Formal Proofs of Tarjan's Strongly Connected Components Algorithm in Why3, Coq, and Isabelle

R. Chen [1], C. Cohen [2], J.-J. Lévy [3], S. Merz [4], L. Théry [2]

[1] ISCAS, Beijing
[2] Univ. Côte d'Azur, Inria, Sophia Antipolis
[3] IRIF, Inria, Paris
[4] Univ. de Lorraine, CNRS, Inria, LORIA, Nancy







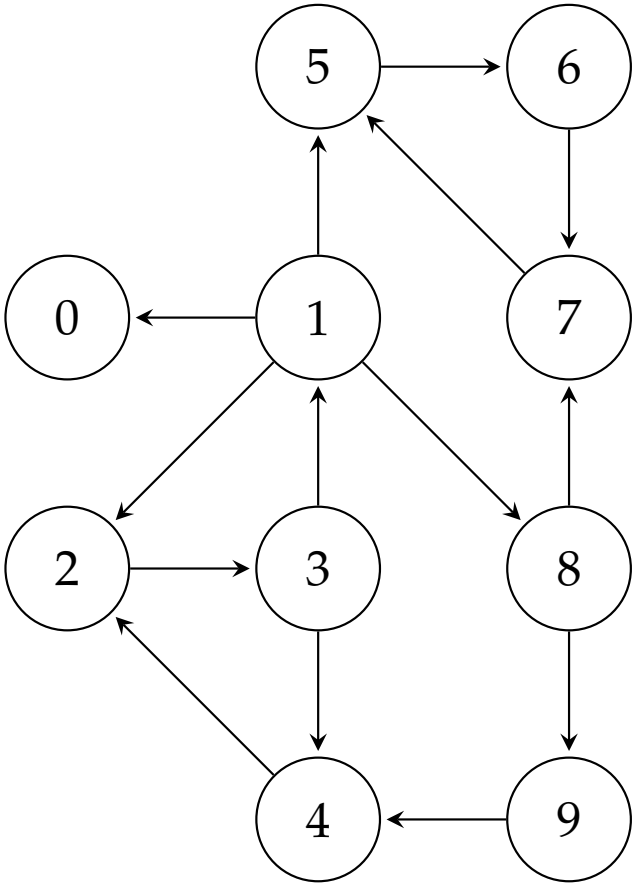
ITP 2019 Portland, OR

Motivations

- Algorithms should be accompanied by formal proofs
  - checked by proof assistants
  - published in conferences or journals
- Graph algorithms have interesting proofs
  - moderately complex invariants
  - testbed for exploring proof automation
- Compare different proof assistants
  - existing Why3 proof of Tarjan's algorithm (VSTTE 2017)
  - focus on widely available, elementary features

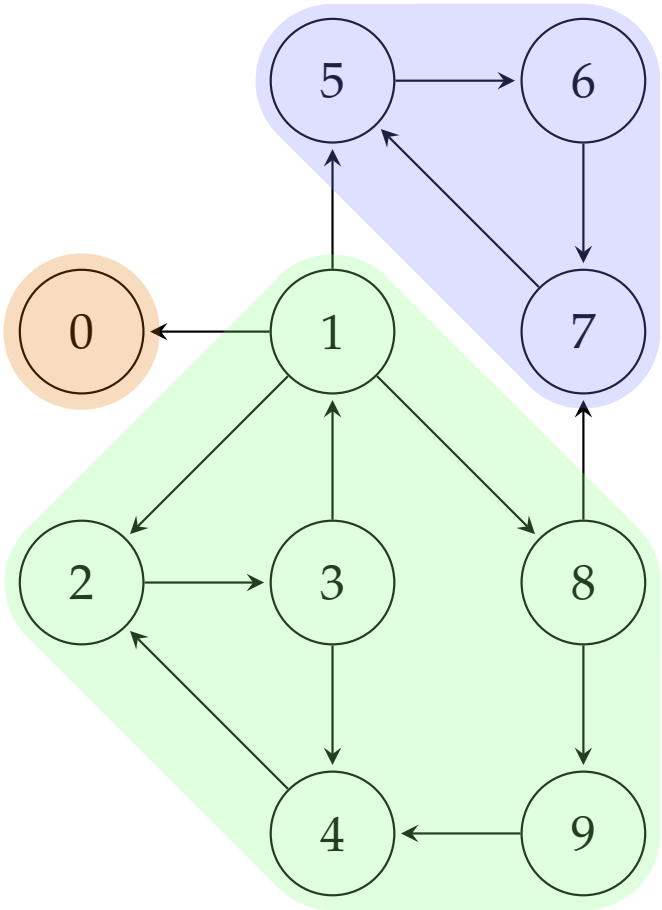
Contents

Strongly Connected Components



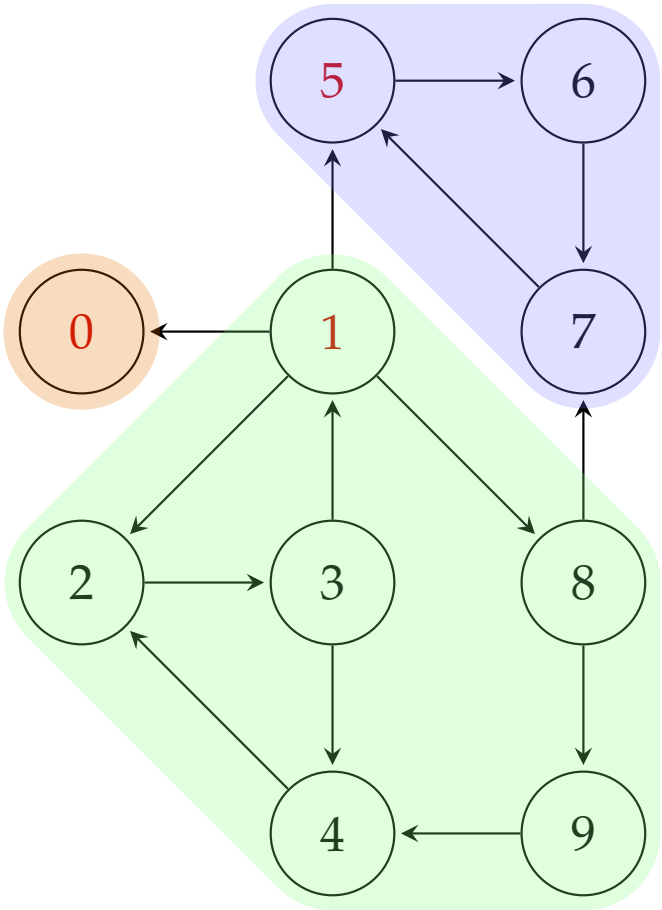
- x, y strongly connected iff mutually reachable
- SCC: maximal set of strongly connected vertices

