

# Proofs of graph algorithms with automation and their readability

Jean-Jacques Lévy  
**Iris - Inria**

Irif, 08/01/2018

# Plan

- motivation
- algorithm
- formal proof
- other systems
- conclusion

.. joint work (in progress) with **Ran Chen** [VSTTE 2017])

also cooperation with Cyril Cohen, Laurent Théry, Stephan Merz

# DFS in Sedgwick in C

```
int val[maxV]; int id = 0;
visit(int k)
{
    struct node *t;
    val[k] = ++id;
    for (t = adj[k]; t != z; t = t->next)
        if (val[t->v] == 0) visit(t->v);
}
listdfs()
{
    int k;
    for (k = 1; k <= V; k++) val[k] = 0;
    for (k = 1; k <= V; k++)
        if (val[k] == 0) visit(k);
}
```

Imperative style

# DFS in Cormen in pseudocode

DFS( $G$ )

```
1 for each vertex  $u \in V[G]$ 
2   do  $color[u] \leftarrow WHITE$ 
3      $\pi[u] \leftarrow NIL$ 
4  $time \leftarrow 0$ 
5 for each vertex  $u \in V[G]$ 
6   do if  $color[u] = WHITE$ 
7     then DFS-VISIT( $u$ )
```

DFS-VISIT( $u$ )

```
1  $color[u] \leftarrow GRAY$       ▷ White vertex  $u$  has just been discovered.
2  $d[u] \leftarrow time \leftarrow time + 1$ 
3 for each  $v \in Adj[u]$       ▷ Explore edge  $(u, v)$ .
4   do if  $color[v] = WHITE$ 
5     then  $\pi[v] \leftarrow u$ 
6         DFS-VISIT( $v$ )
7  $color[u] \leftarrow BLACK$     ▷ Blacken  $u$ ; it is finished.
8  $f[u] \leftarrow time \leftarrow time + 1$ 
```

Imperative style

# DFS in Why3 language

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
  forall x. mem x vertices -> subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices /\ mem y (successors x)
```

# DFS in Why3 language

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
  forall x. mem x vertices -> subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices /\ mem y (successors x)
```

- a **functional** version with **finite sets**

```
let rec dfs (roots visited: set vertex): set vertex =
  if is_empty roots then visited else
    let x = choose roots in
      let roots' = remove x roots in
        if mem x visited then
          dfs roots' visited
        else
          let v' = dfs (successors x) (add x visited) in
            dfs roots' (union visited v')

let dfs_main (roots: set vertex) : set vertex =
  dfs roots empty
```

Functional programming

# DFS in Why3 language

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
  forall x. mem x vertices -> subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices /\ mem y (successors x)
```

- a **functional** version with **finite sets** and type inference

```
let rec dfs roots visited =
  if is_empty roots then visited else
    let x = choose roots in
      let roots' = remove x roots in
        if mem x visited then
          dfs roots' visited
        else
          let v' = dfs (successors x) (add x visited) in
            dfs roots' (union visited v')

let dfs_main roots =
  dfs roots empty
```

Functional programming

# DFS in Why3 language

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
  forall x. mem x vertices -> subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices /\ mem y (successors x)
```

- a **functional** version with **finite sets** and type inference

```
let rec dfs r v =
  if is_empty r then v else
    let x = choose r in
    let r' = remove x r in
    if mem x v then
      dfs r' v
    else
      let v' = dfs (successors x) (add x v) in
      dfs r' (union v v')
```

```
let dfs_main roots =
  dfs roots empty
```


Functional programming



# DFS in imperative style

- data are efficiently implemented:
  - **finite sets** → lists or arrays
  - direct access from nodes to their successors → array of lists of successors
- programming languages problems:
  - array bounds checking
  - mutable variables: uneasy notations, frame problem
- not treated here


# DFS with formal proofs in literature

- for us, formal proofs  checked by computer
- in Isabelle / HOL
- in Coq / ssreflect in the **mathematical components** library

<http://math-comp.github.io/math-comp/html/doc/libgraph.html>


<https://github.com/math-comp/math-comp/blob/master/mathcomp/ssreflect/fingraph.v>

# DFS with formal proofs in literature


- for us, formal proofs  checked by computer
- in Isabelle / HOL
- in Coq / ssreflect in the **mathematical components** library

<http://math-comp.github.io/math-comp/html/doc/libgraph.html>

<https://github.com/math-comp/math-comp/blob/master/mathcomp/ssreflect/fingraph.v>

- proofs to be read by computer
- read these proofs  good training

# Formal proofs read by computer

- proofs with many cases
- very long proofs
- getting harder with structured proofs
  
- automatic provers could simplify
- but often loosing proof certificates
  
- basic algorithms  readable proofs (by humans)
  - teaching formal methods
  - stressing on methodology for simplification

# Correctness proofs in algorithms book

DFS( $G$ )

```
1 for each vertex  $u \in V[G]$ 
2   do  $color[u] \leftarrow WHITE$ 
3      $\pi[u] \leftarrow NIL$ 
4  $time \leftarrow 0$ 
5 for each vertex  $u \in V[G]$ 
6   do if  $color[u] = WHITE$ 
7     then DFS-VISIT( $u$ )
```

DFS-VISIT( $u$ )

```
1  $color[u] \leftarrow GRAY$       ▷ White vertex  $u$  has just been discovered.
2  $d[u] \leftarrow time \leftarrow time + 1$ 
3 for each  $v \in Adj[u]$         ▷ Explore edge  $(u, v)$ .
4   do if  $color[v] = WHITE$ 
5     then  $\pi[v] \leftarrow u$ 
6         DFS-VISIT( $v$ )
7  $color[u] \leftarrow BLACK$     ▷ Blacken  $u$ ; it is finished.
8  $f[u] \leftarrow time \leftarrow time + 1$ 
```

Imperative style

# Correctness proofs in algorithms book

## *Corollary 22.8 (Nesting of descendants' intervals)*

Vertex  $v$  is a proper descendant of vertex  $u$  in the depth-first forest for a (directed or undirected) graph  $G$  if and only if  $u.d < v.d < v.f < u.f$ .

## *Theorem 22.9 (White-path theorem)*

In a depth-first forest of a (directed or undirected) graph  $G = (V, E)$ , vertex  $v$  is a descendant of vertex  $u$  if and only if at the time  $u.d$  that the search discovers  $u$ , there is a path from  $u$  to  $v$  consisting entirely of white vertices.

$\Leftarrow$ : Suppose that there is a path of white vertices from  $u$  to  $v$  at time  $u.d$ , but  $v$  does not become a descendant of  $u$  in the depth-first tree. Without loss of generality, assume that every vertex other than  $v$  along the path becomes a descendant of  $u$ . (Otherwise, let  $v$  be the closest vertex to  $u$  along the path that doesn't become a descendant of  $u$ .) Let  $w$  be the predecessor of  $v$  in the path, so that  $w$  is a descendant of  $u$  ( $w$  and  $u$  may in fact be the same vertex). By Corollary 22.8,  $w.f \leq u.f$ . Because  $v$  must be discovered after  $u$  is discovered, but before  $w$  is finished, we have  $u.d < v.d < w.f \leq u.f$ . Theorem 22.7 then implies that the interval  $[v.d, v.f]$

??

