonroots)
```

Example



Time	Stack content
5	$q_4\, q_3\, q_2\, q_1\, q_0$
6	$q_1\, q_0$
8	$q_5\, q_1\, q_0$
9	$q_1\, q_0$
10	$q_6\, q_1\, q_0$
12	$q_1\, q_0$
14	ϵ

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \rightarrow r \rightsquigarrow q$, $s \in \mathit{Roots}$, $\mathit{ca}(s) > \mathit{ca}(r)$.

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \to r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \to r \rightsquigarrow \rho_r$.

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \rightarrow r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \rightarrow r \rightsquigarrow \rho_r$.
- s is a DFS-ascendant of q, and so $s \rightsquigarrow q$.

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \rightarrow r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \rightarrow r \rightsquigarrow \rho_r$.
- s is a DFS-ascendant of q, and so $s \rightsquigarrow q$.
  Because $s \in Roots$, and $Roots$ subset of DFS-stack.

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \rightarrow r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \rightarrow r \rightsquigarrow \rho_r$.
- s is a DFS-ascendant of q, and so $s \rightsquigarrow q$.
  Because $s \in Roots$, and $Roots$ subset of DFS-stack.
- ρ_r is a DFS-ascendant of s, and so $\rho_r \rightsquigarrow s$.

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \rightarrow r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \rightarrow r \rightsquigarrow \rho_r$.
- s is a DFS-ascendant of q, and so $s \rightsquigarrow q$.
  Because $s \in Roots$, and $Roots$ subset of DFS-stack.
- ρ_r is a DFS-ascendant of s, and so $\rho_r \rightsquigarrow s$.
  Since $q \rightarrow r \rightsquigarrow q$, we have $\rho_r = \rho_q$, and so ρ_r is a DFS-ascendant of q.

Correctness and optimality

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \to r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \to r \rightsquigarrow \rho_r$.
- s is a DFS-ascendant of q, and so $s \rightsquigarrow q$.
  Because $s \in Roots$, and $Roots$ subset of DFS-stack.
- ρ_r is a DFS-ascendant of s, and so $\rho_r \rightsquigarrow s$.
  Since $q \to r \rightsquigarrow q$, we have $\rho_r = \rho_q$, and so ρ_r is a DFS-ascendant of q.
  So either ρ_r is DFS-ascendant of s or vice versa.

Correctness I

If s is popped at line 7, then it belongs to a cycle containing r.

Proof:

- Situation: $q \rightarrow r \rightsquigarrow q$, $s \in Roots$, $ca(s) > ca(r)$.
- We show $\rho_r \rightsquigarrow s \rightsquigarrow q \rightarrow r \rightsquigarrow \rho_r$.
- s is a DFS-ascendant of q, and so $s \rightsquigarrow q$.
  Because $s \in Roots$, and $Roots$ subset of DFS-stack.
- ρ_r is a DFS-ascendant of s, and so $\rho_r \rightsquigarrow s$.
  Since $q \rightarrow r \rightsquigarrow q$, we have $\rho_r = \rho_q$, and so ρ_r is a DFS-ascendant of q.
  So either ρ_r is DFS-ascendant of s or vice versa.
  But s cannot be a DFS-ascendant of ρ_r because $ca(\rho_r) \leq ca(r) < ca(s)$.

Correctness and optimality

Correctness II

If a state s is popped at line 7 and $ca(s) > ca(r)$, then it is not a root.

Correctness and optimality

Correctness II

If a state s is popped at line 7 and $ca(s) > ca(r)$, then it is not a root.

Proof:

- s belongs to a cycle containing r, and, since $ca(s) > ca(r)$, it is not a root.

Correctness and optimality

Correctness III + Optimality

Every reachable state q belonging to some cycle is eventually popped at line 7.
Moreover, q is popped immediately after any cycle containing it is completely explored.

Correctness and optimality

Correctness III + Optimality

Every reachable state q belonging to some cycle is eventually popped at line 7.
Moreover, q is popped immediately after any cycle containing it is completely explored.

Proof:

- Fix a cycle C containing q.