Strongly Connected Components



- x, y strongly connected iff mutually reachable
- SCC: maximal set of strongly connected vertices
- SCCs partition the set of vertices

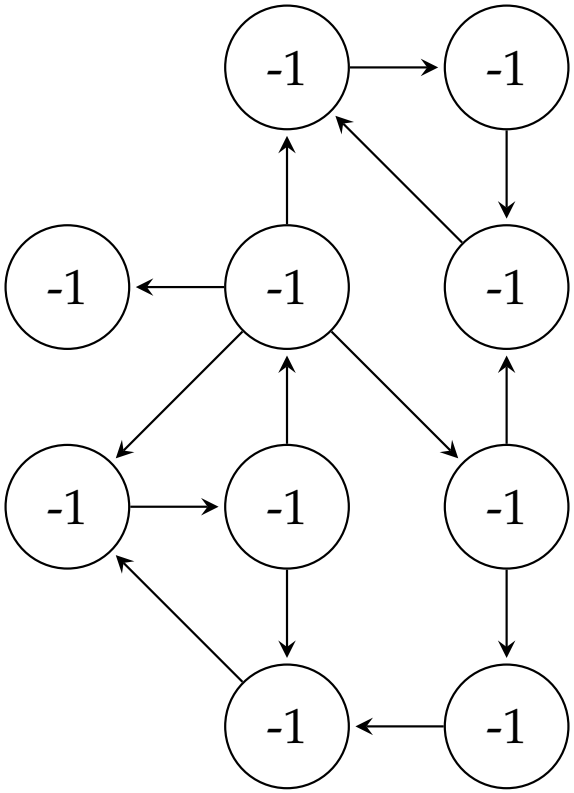
Strongly Connected Components



- x, y strongly connected iff mutually reachable
- SCC: maximal set of strongly connected vertices
- SCCs partition the set of vertices
- Tarjan's algorithm computes SCC bases

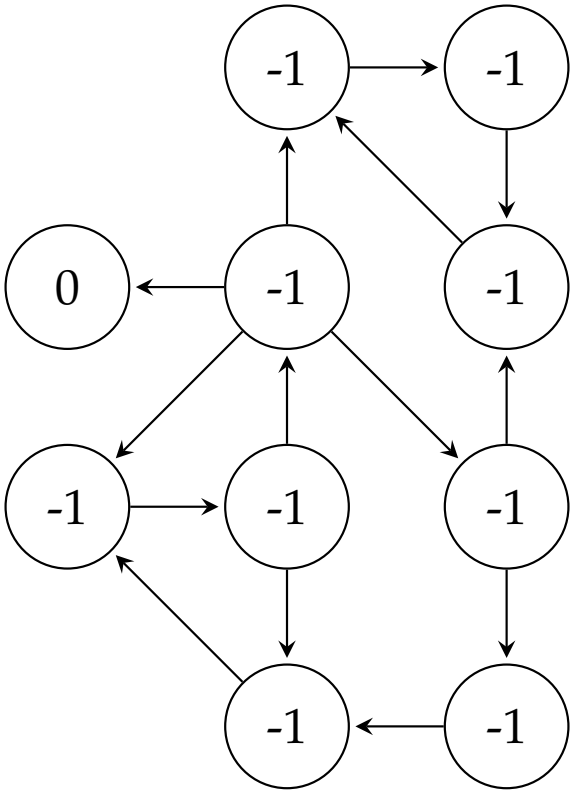
Computing SCCs is a fundamental algorithmic problem