# Random search in Why3 language

```
type vertex
constant vertices: set vertex
function successors vertex : set vertex
axiom successors_vertices:
  forall x. mem x vertices -> subset (successors x) vertices
predicate edge (x y: vertex) = mem x vertices /\ mem y (successors x)
```

- one step of any traversal strategy [[dowek](#), [munoz](#)]

```
let rec random_search r v =
  if is_empty r then v else
    let x = choose r in
      let r' = remove x r in
        if mem x v then
          random_search r' v
        else
          random_search (union r' (successors x)) (add x v)

let random_search_main roots =
  random_search roots empty
```

# Random search completeness

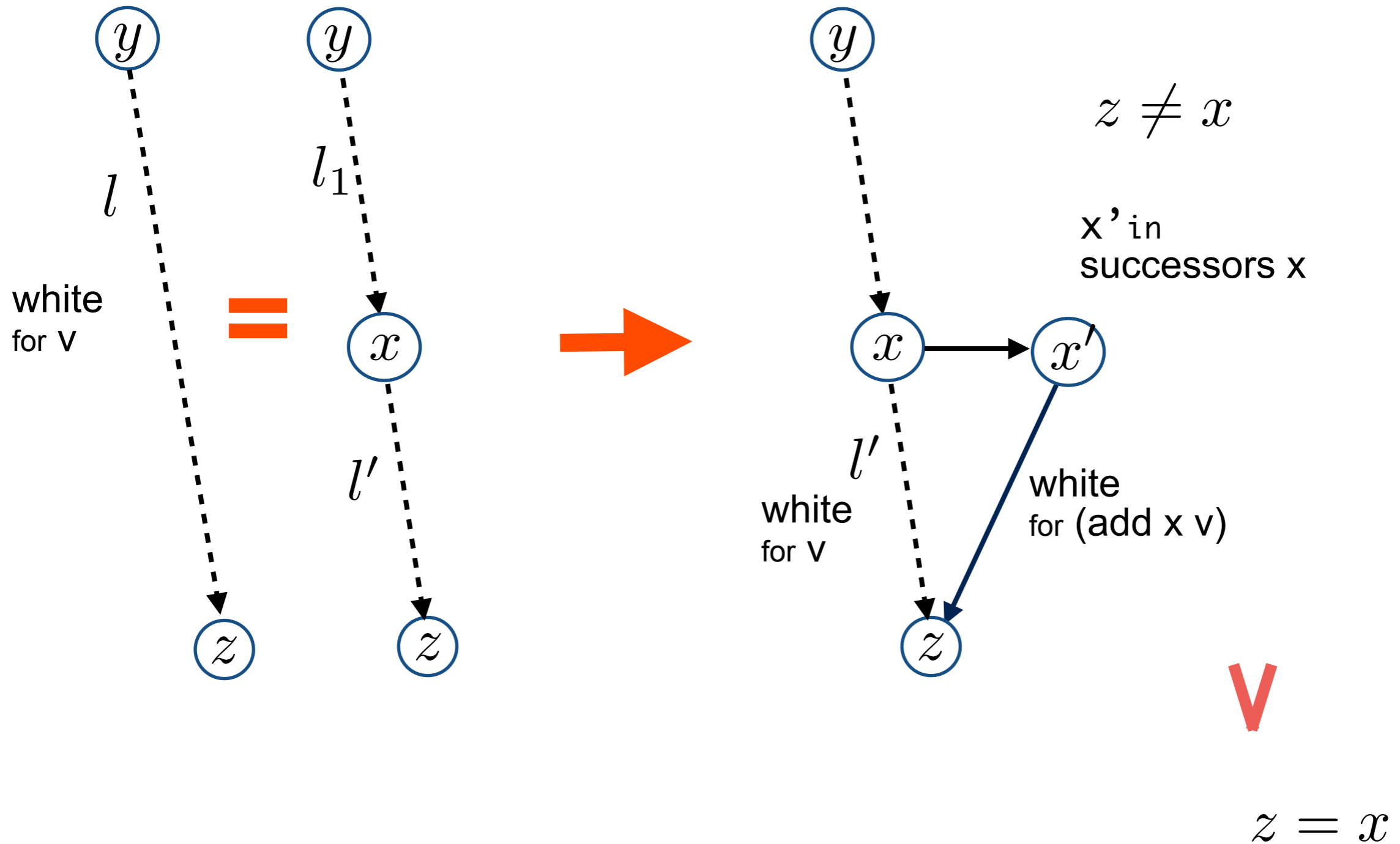
```
predicate white_vertex (x : vertex) (v : set vertex) =  $\neg$  (mem x v)
predicate whitepath (x : vertex) (l : list vertex) (z : vertex) (v : set vertex) =
  path x l z  $\wedge$  ( $\forall y. L.mem y l \rightarrow$  white_vertex y v)  $\wedge$  white_vertex z v
```

```
let rec random_search r v =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {subset v result}
  ensures {forall z y l. mem y r -> whitepath y l z v -> mem z (diff result v)}
  if is_empty r then v else
    let x = choose r in
    let r' = remove x r in
    if mem x v then
      random_search r' v
    else
      let v' = random_search (union r' (successors x)) (add x v) in
      assert {forall z y l. mem y r -> whitepath y l z v -> z <> x ->
        whitepath y l z (add x v) /\
        exists x' l'. mem x' (successors x) /\ whitepath x' l' z (add x v)} ;
      v'
```

```
let random_search_main roots =
  requires {subset roots vertices}
  assert {forall z y l. mem y r -> path y l z -> mem z result}
  random_search roots empty
```



# Random search completeness



# Random search soundness

```
predicate white_vertex (x : vertex) (v : set vertex) =  $\neg$  (mem x v)
predicate whitepath (x : vertex) (l : list vertex) (z : vertex) (v : set vertex) =
  path x l z  $\wedge$  ( $\forall y. L.mem y l \rightarrow$  white_vertex y v)  $\wedge$  white_vertex z v
```

```
let rec random_search r v =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {forall z. mem z (diff result v) -> exists y l. mem y r /\ whitepath y l z v}
  if is_empty r then v else
    let x = choose r in
    let r' = remove x r in
    if mem x v then
      random_search r' v
    else
      let v' = random_search (union r' (successors x)) (add x v) in
      assert {forall z. mem z (diff v' v) -> z <> x ->
        (exists y l. mem y r' /\ (whitepath y l z v)
          \/ (exists l. whitepath x l z v))}
      v'
```

```
let random_search_main roots =
  requires {subset roots vertices}
  assert {forall z. mem z result -> exists y l. mem y r -> path y l z}
  random_search roots empty
```

# Random search soundness

```
predicate white_vertex (x : vertex) (v : set vertex) =  $\neg$  (mem x v)
predicate whitepath (x : vertex) (l : list vertex) (z : vertex) (v : set vertex) =
  path x l z  $\wedge$  ( $\forall y. L.mem y l \rightarrow white\_vertex y v$ )  $\wedge$  white_vertex z v
```

```
let rec random_search r v =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {forall z. mem z (diff result v) -> exists y l. mem y r /\ whitepath y l z v}
  if is_empty r then v else
    let x = choose r in
    let r' = remove x r in
    if mem x v then
      random_search r' v
    else
      let v' = random_search (union r' (successors x)) (add x v) in
      assert {forall z. mem z (diff v' v) -> z <> x ->
        (exists y l. (mem y r' /\ (whitepath y l z v
          by whitepath y l z (add x v) ))
          \/ ((exists l. whitepath x l z v)
            by exists y' l'. mem y' (successors x) /\ whitepath y' l' z v))};
      v'
```

```
let random_search_main roots =
  requires {subset roots vertices}
  assert {forall z. mem z result -> exists y l. mem y r -> path y l z}
  random_search roots empty
```