Correctness and optimality

Correctness III + Optimality

Every reachable state q belonging to some cycle is eventually popped at line 7.
Moreover, q is popped immediately after any cycle containing it is completely explored.

Proof:

- Fix a cycle C containing q.
- Let r be the last successor of q along C such that at time $ca(q)$ there is a path of unexplored states from q to r (count q as unexplored, possibly $q = r$).

Correctness III + Optimality

Every reachable state q belonging to some cycle is eventually popped at line 7.
Moreover, q is popped immediately after any cycle containing it is completely explored.

Proof:

- Fix a cycle C containing q.
- Let r be the last successor of q along C such that at time $ca(q)$ there is a path of unexplored states from q to r (count q as unexplored, possibly $q = r$).
- Let s be the successor of r along C.

Correctness and optimality

Correctness III + Optimality

Every reachable state q belonging to some cycle is eventually popped at line 7.
Moreover, q is popped immediately after any cycle containing it is completely explored.

Proof:

- Fix a cycle C containing q.
- Let r be the last successor of q along C such that at time $ca(q)$ there is a path of unexplored states from q to r (count q as unexplored, possibly $q = r$).
- Let s be the successor of r along C.
- $ca(s) \leq ca(q) \leq ca(r)$, and so $ca(s) \leq ca(r)$.

Correctness and optimality

Correctness III + Optimality

Every reachable state q belonging to some cycle is eventually popped at line 7.
Moreover, q is popped immediately after any cycle containing it is completely explored.

Proof:

- Fix a cycle C containing q.
- Let r be the last successor of q along C such that at time $ca(q)$ there is a path of unexplored states from q to r (count q as unexplored, possibly $q = r$).
- Let s be the successor of r along C.
- $ca(s) \leq ca(q) \leq ca(r)$, and so $ca(s) \leq ca(r)$.
- So q is popped at line 7 when $q \rightarrow r$ is explored, or earlier.

Correctness and optimality

Correctness III

Every state discovered by the search and not belonging to any cycle is eventually popped at line 12.

Proof:

Easy.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is "yes" iff $t < ret(\rho_r)$.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is "yes" iff $t < ret(\rho_r)$.

Proof:

- Situation: $ca(q) \leq t < ret(q)$, $q \to r$, $ca(r) \leq t$.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is “yes” iff $t < ret(\rho_r)$.

Proof:

- Situation: $ca(q) \leq t < ret(q)$, $q \rightarrow r$, $ca(r) \leq t$.
- Assume $r \rightsquigarrow q$. If $t \geq ret(\rho_r)$, then $t \geq ret(q)$, contradiction. So $t < ret(\rho_r)$

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is "yes" iff $t < ret(\rho_r)$.

Proof:

- Situation: $ca(q) \leq t < ret(q)$, $q \rightarrow r$, $ca(r) \leq t$.
- Assume $r \rightsquigarrow q$. If $t \geq ret(\rho_r)$, then $t \geq ret(q)$, contradiction. So $t < ret(\rho_r)$
- Assume $r \not\rightsquigarrow q$. Then $q \rightsquigarrow \rho_r \not\rightsquigarrow q$.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is “yes” iff $t < ret(\rho_r)$.

Proof:

- Situation: $ca(q) \leq t < ret(q)$, $q \rightarrow r$, $ca(r) \leq t$.
- Assume $r \rightsquigarrow q$. If $t \geq ret(\rho_r)$, then $t \geq ret(q)$, contradiction. So $t < ret(\rho_r)$
- Assume $r \not\rightsquigarrow q$. Then $q \rightsquigarrow \rho_r \not\rightsquigarrow q$.
  By postorder lemma, $ret(\rho_r) < ret(q)$.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is “yes” iff $t < ret(\rho_r)$.

Proof:

- Situation: $ca(q) \leq t < ret(q)$, $q \rightarrow r$, $ca(r) \leq t$.
- Assume $r \rightsquigarrow q$. If $t \geq ret(\rho_r)$, then $t \geq ret(q)$, contradiction. So $t < ret(\rho_r)$
- Assume $r \not\rightsquigarrow q$. Then $q \rightsquigarrow \rho_r \not\rightsquigarrow q$.