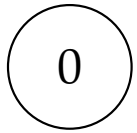
Exercising Tarjan's Algorithm



vertex	0	1	2	3	4	5	6	7	8	9
low link										

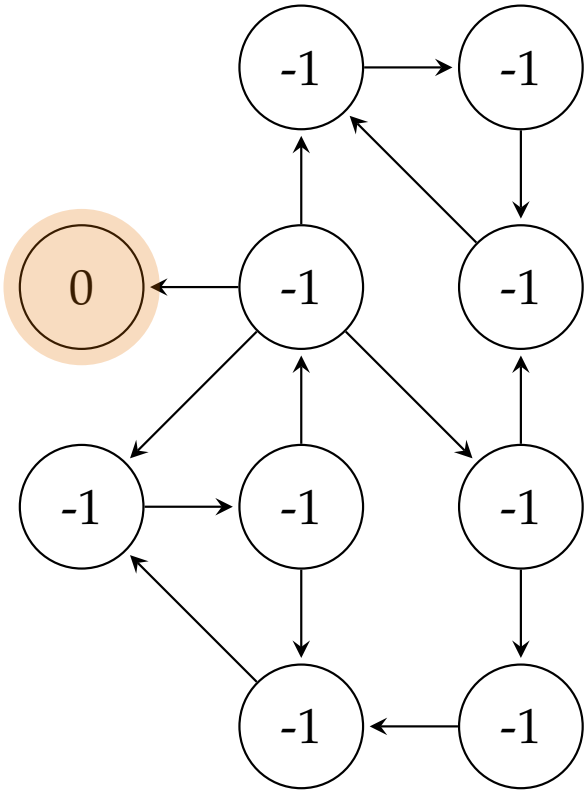
Exercising Tarjan's Algorithm





vertex	0	1	2	3	4	5	6	7	8	9
low link										

Exercising Tarjan's Algorithm

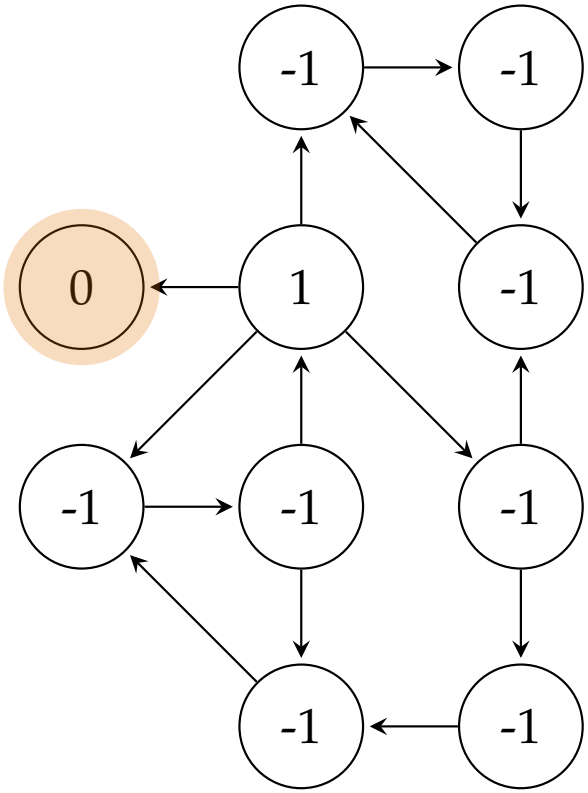


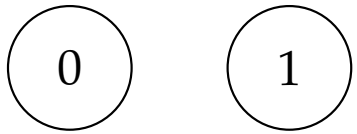
0 ␣

0

vertex	0	1	2	3	4	5	6	7	8	9
low link	∞									

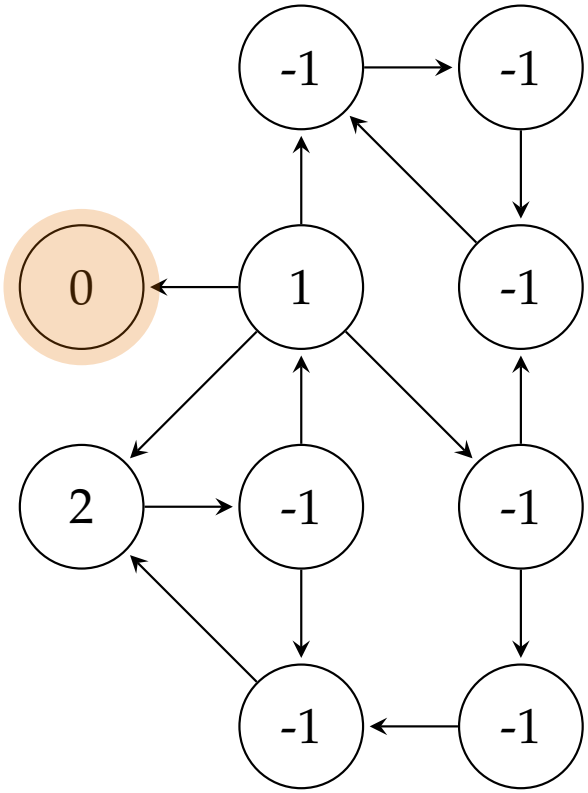
Exercising Tarjan's Algorithm

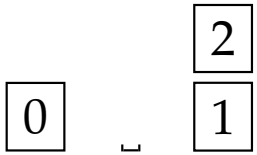


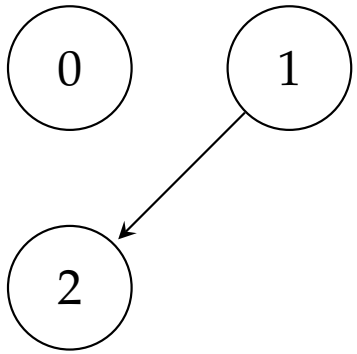


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞									

Exercising Tarjan's Algorithm

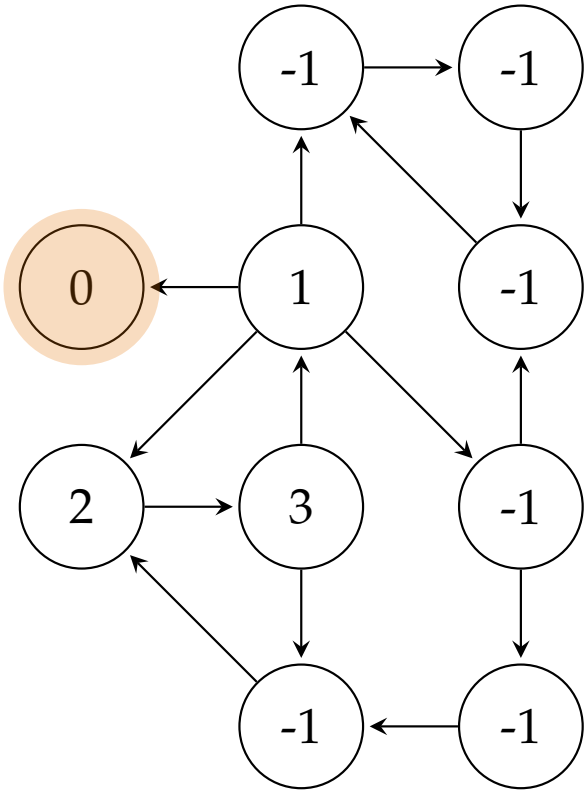


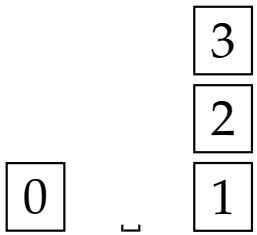