# Random search soundness

```
predicate white_vertex (x : vertex) (v : set vertex) =  $\neg$  (mem x v)
predicate whitepath (x : vertex) (l : list vertex) (z : vertex) (v : set vertex) =
  path x l z  $\wedge$  ( $\forall y. L.mem y l \rightarrow white\_vertex y v$ )  $\wedge$  white_vertex z v
```

```
let rec random_search r v =
  requires {subset r vertices}
  requires {subset v vertices}
  ensures {forall z. mem z (diff result v) -> exists y l. mem y r /\ whitepath y l z v}
  if is_empty r then v else
    let x = choose r in
    let r' = remove x r in
    if mem x v then
      random_search r' v
    else
      let v' = random_search (union r' (successors x)) (add x v) in
      assert {forall z. mem z (diff v' v) -> z <> x ->
        (exists y l. (mem y r' /\ (whitepath y l z v
          by whitepath y l z (add x v) ))
        \/\ ((exists l. whitepath x l z v)
          by exists y' l'. mem y' (successors x) /\ whitepath y' l' z v))};
      v'
```

```
let random_search_main roots =
  requires {subset roots vertices}
  assert {forall z. mem z result -> exists y l. mem y r -> path y l z}
  random_search roots empty
```



# Other algorithms on graphs

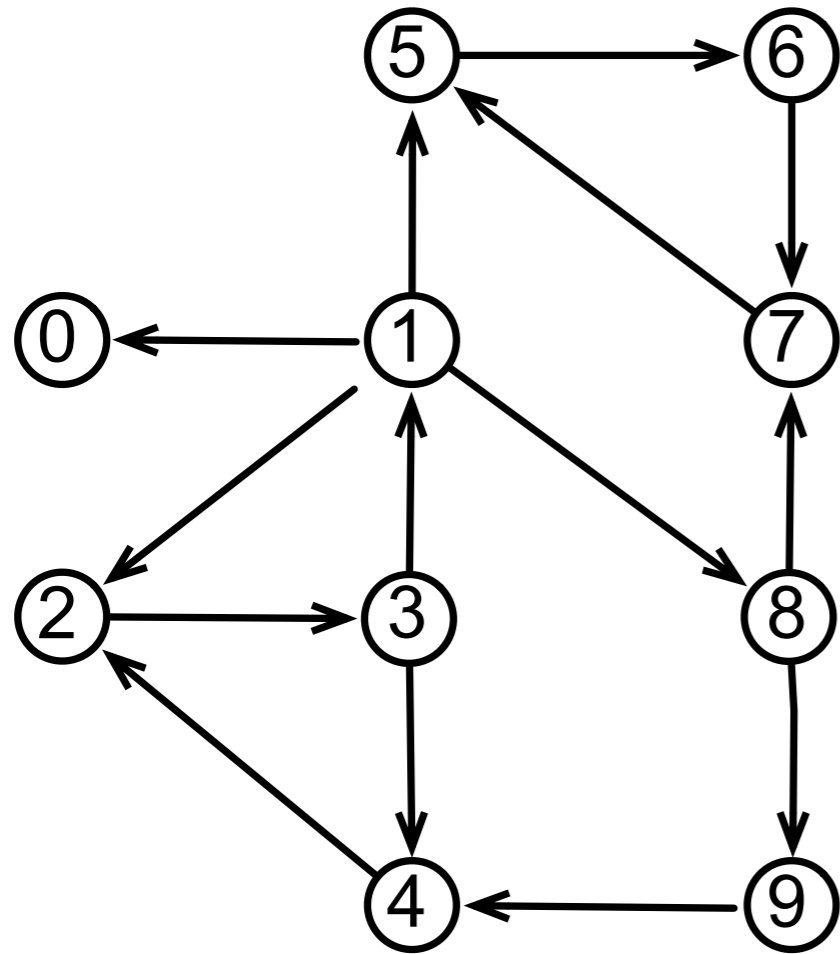
- dfs (recursive), dfs (iterative), bfs
- dfs (imperative)
- dag test
- dfs (recursive with non-black-to-white predicate)
- dfs (undirected graphs with non-white-to-black proof)
- minimum spanning tree (50% done)
- strongly connected components (kosaraju)
- strongly connected components (tarjan)

<http://jeanjacqueslevy.net/why3>

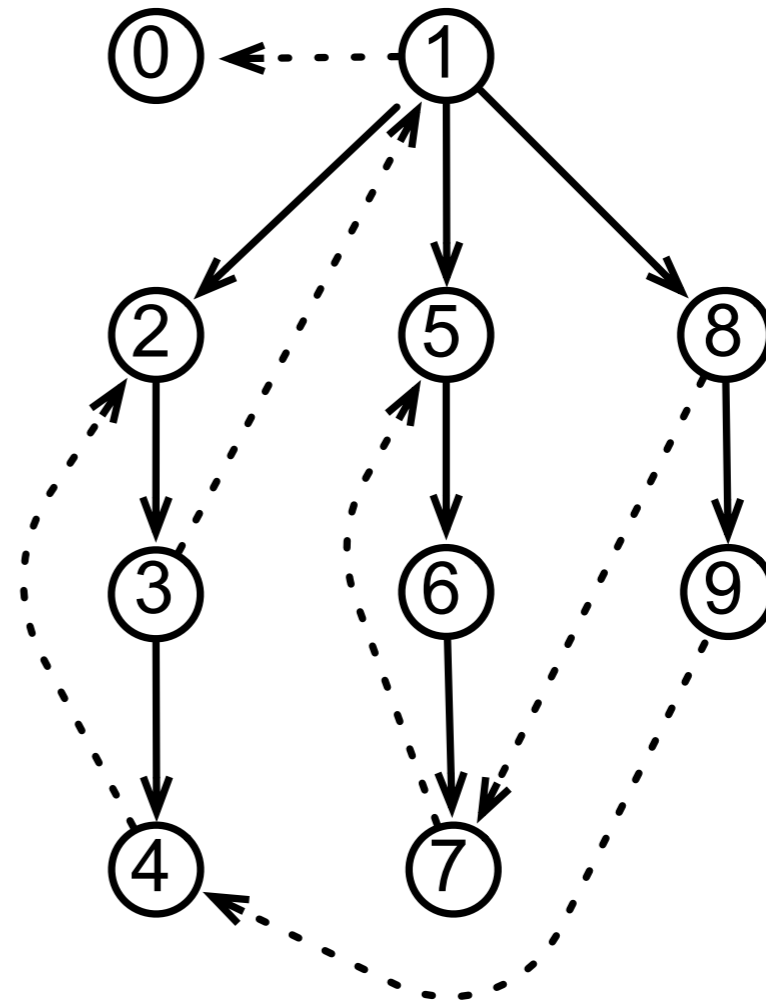
# One-pass linear-time algorithm

[tarjan 1972]