  By postorder lemma, $ret(\rho_r) < ret(q)$.
  Case 1: $ca(\rho_r) < ret(\rho_r) < ca(q) \leq t < ret(q)$. Done.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is “yes” iff $t < ret(\rho_r)$.

Proof:

- Situation: $ca(q) \leq t < ret(q)$, $q \rightarrow r$, $ca(r) \leq t$.
- Assume $r \rightsquigarrow q$. If $t \geq ret(\rho_r)$, then $t \geq ret(q)$, contradiction. So $t < ret(\rho_r)$
- Assume $r \not\rightsquigarrow q$. Then $q \rightsquigarrow \rho_r \not\rightsquigarrow q$.
  By postorder lemma, $ret(\rho_r) < ret(q)$.
  Case 1: $ca(\rho_r) < ret(\rho_r) < ca(q) \leq t < ret(q)$. Done.
  Case 2: $ca(q) < ca(\rho_r) \leq ca(r) < ret(\rho_r) < ret(q)$.
  Since at time t we are executing $dfs(q)$, we have $ret(\rho_r) < t \leq ret(q)$.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is “yes” iff $t < ret(\rho_r)$.

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is "yes" iff $t < ret(\rho_r)$.

Idea

- Recall $ca(r) \leq t$.
- At time $ret(\rho)$ removes all nodes from ρ's SCC from Rots and Nonroots.
- So r stays in Stack exactly during the interval $[ca(r), ret(root(t))]$, and therefore:

Implementing the oracle

Lemma

Asume the oracle is asked at time t whether $r \rightsquigarrow q$.
The answer is "yes" iff $t < ret(\rho_r)$.

Idea

- Recall $ca(r) \leq t$.
- At time $ret(\rho)$ removes all nodes from ρ's SCC from Rots and Nonroots.
- So r stays in Stack exactly during the interval $[ca(r), ret(root(t))]$, and therefore:

  $t < ret(\rho_r)$ iff $r \in Roots \cup Nonroots$ at time t.

Couvrer and Gabow's algorithm [C99,G00]

```
GCG(q)
    push(q, Roots);
    for each transition q → r
        if r not yet explored then GCG(r)
        elseif r ∈ Roots ∪ Nonroots then
            repeat
                s :=pop(Roots); push(Nonroots);
                if s is accepting report "nonempty"
            until ca(s) ≤ ca(r);
            push(s, Roots); pop(Nonroots)
    if top(Roots) = q then
        pop(Roots);
        while ca(top(Nonroots)) > ca(q)
            pop(Nonroots)
```

Extension to generalized Büchi automata

Store for each state $q \in Roots$ a subset $q.acc$ of accepting sets, maintaining the following invariant:

- $q.acc$ contains all the accepting sets intersecting q's SCC in the part of the graph explored so far.

Extension to generalized Büchi automata

Store for each state $q \in Roots$ a subset $q.acc$ of accepting sets, maintaining the following invariant:

- $q.acc$ contains all the accepting sets intersecting q's SCC in the part of the graph explored so far.

When GC(q) pops a cycle, add all the acc's of the popped states to $q.acc$.

```
EGC(q)
    push(q, Roots);
    q.acc := accepting sets containing q;
    for each transition q → r
        if r not yet explored then EGC(r)
        elseif r ∈ Roots ∪ Nonroots then
            repeat
                s :=pop(Roots); push(s, Nonroots);
                q.acc := q.acc ∪ s.acc
            until ca(s) ≤ ca(r);
            push(s, Roots); pop(Nonroots);
            if q.acc = all accepting sets report "nonempty"
    if q = top(Roots) then
        pop(Roots);
        while ca(top(Nonroots)) > ca(q)
            pop(Nonroots)
```

Couvreur's observation [C99]

The SCC of a root can also be determined as follows:

- Introduce one extra bit b_q for evey state q. Initially $b_q = 0$.
- For every root ρ: at time $ret(\rho)$ conduct a DFS to set to 1 the bits of all states reachable from ρ.
- The set of states that had to be flipped constitute ρ's SCC.