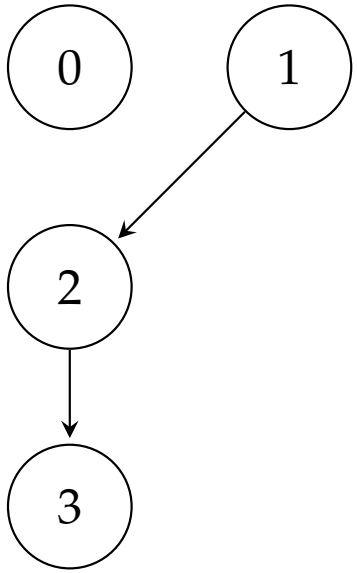


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞									

Exercising Tarjan's Algorithm

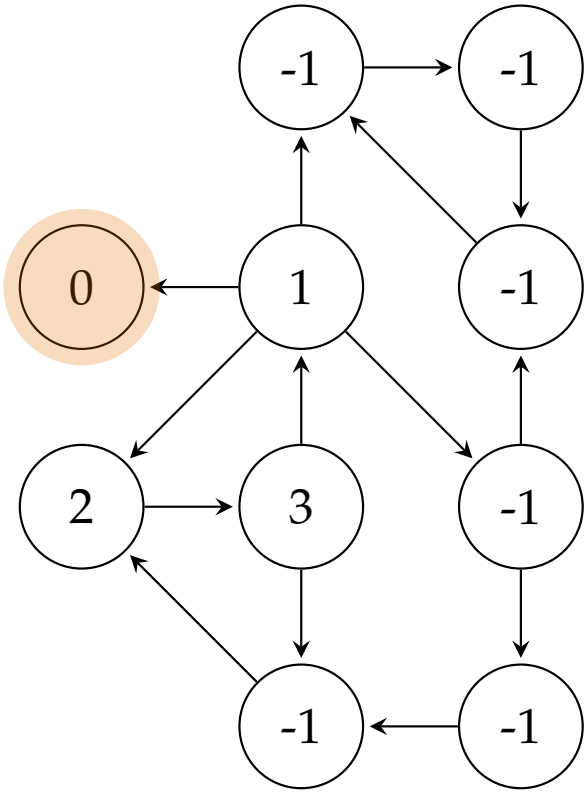


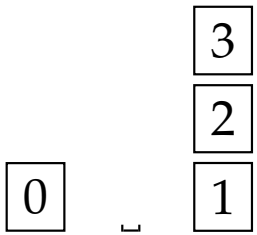


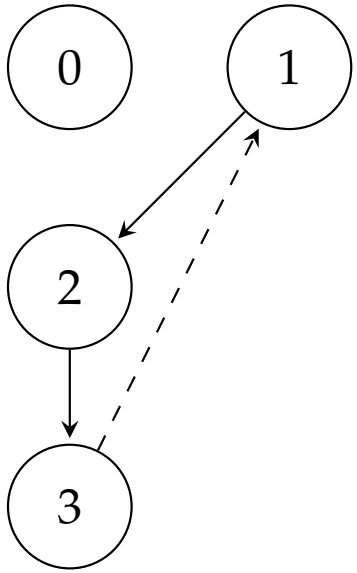


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞									

Exercising Tarjan's Algorithm

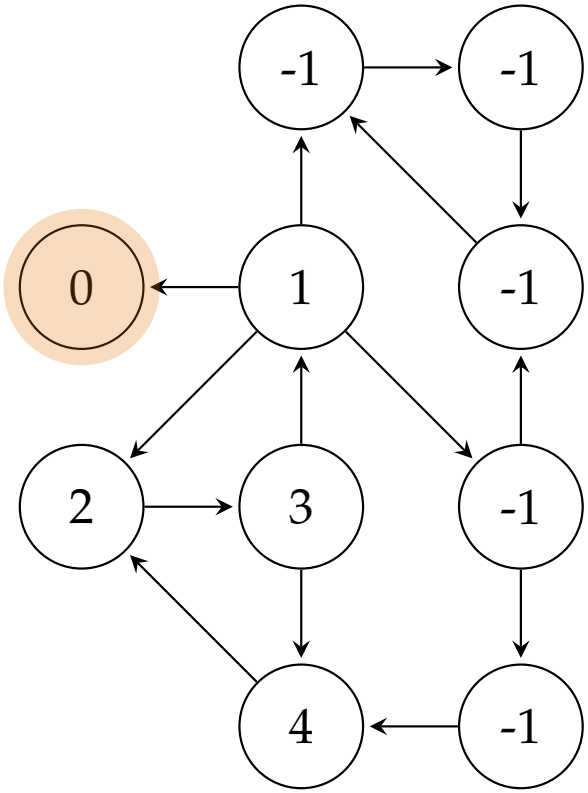


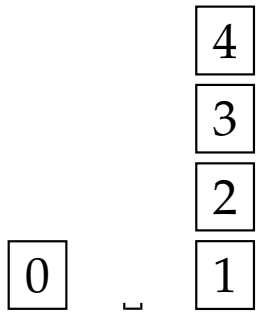


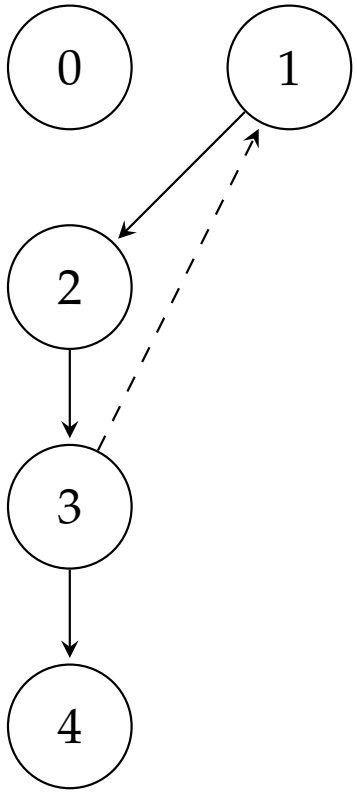


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞			(1)						

Exercising Tarjan's Algorithm

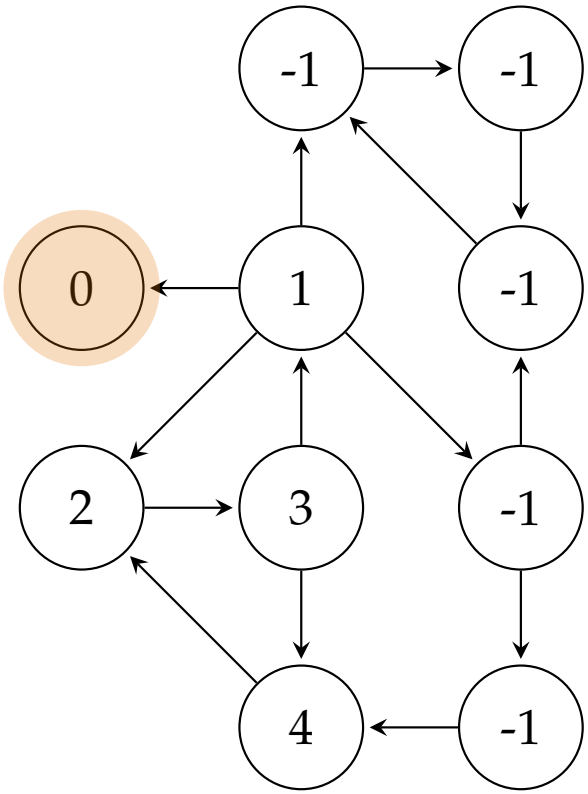


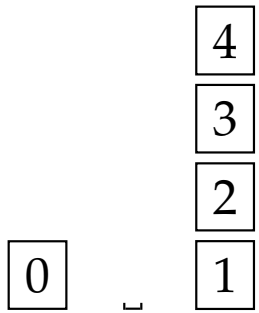


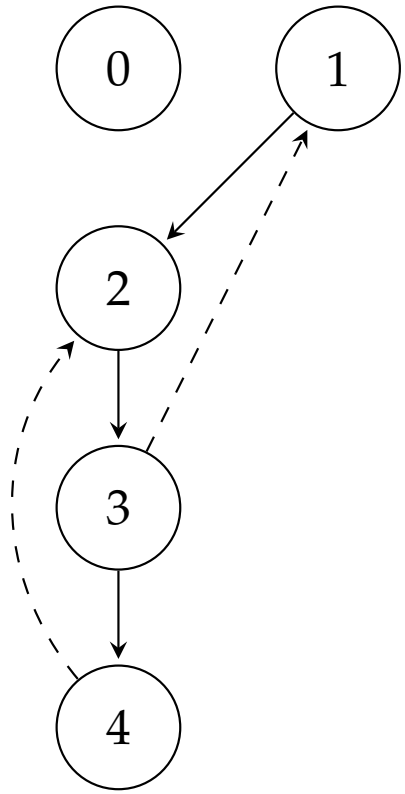


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞			(1)						