# Depth-first-search

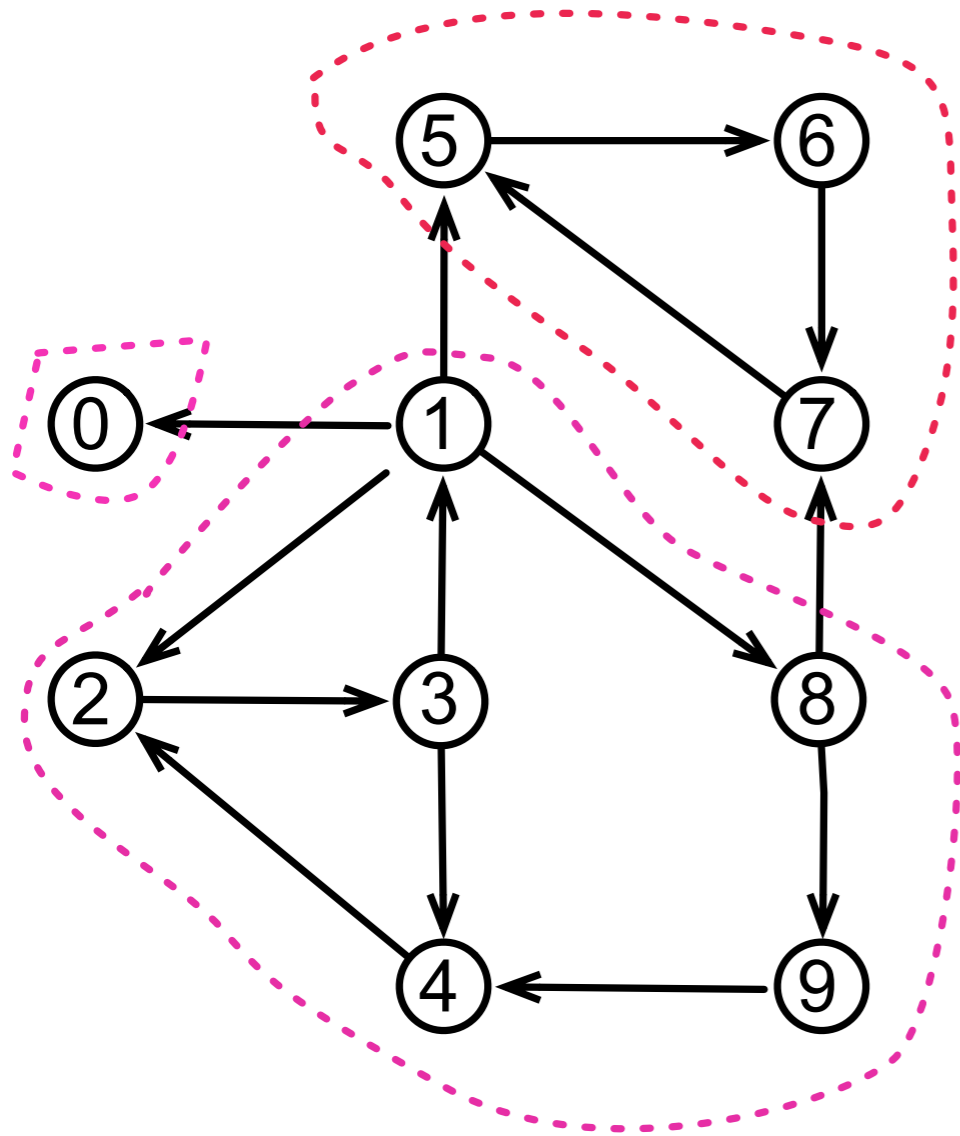


graph

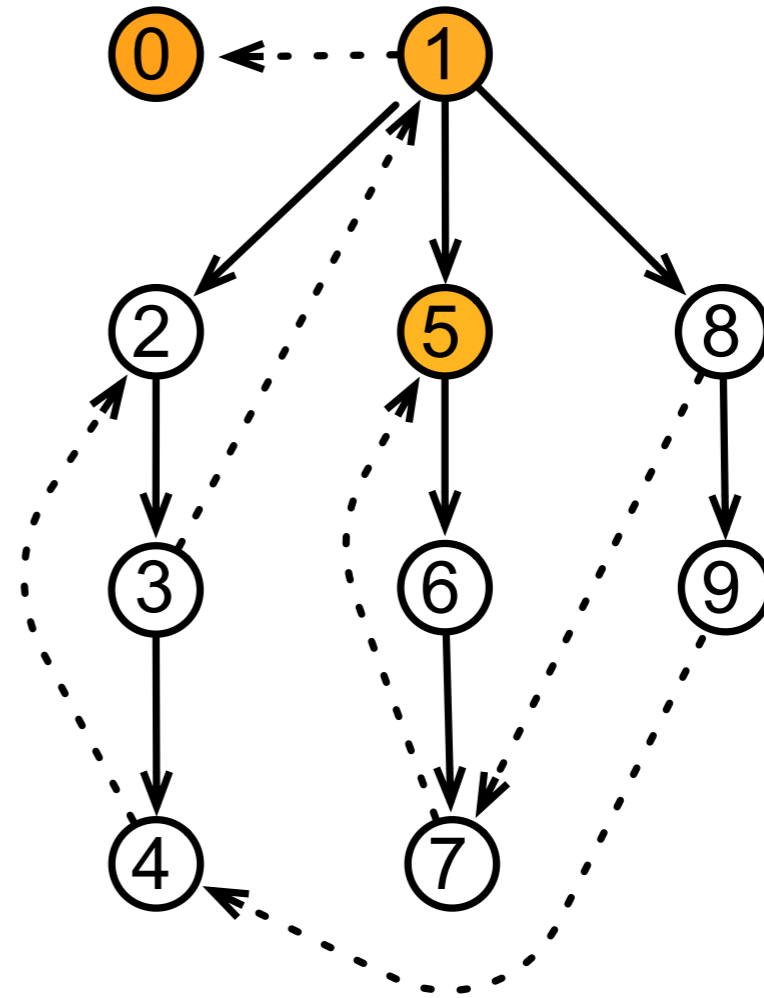


spanning tree (forest)