Couvreur's observation [C99]

The SCC of a root can also be determined as follows:

- Introduce one extra bit b_q for evey state q. Initially $b_q = 0$.
- For every root ρ: at time $ret(\rho)$ conduct a DFS to set to 1 the bits of all states reachable from ρ.
- The set of states that had to be flipped constitute ρ's SCC.

Gets rid of Nonroots, but requires one extra DFS.

End of the story? No!

Černá and Pelánek's observation [ČP03]

- Many LTL specifications are translated into weak Büchi automata.
- The four-colour algorithm without the second search is correct for weak automata.

End of the story? No!

Černá and Pelánek's observation [ČP03]

- Many LTL specifications are translated into weak Büchi automata.
- The four-colour algorithm without the second search is correct for weak automata.

Schwoon and E. [SE05]

The four-colour algorithm without the second searches is optimal for weak automata.

End of the story?

	Nested-DFS	SCC-based
Time	2 post ops	1/2 post op
Space	2 bits	2/1 numbers
Optimal	Only for WBA	Yes
Ext. to GBA	No	Yes

End of the story?

	Nested-DFS	SCC-based
Time	2 post ops	1/2 post op
Space	2 bits	2/1 numbers
Optimal	Only for WBA	Yes
Ext. to GBA	No	Yes

Practical relevance of differences in space complexity

- Small when state descriptors explicitely stored.
  (state descriptors are often dozens of bytes long)
- Large when state-hashing is applied.
  (one or two bits for storing a state)

Open questions

- Are there optimal algorithms requiring only a constant number of additional bits per state?

Open questions

- Are there optimal algorithms requiring only a constant number of additional bits per state?
- Are there algorithms for GBA requiring only a constant number of additional bits per state?

Open questions

- Are there optimal algorithms requiring only a constant number of additional bits per state?
- Are there algorithms for GBA requiring only a constant number of additional bits per state?
- Can a shortest counterexample be computed in linear time?

Universal search

- Introduced by Levin.

- Introduced by Levin.
- Used here as a theoretical justification of the need for going beyond Big-Oh analysis.

- Introduced by Levin.
- Used here as a theoretical justification of the need for going beyond Big-Oh analysis.

Intuitively . . .

- Introduced by Levin.
- Used here as a theoretical justification of the need for going beyond Big-Oh analysis.

Intuitively . . .

- Let $A[x]$ be an algorithm computing $F(x)$ in $f(n)$ time. A is optimal for F if no other algorithm computes F in $o(f(n))$ time.

- Introduced by Levin.
- Used here as a theoretical justification of the need for going beyond Big-Oh analysis.

Intuitively . . .

- Let $A[x]$ be an algorithm computing $F(x)$ in $f(n)$ time. A is optimal for F if no other algorithm computes F in $o(f(n))$ time.
- We give a universal algorithm that is optimal for every F.

- Introduced by Levin.
- Used here as a theoretical justification of the need for going beyond Big-Oh analysis.

Intuitively . . .

- Let $A[x]$ be an algorithm computing $F(x)$ in $f(n)$ time. A is optimal for F if no other algorithm computes F in $o(f(n))$ time.
- We give a universal algorithm that is optimal for every F.
- Corollary: if constants don't matter we are all useless!

A bit more formally . . .

- Fix a formal system (i.e., ZF).
- A function is provably computable if some algorithm computes it and the algorithm's correctness is a theorem of the system.

A bit more formally . . .

- Fix a formal system (i.e., ZF).
- A function is provably computable if some algorithm computes it and the algorithm's correctness is a theorem of the system.

Theorem (Levin)

There is an algorithm $U[F, x]$ such that $U[F, -]$ is optimal for every provably computable function F.

A non-optimal algorithm $U_1[F, -]$

We describe first an obviously correct algorithm $U_1[F, -]$.
On input x, $U_1[F, -]$ behaves as follows:

- $U_1[F, -]$ enumerates all pairs $\Pi = (P, D)$, where P program and D derivation of the formal system.
  Let $\Pi_1, \Pi_2, \Pi_3 \ldots$ be this enumeration.