Exercising Tarjan's Algorithm

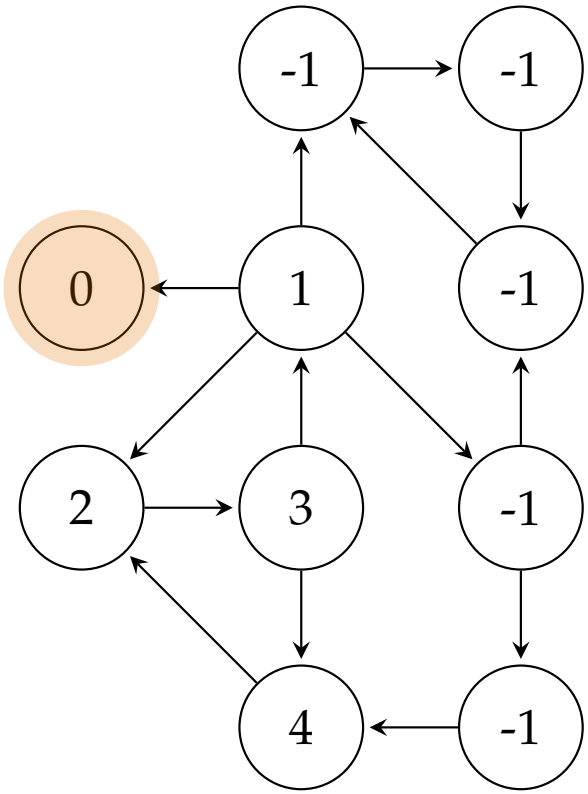


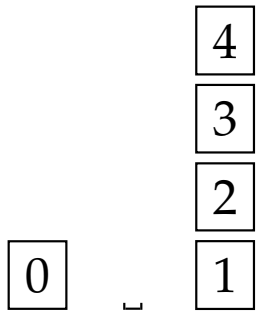


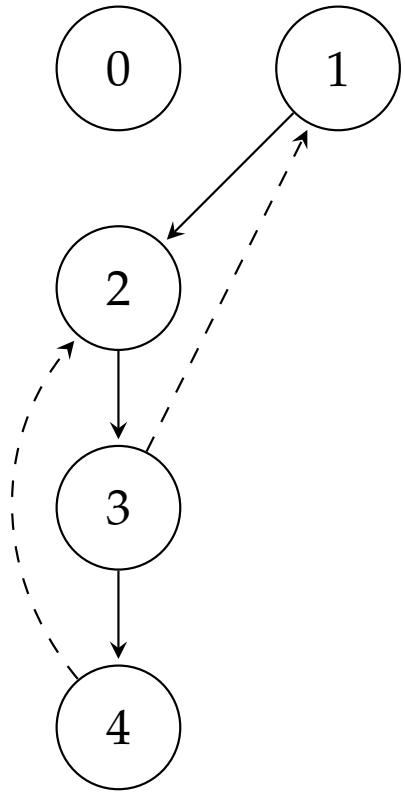


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞			(1)	2					

Exercising Tarjan's Algorithm

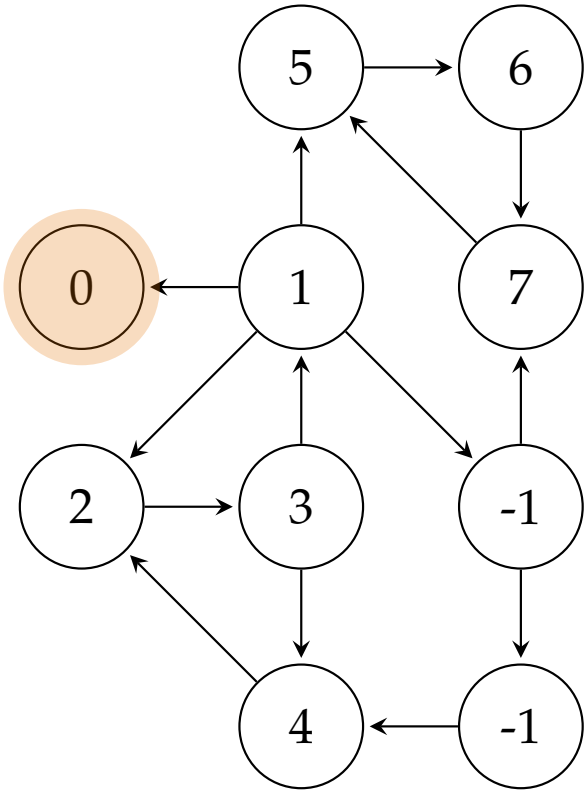


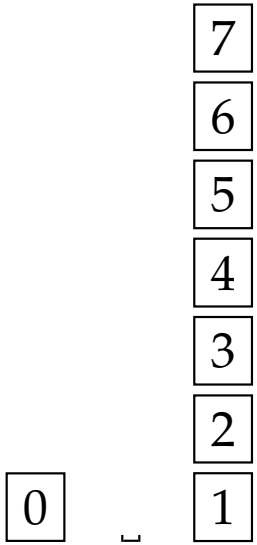


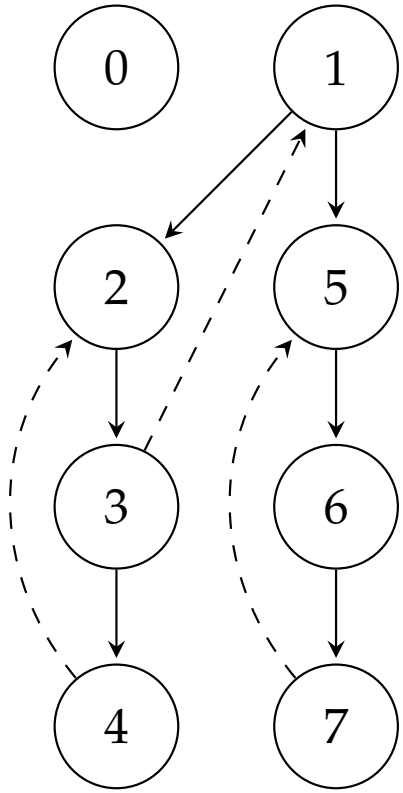


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞	(1)	1	1	2					

Exercising Tarjan's Algorithm

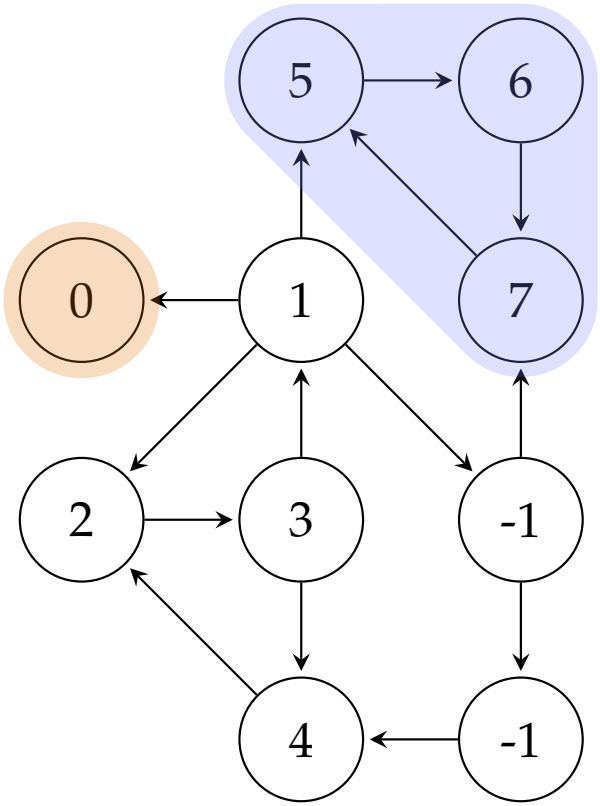