# The algorithm (1/3)



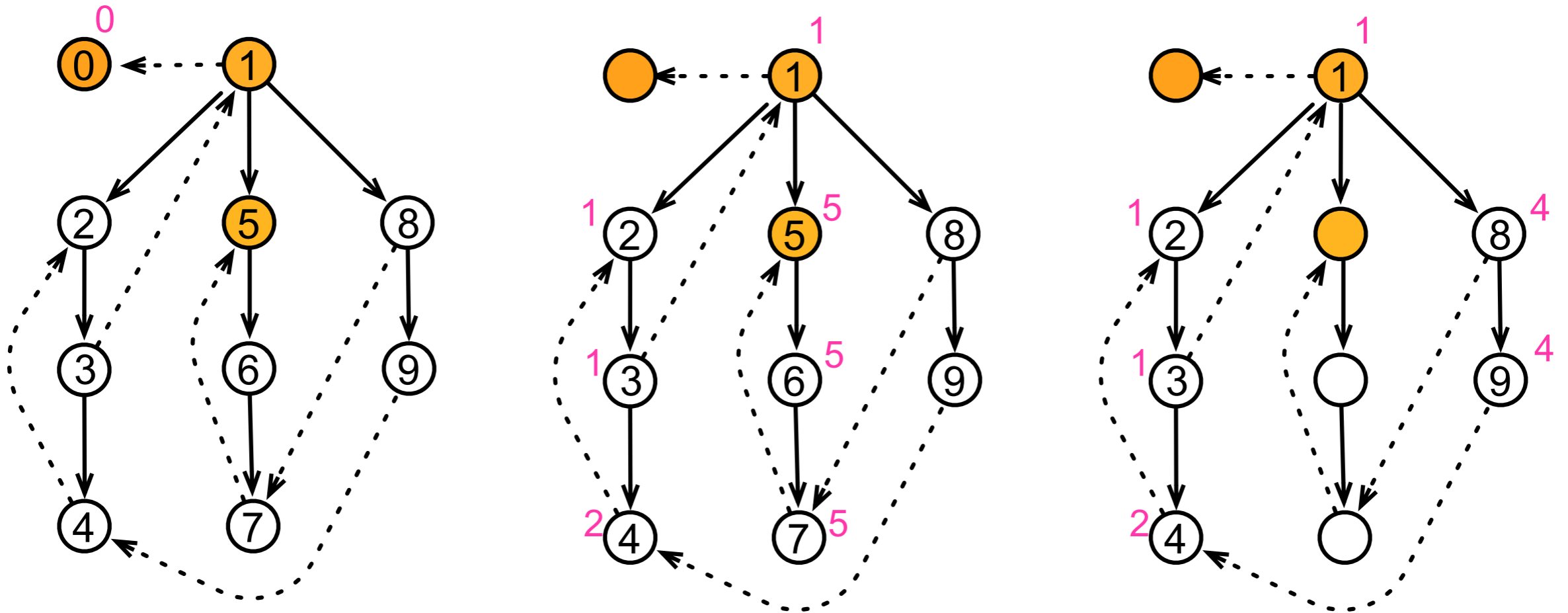
3 SCCs (strongly connected components)



3 vertices are their bases



# The algorithm (2/3)



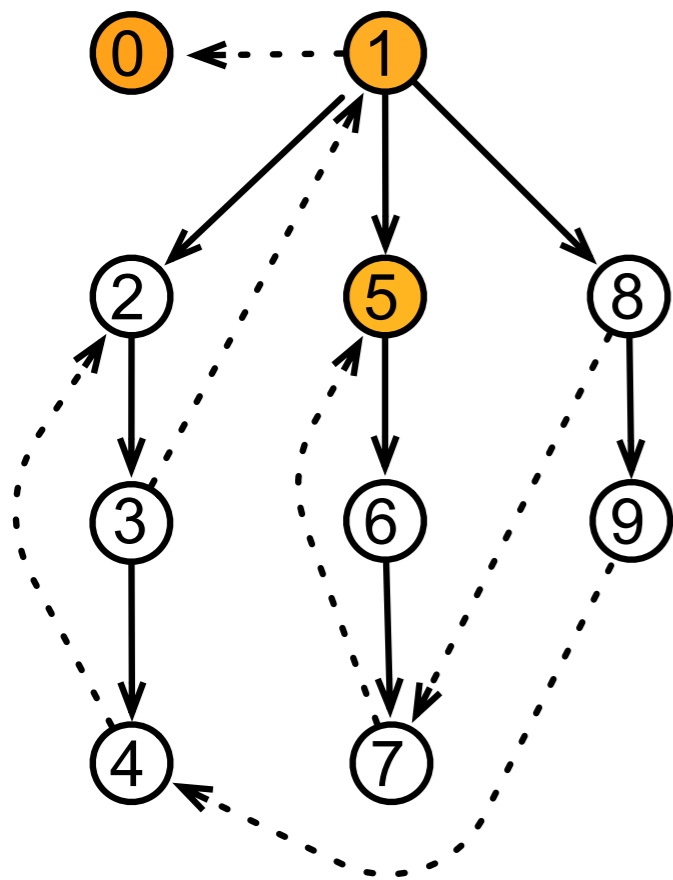
$$LOWLINK(x) = \min ( \{num[x]\} \cup \{num[y] \mid x \xrightarrow{*} \cdots \rightarrow y$$

$$\wedge x \text{ and } y \text{ are in same}$$

$$\text{connected component} \} )$$

# The algorithm (3/3)

successive values of the working stack



0	1	1	1	1	1	1	1	1	1	0
		2	2	2	2	2	2	2	2	1
			3	3	3	3	3	3	3	2
				4	4	4	4	4	4	3
					5	5	5	8	8	4
						6	6	9	9	5
							7			6

0  
 1  
 2  
 3  
 4  
 5  
 6  
 ↓ increasing rank

# The program

```
let rec printSCC (x: int) (s: stack int)
  (num: array int) (sn: ref int) =
  Stack.push x s;
  num[x] ← !sn; sn := !sn + 1;
  let low = ref num[x] in
  foreach y in (successors x) do
    let m = if num[y] = -1
      then printSCC y s num sn
      else num[y] in
    low := Math.min m !low
  done;
```

```
if !low = num[x] then begin
  repeat
    let y = Stack.pop s in
    Printf.printf "%d " y;
    num[y] ← max_int;
    if y = x then break;
  done;
  Printf.printf "\n";
  low := max_int;
end;
return !low;
```

- print each component on a line

Imperative style

# Proof in algorithms books (1/2)


- consider the spanning trees (forest)
- tree structure of strongly connected components
- 2-3 lemmas about ancestors in spanning trees

*LEMMA 10. Let  $v$  and  $w$  be vertices in  $G$  which lie in the same strongly connected component. Let  $F$  be a spanning forest of  $G$  generated by repeated depth-first search. Then  $v$  and  $w$  have a common ancestor in  $F$ . Further, if  $u$  is the highest numbered common ancestor of  $v$  and  $w$ , then  $u$  lies in the same strongly connected component as  $v$  and  $w$ .*

$$\text{LOWLINK}(x) = \min \left( \{ \text{num}[x] \} \cup \{ \text{num}[y] \mid x \xrightarrow{*} \hookrightarrow y \right. \\ \left. \wedge x \text{ and } y \text{ are in same} \right. \\ \left. \text{connected component} \} \right)$$

*LEMMA 12. Let  $G$  be a directed graph with LOWLINK defined as above relative to some spanning forest  $F$  of  $G$  generated by depth-first search. Then  $v$  is the root of some strongly connected component of  $G$  if and only if  $\text{LOWLINK}(v) = v$ .*

# Proof in algorithms book (2/2)

- give the program
- proof  program
- that part of the proof is very informal

# Our program (1/3)

```
let rec dfs1 x e =
  let n = e.sn in
  let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
  let (s2, s3) = split x e1.stack in
  if n1 < n then (n1, e1) else
    (max_int(), {stack = s3; sccs = add (elements s2) e1.sccs;
                 sn = e1.sn; num = set_max_int s2 e1.num})

with dfs roots e = if is_empty roots then (max_int(), 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)

let tarjan () =
  let e0 = {stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
  let (_, e') = dfs vertices e0 in e'.sccs
```