- For every $\Pi_i = (P_i, D_i)$: $U_1[F, -]$ checks if D_i is a proof that P_i computes F. If so, $U_1[F, -]$ computes $P_i[x]$ and stops.

A non-optimal algorithm $U_1[F, -]$

We describe first an obviously correct algorithm $U_1[F, -]$.
On input x, $U_1[F, -]$ behaves as follows:

- $U_1[F, -]$ enumerates all pairs $\Pi = (P, D)$, where P program and D derivation of the formal system.
  Let $\Pi_1, \Pi_2, \Pi_3 \ldots$ be this enumeration.
- For every $\Pi_i = (P_i, D_i)$: $U_1[F, -]$ checks if D_i is a proof that P_i computes F. If so, $U_1[F, -]$ computes $P_i[x]$ and stops.

The algorithm $U[F, -]$

$U[F, x]$ dovetails the computations of $U_1[F, -]$. It spends:

- every second step on Π_1;
- every second step of the remaining ones on Π_2;
- every second step of the remaining ones on Π_3, etc.

Claim

If P runs in $f(n)$ time, then $U[F, -]$ runs in $O(f(n))$ time.

Claim

If P runs in $f(n)$ time, then $U[F, -]$ runs in $O(f(n))$ time.

Proof idea:

Let i be the smallest index such that $P_i = P$ and D_i proves that P computes F. (Observe: i independent of x!)

Claim

If P runs in $f(n)$ time, then $U[F, -]$ runs in $O(f(n))$ time.

Proof idea:

Let i be the smallest index such that $P_i = P$ and D_i proves that P computes F. (Observe: i independent of x!)
Then $U[F, -]$ terminates on input x after executing $f(x)$ steps of Π_i, or earlier.

Claim

If P runs in $f(n)$ time, then $U[F, -]$ runs in $O(f(n))$ time.

Proof idea:

Let i be the smallest index such that $P_i = P$ and D_i proves that P computes F. (Observe: i independent of x!)
Then $U[F, -]$ terminates on input x after executing $f(x)$ steps of Π_i, or earlier.
Total number of steps executed by $U[F, -]$ on x:

So $U[F, -]$ takes at most $2^{i+1} \cdot f(x) = O(f(x))$ steps.

Claim

If P runs in $f(n)$ time, then $U[F, -]$ runs in $O(f(n))$ time.

Proof idea:

Let i be the smallest index such that $P_i = P$ and D_i proves that P computes F. (Observe: i independent of x!)
Then $U[F, -]$ terminates on input x after executing $f(x)$ steps of Π_i, or earlier.
Total number of steps executed by $U[F, -]$ on x:

- Steps spent on $\Pi_i, \Pi_{i-1}, \ldots, \Pi_1$:
  $f(x) + 2f(x) + 2^2 f(x) + \ldots + 2^i f(x) = (2^{i+1} - 1) f(x)$

So $U[F, -]$ takes at most $2^{i+1} \cdot f(x) = O(f(x))$ steps.

Claim

If P runs in $f(n)$ time, then $U[F, -]$ runs in $O(f(n))$ time.

Proof idea:

Let i be the smallest index such that $P_i = P$ and D_i proves that P computes F. (Observe: i independent of x!)
Then $U[F, -]$ terminates on input x after executing $f(x)$ steps of Π_i, or earlier.
Total number of steps executed by $U[F, -]$ on x:

- Steps spent on $\Pi_i, \Pi_{i-1}, \ldots, \Pi_1$:
  $f(x) + 2f(x) + 2^2 f(x) + \ldots + 2^i f(x) = (2^{i+1} - 1)f(x)$
- Steps spent on $\Pi_{i+1}, \Pi_{i+2}, \ldots$:
  $\frac{1}{2}f(x) + \frac{1}{4}f(x) + \ldots + 1 \leq f(x) = f(x)$

So $U[F, -]$ takes at most $2^{i+1} \cdot f(x) = O(f(x))$ steps.

Conclusions

Conclusions

- Going beyond Big-Oh analysis in verification is important.

Conclusions

- Going beyond Big-Oh analysis in verification is important.
- It is not only about heuristics and hacking: good theory is waiting for us there.