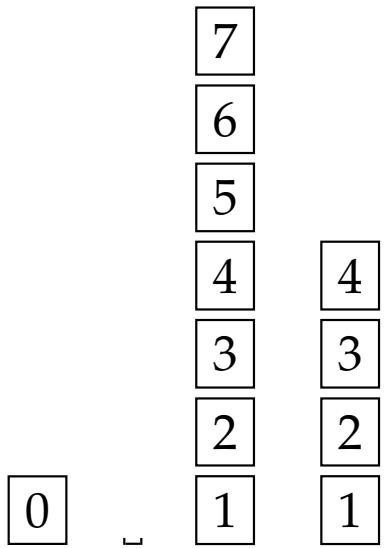


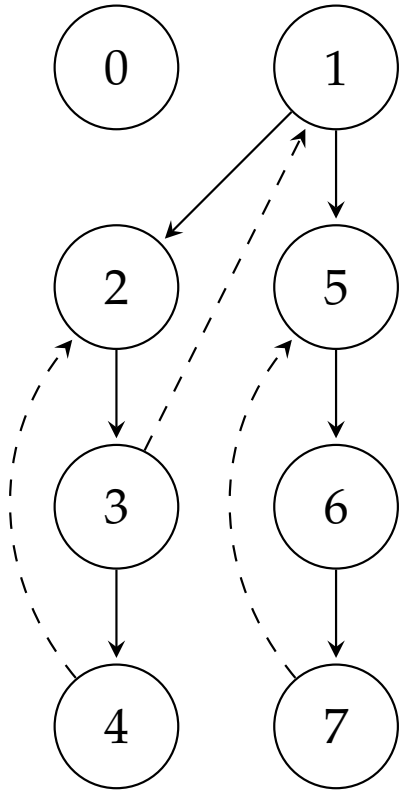


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞	(1)	1	1	2	5	5	5		

Exercising Tarjan's Algorithm

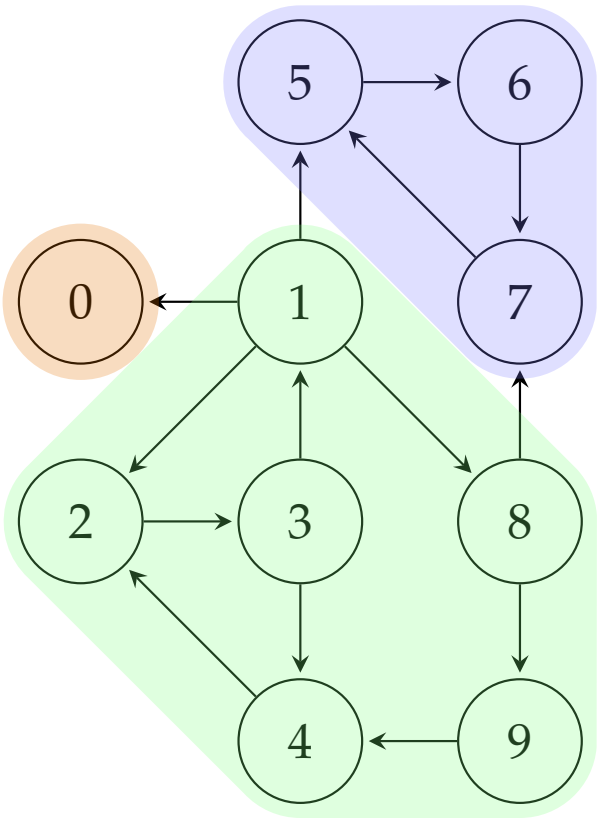


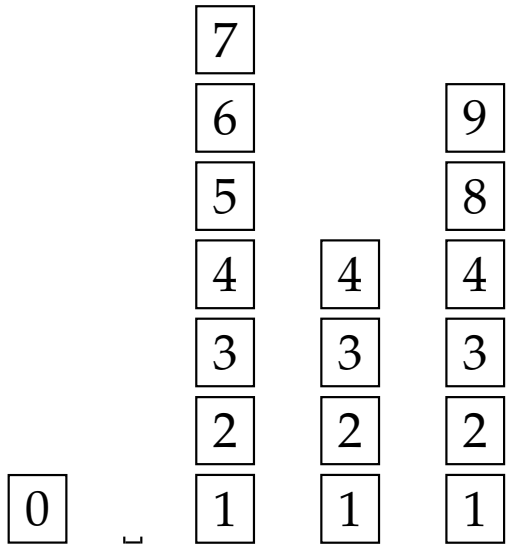


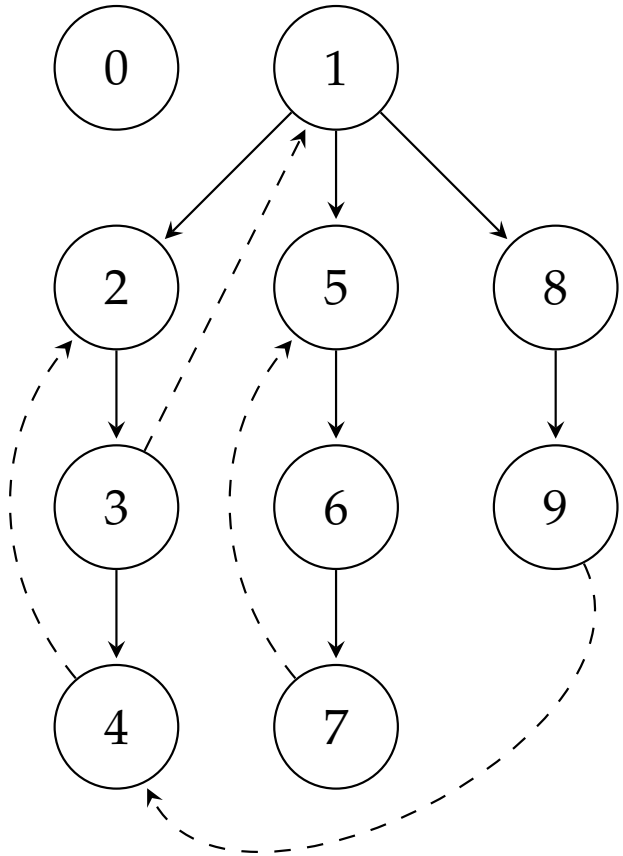


vertex	0	1	2	3	4	5	6	7	8	9
low link	∞	(1)	1	1	2	5	5	5		

Exercising Tarjan's Algorithm







vertex	0	1	2	3	4	5	6	7	8	9
low link	∞	1	1	1	2	5	5	5	4	4

Tarjan's Algorithm (1/2)