e1.stack

s3

x

s2

returns *LOWLINK*(x) and new environment

Functional programming

# Formal proof

using Why3

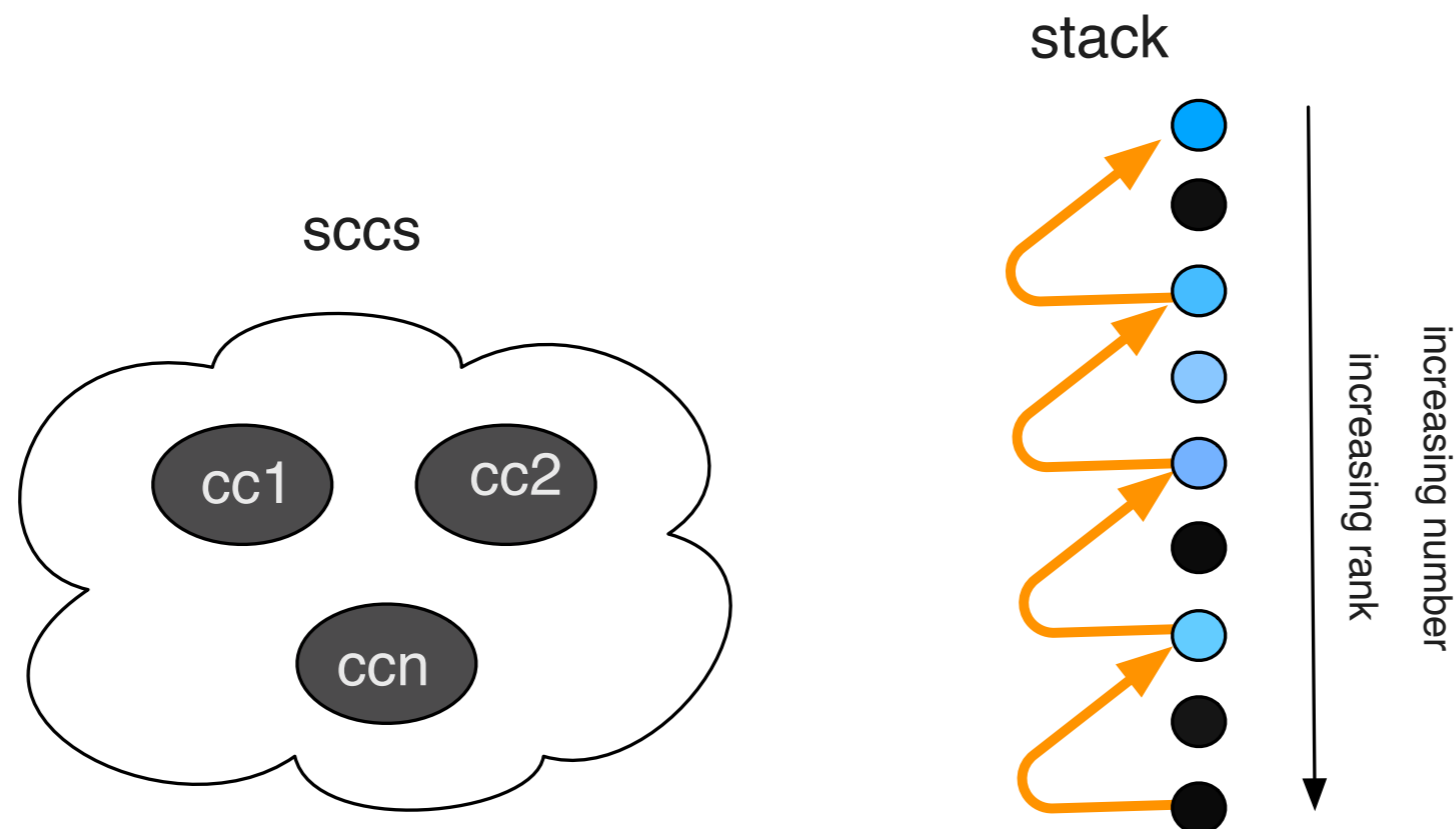
# Plan of proof (1/2)

- define **reachability** in graphs and SCCs
  - prove a few lemmas about positions in stacks (**ranks**)
  - define **invariants** on environments
  - give **pre-post conditions** for functions
  - add a few intermediate **assertions** in function bodies
- 
- avoid paths, prefer edges



# Plan of proof (2/2)

- vertices have colors
  - white = unvisited
  - gray = being visited
  - black = visited
- invariant on environment



vertex in stack reaches all vertices with higher rank

# Invariants

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

# Invariants

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

```
predicate wf_color (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in  
  subset (union g b) vertices /\  
  inter b g == empty /\  
  elements s == union g (diff b (set_of ccs)) /\  
  subset (set_of ccs) b
```

# Invariants

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

```
predicate wf_color (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in  
  subset (union g b) vertices /\  
  inter b g == empty /\  
  elements s == union g (diff b (set_of ccs)) /\  
  subset (set_of ccs) b
```

```
predicate wf_num (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs; sn = n; num = f} = e in  
  (forall x. -1 <= f[x] < n <= max_int() \ / f[x] = max_int()) /\  
  n = cardinal (union g b) /\  
  (forall x. f[x] = max_int() <-> mem x (set_of ccs)) /\  
  (forall x. f[x] = -1 <-> not mem x (union g b)) /\  
  (forall x y. lmem x s -> lmem y s -> f[x] < f[y] <-> rank x s < rank y s)
```

# Invariants

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

```
predicate wf_color (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in  
  subset (union g b) vertices /\  
  inter b g == empty /\  
  elements s == union g (diff b (set_of ccs)) /\  
  subset (set_of ccs) b
```

```
predicate wf_num (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs; sn = n; num = f} = e in  
  (forall x. -1 <= f[x] < n <= max_int() \ / f[x] = max_int()) /\  
  n = cardinal (union g b) /\  
  (forall x. f[x] = max_int() <-> mem x (set_of ccs)) /\  
  (forall x. f[x] = -1 <-> not mem x (union g b)) /\  
  (forall x y. lmem x s -> lmem y s -> f[x] < f[y] <-> rank x s < rank y s)
```

```
predicate no_black_to_white (blacks grays: set vertex) =  
  forall x x'. edge x x' -> mem x blacks -> mem x' (union blacks grays)
```

# Invariants

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

```
predicate wf_color (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs} = e in  
  subset (union g b) vertices /\  
  inter b g == empty /\  
  elements s == union g (diff b (set_of ccs)) /\  
  subset (set_of ccs) b
```

```
predicate wf_num (e: env) =  
  let {stack = s; blacks = b; grays = g; sccs = ccs; sn = n; num = f} = e in  
  (forall x. -1 <= f[x] < n <= max_int() \/\ f[x] = max_int()) /\  
  n = cardinal (union g b) /\  
  (forall x. f[x] = max_int() <-> mem x (set_of ccs)) /\  
  (forall x. f[x] = -1 <-> not mem x (union g b)) /\  
  (forall x y. lmem x s -> lmem y s -> f[x] < f[y] <-> rank x s < rank y s)
```

```
predicate no_black_to_white (blacks grays: set vertex) =  
  forall x x'. edge x x' -> mem x blacks -> mem x' (union blacks grays)
```

```
predicate wf_env (e: env) = let {stack = s; blacks = b; grays = g} = e in  
  wf_color e /\ wf_num e /\  
  no_black_to_white b g /\ simplelist s /\  
  (forall x y. mem x g -> lmem y s -> rank x s <= rank y s -> reachable x y) /\  
  (forall y. lmem y s -> exists x. mem x g /\ rank x s <= rank y s /\ reachable y x)
```

# Pre/Post-conditions

```
let rec dfs1 x e =  
requires {mem x vertices} (* R1 *)  
requires {access_to e.grays x} (* R2 *)  
requires {not mem x (union e.blacks e.grays)} (* R3 *)
```

# Pre/Post-conditions

```
let rec dfs1 x e =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
(* invariants *)
requires {wf_env e} (* I1a *)
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
returns {(_, e') -> wf_env e'} (* I1b *)
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
```