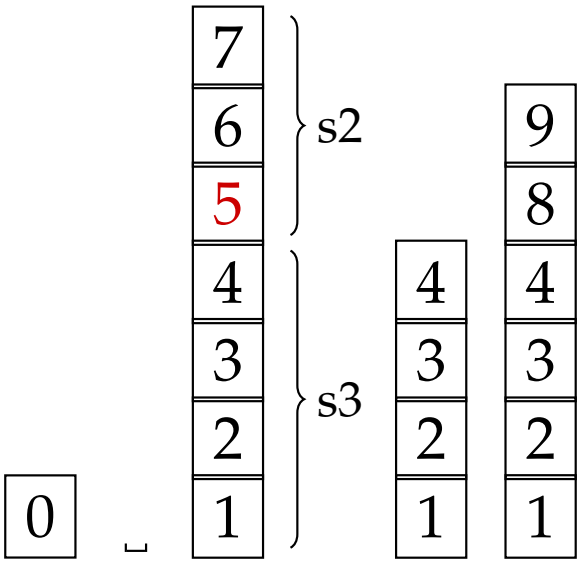
```
type vertex
constant vertices: set vertex
function successors vertex : set vertex

type env = {stack: list vertex;
            sccs: set (set vertex);
            sn: int; num: map vertex int}



let tarjan () =
  let e = {stack = Nil; sccs = empty;
           sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e in e'.sccs
```

Tarjan's Algorithm (2/2)

```
let rec dfs roots e =
  if is_empty roots then (+∞, e) else
  let x = choose roots in
  let roots' = remove x roots in
  let (n1, e1) = if e.num[x] ≠ -1
                 then (e.num[x], e) else dfs1 x e in
  let (n2, e2) = dfs roots' e1 in (min n1 n2, e2)
```

Tarjan's Algorithm (2/2)

```
let rec dfs roots e =
  if is_empty roots then (+∞, e) else
  let x = choose roots in
  let roots' = remove x roots in
  let (n1, e1) = if e.num[x] ≠ -1
                 then (e.num[x], e) else dfs1 x e in
  let (n2, e2) = dfs roots' e1 in (min n1 n2, e2)
```




```
with dfs1 x e =
let n0 = e.sn in
let (n1, e1) = dfs (successors x)
                   (add_stack_incr x e) in
if n1 < n0 then (n1, e1) else
  let (s2, s3) = split x e1.stack in
  (+∞, {stack = s3;
        sccs = add (elts s2) e1.sccs;
        sn = e1.sn;
        num = set_infty s2 e1.num})
```

Contents

Coloring Nodes

- Monitor to what extent a node has been explored
  - white not yet visited
  - grey exploration has started
  - black node (and successors) has been fully explored
- Extend environment for the proof

```
type env = {ghost black : set vertex;
            ghost grey : set vertex;
            stack: list vertex;
            sccs: set (set vertex);
            sn: int; num: map vertex int}
```

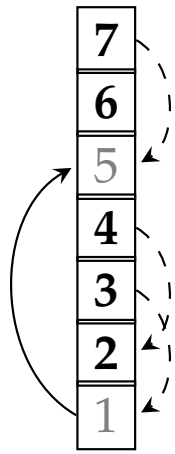
- Adapt algorithm accordingly
  - ghost fields do not affect computation: used for assertions

Invariants of the Algorithm

- Structural invariants
  - coloring: no edge from black to white, SCC nodes are black, ...
  - numbering: white nodes -1, ascending numbers on stack, ...

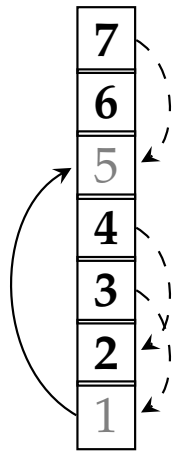
Invariants of the Algorithm

- Structural invariants
  - coloring: no edge from black to white, SCC nodes are black, ...
  - numbering: white nodes -1, ascending numbers on stack, ...
- Reachability of nodes on the stack
  - every node reaches every higher node
  - every node reaches some grey lower node



Invariants of the Algorithm

- Structural invariants
  - coloring: no edge from black to white, SCC nodes are black, ...
  - numbering: white nodes -1, ascending numbers on stack, ...
- Reachability of nodes on the stack
  - every node reaches every higher node
  - every node reaches some grey lower node



- SCCs computed so far
  - sccs field contains precisely the black SCCs of the graph

Pre- and Postconditions

```
let rec dfs1 x e  =
(* pre-conditions *)
requires {mem x vertices ∧ not mem x (union e.black e.gray)}
requires {∀y. mem y e.gray → reachable y x}
requires {wf_env e}

(* post-conditions *)
returns {(_, e') → wf_env e' ∧ subenv e e'}
returns {(_, e') → mem x e'.black}
returns {(n, e') → n ≤ e'.num[x]}
returns {(n, e') → n = +∞ ∨
         (∃y. lmem y e.stack ∧ n = e.num[y] ∧ reachable x y)}
returns {(n, e') →
         ∀y. xedge_to e'.stack e.stack y → n ≤ e'.num[y]}
```

Pre- and Postconditions

```
let rec dfs1 x e  =
(* pre-conditions *)
requires {mem x vertices ∧ not mem x (union e.black e.gray)}
requires {∀y. mem y e.gray → reachable y x}
requires {wf_env e}

(* post-conditions *)
returns {(_, e') → wf_env e' ∧ subenv e e'}
returns {(_, e') → mem x e'.black}
returns {(n, e') → n ≤ e'.num[x]}
returns {(n, e') → n = +∞ ∨
         (∃y. lmem y e.stack ∧ n = e.num[y] ∧ reachable x y)}
returns {(n, e') →
         ∀y. xedge_to e'.stack e.stack y → n ≤ e'.num[y]}
```

- Similar assertions for function `dfs`

Contents

The Proof in Why3 (1)

- Proof guidance mainly through assertions
  - help bridge pre- and post-conditions
  - very natural for algorithm designers and programmers
  - finding helpful assertions requires expertise

```
let n0 = e.sn in
let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
if n1 < n0 then begin
  assert {n1 ≠ +∞};
  assert {∃y. y ≠ x ∧ mem y e1.gray
            ∧ e1.num[y] < e1.num[x]
            ∧ in_same_scc x y};
  (n1, add_black x e1) end
else ...
```

The Proof in Why3 (1)

- Proof guidance mainly through assertions
  - help bridge pre- and post-conditions
  - very natural for algorithm designers and programmers
  - finding helpful assertions requires expertise

```
let n0 = e.sn in
let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
if n1 < n0 then begin
  assert {n1 ≠ +∞};
  assert {∃y. y ≠ x ∧ mem y e1.gray
            ∧ e1.num[y] < e1.num[x]
            ∧ in_same_scc x y};
  (n1, add_black x e1) end
else ...
```

- Besides: several lemmas about auxiliary functions

The Proof in Why3 (2)

- Diversity of proof backends
  - most POs are discharged by SMT solvers or superposition provers
  - two Coq proofs, in particular completeness of SCC
  - backends (and translations) are part of the trusted code base

The Proof in Why3 (2)

- Diversity of proof backends
  - most POs are discharged by SMT solvers or superposition provers
  - two Coq proofs, in particular completeness of SCC
  - backends (and translations) are part of the trusted code base
- Proof of termination
  - state appropriate variant expressions

```
let rec dfs1 x e  = variant {
   cardinal (diff vertices (union e.black e.gray)),
   1, 0}
with dfs roots e = variant {
   cardinal (diff vertices (union e.black e.gray)),
   cardinal roots, 1}
```

  - proved automatically, relying on existing assertions

The Proof in Coq (1)

- More abstract reformulation of the algorithm
  - version based on Why3 formalization considered too tedious
  - stripped-down representation of environment

```
Record env := Env {esccs : {set {set V}};
                   num: {ffun V → nat}}.
```

  - stack: vertices x with $0 \leq num\ x < \infty$, in order

The Proof in Coq (1)

- More abstract reformulation of the algorithm
  - version based on Why3 formalization considered too tedious
  - stripped-down representation of environment

```
Record env := Env {esccs : {set {set V}};
                   num: {ffun V → nat}}.
```

  - stack: vertices x with $0 \leq num\ x < \infty$, in order

- Avoid mutually recursive function definition
  - two separate functions `dfs` and `dfs1`
  - each takes a function argument representing the other function
  - tied together by a recursor, with "fuel" parameter

The Proof in Coq (2)

- Proof of partial correctness
  - define predicates for invariants, pre- and post-conditions
  - correctness based on implications relating those predicates

The Proof in Coq (2)

- Proof of partial correctness
  - define predicates for invariants, pre- and post-conditions
  - correctness based on implications relating those predicates
- Proof of termination
  - trivial because "fuel" decreases on every recursive call
  - show that the computation never runs out of "fuel"

```
Theorem rec_terminates k (roots : {set V}) e :
  k ≥ #|~: visited e| * (∞.+1) + #|roots| →
  dfs_correct (rec k) roots e.
Theorem tarjan_correct : tarjan = gsccs.
```

The Proof in Coq (2)

- Proof of partial correctness
  - define predicates for invariants, pre- and post-conditions
  - correctness based on implications relating those predicates
- Proof of termination
  - trivial because "fuel" decreases on every recursive call
  - show that the computation never runs out of "fuel"

```
Theorem rec_terminates k (roots : {set V}) e :
  k ≥ #|~: visited e| * (∞.+1) + #|roots| →
  dfs_correct (rec k) roots e.
Theorem tarjan_correct : tarjan = gsccs.
```

- Comments on the proof
  - mostly elementary tactics with little automation
  - overall 900 lines, most proofs are short (max. 124 loc)

The Proof in Isabelle/HOL (1)

- Formalization closely aligned with Why3 version
  - represent graph structure as Isabelle locale
  - take advantage of higher-order features, e.g. for reachability

The Proof in Isabelle/HOL (1)

- Formalization closely aligned with Why3 version
  - represent graph structure as Isabelle locale
  - take advantage of higher-order features, e.g. for reachability
- Delayed termination proof
  - termination requires mild pre-condition
  - prove domain condition by well-founded induction on variants

```
theorem dfs1_dfs_termination:
  ⟦x ∈ vertices - colored e; colored_num e⟧ ⟹
      dfs1_dfs_dom (Inl(x,e))
  ⟦roots ⊆ vertices; colored_num e⟧ ⟹
      dfs1_dfs_dom (Inr(roots,e))
```

The Proof in Isabelle/HOL (2)

- Proof of partial correctness
  - define predicates for invariants, pre- and post-conditions
  - prove appropriate implications to infer partial correctness
- Combine partial correctness and termination

```
theorem dfs_correct:
  dfs1_pre x e ⟹ dfs1_post x e (dfs1 x e)
  dfs_pre roots e ⟹ dfs_post roots e (dfs roots e)
theorem tarjan_correct:
  tarjan = { C . is_scc C ∧ C ⊆ vertices }
```

The Proof in Isabelle/HOL (2)

- Proof of partial correctness
  - define predicates for invariants, pre- and post-conditions
  - prove appropriate implications to infer partial correctness
- Combine partial correctness and termination

```
theorem dfs_correct:
  dfs1_pre x e ⟹ dfs1_post x e (dfs1 x e)
  dfs_pre roots e ⟹ dfs_post roots e (dfs roots e)
theorem tarjan_correct:
  tarjan = { C . is_scc C ∧ C ⊆ vertices }
```

- Comments on the proof
  - extensive use of sledgehammer, but finer-grained interactions
  - majority of proofs require few steps, max. 48 interactions

Contents

Subjective Comparison of the Systems

	Why3	Coq	Isabelle
expressivity	−	+	+
readability	+	−	+
stability w.r.t changes	−	+	+
ease of use	−	−	−
automation	+	−	(+)
trusted code base	−	+	+
lines of automatic proof	395	0	314 ui
lines of manual proof	90	898	1690

Do we expect our systems to be used by programmers?

Overall Comments on the Proof

- Formal proofs different from arguments in Tarjan's paper
  - colors, numbering, reachability between nodes on stack
  - no explicit mention of spanning forest
- Essentially no use of graph theory
  - did not need anything beyond (mutual) reachability
  - correctness of Tarjan's algorithm doesn't require libraries
- Choice of level of abstraction
  - tension between concise proofs and explicit data structures
  - aimed for elementary definitions, no refinement to programs
- We would love to see the proof in your system!

`www-sop.inria.fr/marelle/Tarjan/contributions.html`