# Pre/Post-conditions

```
let rec dfs1 x e =
  requires {mem x vertices} (* R1 *)
  requires {access_to e.grays x} (* R2 *)
  requires {not mem x (union e.blacks e.grays)} (* R3 *)
  (* invariants *)
  requires {wf_env e} (* I1a *)
  requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
  returns {(_, e') -> wf_env e'} (* I1b *)
  returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)

  (* monotony *)
  returns {(_, e') -> subenv e e'}
```

# Pre/Post-conditions

```
let rec dfs1 x e =  
requires {mem x vertices} (* R1 *)  
requires {access_to e.grays x} (* R2 *)  
requires {not mem x (union e.blacks e.grays)} (* R3 *)  
(* invariants *)  
requires {wf_env e} (* I1a *)  
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)  
returns {(_, e') -> wf_env e'} (* I1b *)  
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
```

```
(* monotony *)  
returns {(_, e') -> subenv e e'}
```

$$e.sccs \subseteq e'.sccs$$

$$e.blacks \subseteq e'.blacks$$

$$e.grays = e'.grays$$

e'.stack

e.stack



# Pre/Post-conditions

```
let rec dfs1 x e =
requires {mem x vertices} (* R1 *)
requires {access_to e.grays x} (* R2 *)
requires {not mem x (union e.blacks e.grays)} (* R3 *)
(* invariants *)
requires {wf_env e} (* I1a *)
requires {forall cc. mem cc e.sccs <-> subset cc e.blacks /\ is_scc cc} (* I2a *)
returns {(_, e') -> wf_env e'} (* I1b *)
returns {(_, e') -> forall cc. mem cc e'.sccs <-> subset cc e'.blacks /\ is_scc cc} (* I2b *)
(* post-cond *)
returns {(n, e') -> n <= e'.num[x]} (* PC1 *)
returns {(n, e') -> n = max_int() \/ num_of_reachable n x e'} (* PC2 *)
returns {(n, e') -> forall y. xedge_to e'.stack e.stack y -> n <= e'.num[y]} (* PC3 *)
returns {(_, e') -> mem x e'.blacks} (* PC4 *)
(* monotony *)
returns {(_, e') -> subenv e e'}
```

$$e.sccs \subseteq e'.sccs$$

$$e.blacks \subseteq e'.blacks$$

$$e.grays = e'.grays$$

e'.stack

e.stack



X

# Assertions

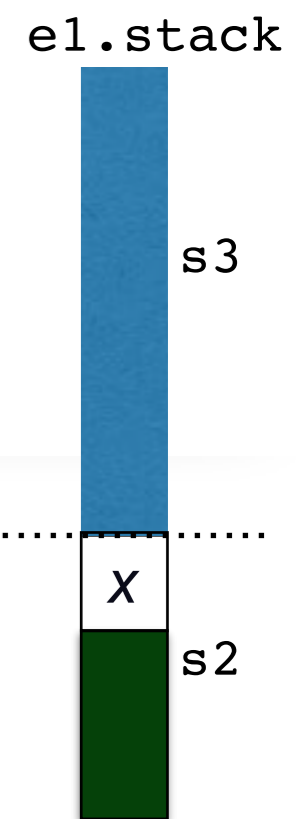
```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
```

```
if n1 < n then begin
```

```
  (n1, add_blacks x e1) end
```

```
else begin
```

```
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
    stack = s3; sccs = add (elements s2) e1.sccs;
    sn = e1.sn; num = set_max_int s2 e1.num}) end
```



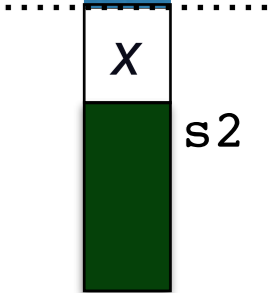
[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

# Assertions

e1.stack

s3

s2



```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subsc (elements s2)};
if n1 < n then begin

  (n1, add_blacks x e1) end
else begin
```

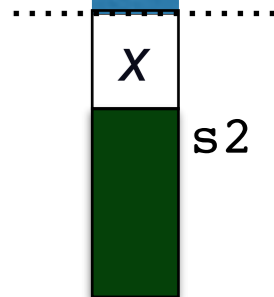
```
(max_int(), {blacks = add x e1.blacks; grays = e.grays;
  stack = s3; sccs = add (elements s2) e1.sccs;
  sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

# Assertions

e1.stack

```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subsc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else begin
```

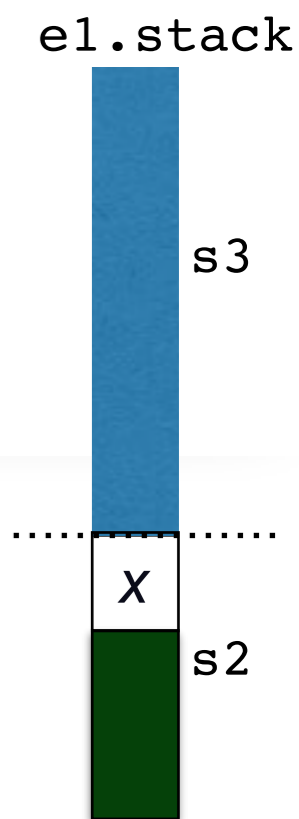


```
(max_int(), {blacks = add x e1.blacks; grays = e.grays;
  stack = s3; sccs = add (elements s2) e1.sccs;
  sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

# Assertions

```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_scc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
    stack = s3; sccs = add (elements s2) e1.sccs;
    sn = e1.sn; num = set_max_int s2 e1.num}) end
```



[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

# Assertions

e1.stack

s3

s2

x

```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_scc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
    stack = s3; sccs = add (elements s2) e1.sccs;
    sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

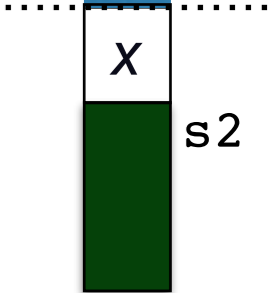


# Assertions

e1.stack

s3

s2



```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subsc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
    stack = s3; sccs = add (elements s2) e1.sccs;
    sn = e1.sn; num = set_max_int s2 e1.num}) end
```

[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

# Assertions

e1.stack

s3

x

s2

```
let n = e.sn in
let (n1, e1) =
  dfs' (successors x) (add_stack_incr x e) in
let (s2, s3) = split x e1.stack in
assert {is_last x s2 /\ s3 = e.stack /\ subset (elements s2) (add x e1.blacks)};
assert {is_subsc (elements s2)};
if n1 < n then begin
  assert {exists y. mem y e.grays /\ lmem y e1.stack /\ e1.num[y] < e1.num[x] /\ reachable x y};
  (n1, add_blacks x e1) end
else begin
  assert {forall y. in_same_scc y x -> lmem y s2};
  assert {is_scc (elements s2)};
  assert {inter e.grays (elements s2) = empty by inter e.grays (elements s2) == empty};
  (max_int(), {blacks = add x e1.blacks; grays = e.grays;
    stack = s3; sccs = add (elements s2) e1.sccs;
    sn = e1.sn; num = set_max_int s2 e1.num}) end
```

Coq

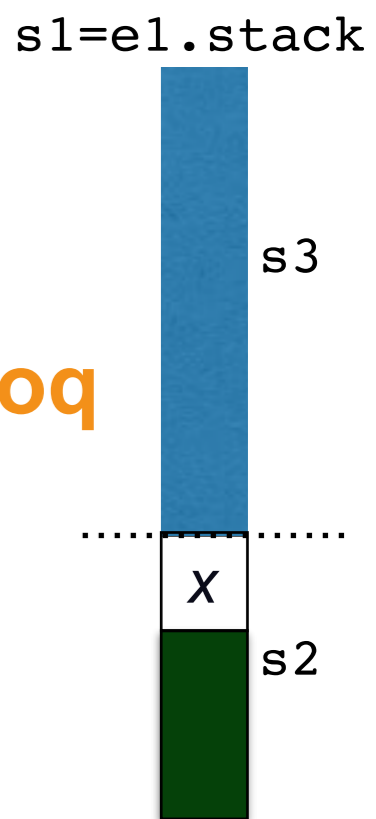
[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]

# Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq

- proof by contradiction:  $\exists y, \text{in\_same\_scc } y \ x \wedge y \notin s2$
- $\exists x' y', \text{reachable } x \ x' \wedge \text{edge } x' \ y' \wedge \text{reachable } y' \ y \wedge x' \in s2 \wedge y' \notin s2$



# Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq

- proof by contradiction:  $\exists y, \text{in\_same\_scc } y \ x \wedge y \notin s2$
- $\exists x' y', \text{reachable } x \ x' \wedge \text{edge } x' \ y' \wedge \text{reachable } y' \ y \wedge x' \in s2 \wedge y' \notin s2$
- 3 cases:

s1=e1.stack

s3

x

s2

# Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq

- proof by contradiction:  $\exists y, \text{in\_same\_scc } y \ x \wedge y \notin s2$
- $\exists x' y', \text{reachable } x \ x' \wedge \text{edge } x' \ y' \wedge \text{reachable } y' \ y \wedge x' \in s2 \wedge y' \notin s2$
- 3 cases:

[1]  $y'$  is white

$x' = x$  then  $y' \in \text{successors } x \rightarrow y'$  is black

$x' \neq x$  then  $x'$  is black  $\rightarrow \neg \text{no\_black\_to\_white } b1 \ g1$

s1=e1.stack

s3

x

s2

# Assertions

```
assert {forall y. in_same_scc y x -> lmem y s2};
```

Coq

- proof by contradiction:  $\exists y, \text{in\_same\_scc } y \ x \wedge y \notin s2$
- $\exists x'y', \text{reachable } x \ x' \wedge \text{edge } x' \ y' \wedge \text{reachable } y' \ y \wedge x' \in s2 \wedge y' \notin s2$
- 3 cases:

[1]  $y'$  is white

$x' = x$  then  $y' \in \text{successors } x \rightarrow y'$  is black

$x' \neq x$  then  $x'$  is black  $\rightarrow \neg \text{no\_black\_to\_white } b1 \ g1$

[2]  $y' \in e1.sccs$  then  $\text{in\_same\_scc } y' \ x \rightarrow x$  is black

s1=e1.stack

s3

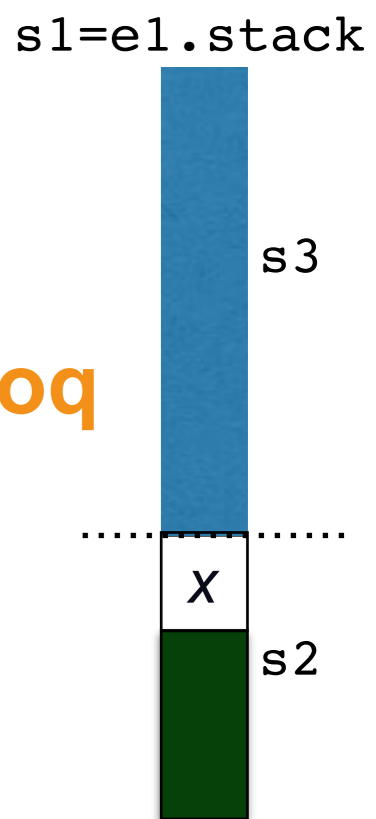
x

s2

# Assertions

`assert {forall y. in_same_scc y x -> lmem y s2};`

Coq



- proof by contradiction:  $\exists y, \text{ in\_same\_scc } y \ x \wedge y \notin s2$
- $\exists x'y', \text{ reachable } x \ x' \wedge \text{ edge } x' \ y' \wedge \text{ reachable } y' \ y \wedge x' \in s2 \wedge y' \notin s2$
- 3 cases:

[1]  $y'$  is white

$x' = x$  then  $y' \in \text{successors } x \rightarrow y'$  is black

$x' \neq x$  then  $x'$  is black  $\rightarrow \neg \text{no\_black\_to\_white } b1 \ g1$

[2]  $y' \in e1.sccs$  then  $\text{in\_same\_scc } y' \ x \rightarrow x$  is black

[3]  $y' \in s3 \rightarrow \text{rank } y' \ s1 < \text{rank } x \ s1 \rightarrow e1.\text{num}[y'] < e1.\text{num}[x] = e.\text{num}[x] = n$

$x' = x$  then  $y' \in \text{successors } x \rightarrow n1 \leq e1.\text{num}[y']$

$x' \neq x$  then  $\text{xedge\_to } s1 \ (\text{Cons } x \ s3) \ y' \rightarrow n1 \leq e1.\text{num}[y']$

# Proof stats

provers	Alt- Ergo	CVC3	CVC4	Coq	E- prover	Spass	Yices	Z3	all	#VC	#PO
38 lemmas	2.35	0.23	5.79		0.66	0.75	0.21		9.99	77	38
split	0.09	0.2							0.29	6	6
add_stack_incr	0.01								0.01	1	1
add_blacks	0.01								0.01	1	1
set_max_int	0.02								0.02	1	1
dfs1	53.52	12.88	36.39	3.06	28.06			9.01	142.92	218	24
dfs	4.6	0.23	11.63					0.31	16.77	51	35
tarjan	0.44								0.44	16	6
total	61.04	13.54	53.81	3.06	28.72	0.75	0.21	9.32	170.45	371	112

[ <http://jeanjacqueslevy.net/why3/graph/abs/scct/1-7/scc.html> ]



# Other systems

# Coq / ssreflect

[cyril cohen, laurent théry, JJL]

- port in 1 week
- graphs and finite sets already in mathematical components
- problems with termination (hacky & higher-order)
- 920 lines

[<http://github.com/CohenCyril/tarjan>]

# Isabelle / HOL

[stephan merz]

- port in 1 month
- use many strategies (metis, blast, sledgehammer)
- still problems with proving termination
- 31 pages

[<http://jeanjacqueslevy.net/why3/graph/abs/scct/isa/Tarjan.pdf>]

# Fstar

[kenji maillard, catalin hritcu]

- start discuss with them
- Z3 single automatic prover
- ??

The background features four large, overlapping circles in vibrant colors: yellow, green, blue, and red. The circles are arranged in a way that they partially overlap each other, creating a dynamic and colorful composition. The word "Conclusion" is centered over the intersection of the yellow, green, and red circles.

Conclusion

# Future works

- library for formal proofs on graphs
- other graph algorithms
- **beyond** graphs ...
- teaching formal methods on **test cases**
- **imperative** programs
- **Frama-C** embedded programs written in C
- readable formal proofs ?

[<http://jeanjacqueslevy.net/why3>]