

Strongly Connected Components in graphs, formal proof of Tarjan1972 algorithm

Jean-Jacques Levy, Ran Chen

▶ To cite this version:

Jean-Jacques Levy, Ran Chen. Strongly Connected Components in graphs, formal proof of Tarjan1972 algorithm. Groupe de travail LTP du GDR GPL , Nov 2016, Orsay, France. 2016. hal-01422227

HAL Id: hal-01422227 https://inria.hal.science/hal-01422227v1

Submitted on 24 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Strongly Connected Components in graphs, formal proof of Tarjan1972 algorithm

jean-jacques.levy@inria.fr

LTP, PCRI, 28-11-2016

Plan

- motivation
- algorithm
- pre-/post-conditions
- imperative programs
- conclusion

.. joint work (in progress) with Ran Chen

Motivation

- nice algorithms should have simple formal proofs
- to be fully published in articles or journals
- how to publish formal proofs ?
- Coq proofs seem to me unreadable (and by normal human ?)
- Why3 allows mix of automatic and interactive proofs
- first-order logic is easy to understand
- algorithms on graphs = a good testbed

A one-pass lineartime algorithm

The algorithm (1/2)



- depth-first search algorithm
- with pushing non visited vertices into a working stack
- and computing oldest vertex reachable by at most a single « back-edge »
- when that oldest vertex is equal to currently visited vertex, a new strongly connected component is in the working stack on top of current vertex.
- then pop working stack until currently visited vertex

The algorithm (2/2)



les valeurs successives de la pile



The algorithm (2/2)



les valeurs successives de la pile



Proof in algorithms book

- consider the spanning trees (forrest)
- Christmas 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.

• give the program



The program (1/3)

- a functional version with lists and finite sets
- the working stack is a list

```
function rank (x: vertex) (s: list vertex): int =
  match s with
  | Nil → max_int()
  | Cons y s' → if x = y && not (lmem x s') then length s' else rank x s'
  end
```

```
function max_int (): int = cardinal vertices
```

```
let rec split (x : \alpha) (s: list \alpha) : (list \alpha, list \alpha) =
returns{(s1, s2) \rightarrow s1 ++ s2 = s}
returns{(s1, _) \rightarrow lmem x s \rightarrow is_last_of x s1}
match s with
| Nil \rightarrow (Nil, Nil)
| Cons y s' \rightarrow if x = y then (Cons x Nil, s') else
let (s1', s2) = split x s' in
((Cons y s1'), s2)
end
```

The program (2/3)

• blacks, grays are sets of vertices; sccs is a set of sets of vertices

• naming conventions:

x, y, z for vertices; *b* for black sets; *s* for stacks;

cc for connected components;

sccs for sets of connected components

```
let rec dfs1 x blacks (ghost grays) stack sccs =
  let m = rank x (Cons x stack) in
  let (m1, b1, s1, sccs1) =
    dfs' (successors x) blacks (add x grays) (Cons x stack) sccs in
    if m1 ≥ m then
    let (s2, s3) = split x s1 in
    (max_int(), add x b1, s3, add (elements s2) sccs1)
  else
    (m1, add x b1, s1, sccs1)
```

The program (3/3)

```
with dfs' roots blacks (ghost grays) stack sccs =
  if is_empty roots then
    (max_int(), blacks, stack, sccs)
  else
    let x = choose roots in
    let roots' = remove x roots in
    let (m1, b1, s1, sccs1) =
      if lmem x stack then
        (rank x stack, blacks, stack, sccs)
      else if mem x blacks then
        (max_int(), blacks, stack, sccs)
      else
        dfs1 x blacks grays stack sccs in
    let (m2, b2, s2, sccs2) =
      dfs' roots' b1 grays s1 sccs1 in
    (min m1 m2, b2, s2, sccs2)
```

Pre-/Post-conditions

Pre/Post-conditions (1/3)

let rec dfs1 x blacks (ghost grays) stack sccs =
requires{mem x vertices} (* R1 *)
requires{access_to grays x} (* R2 *)
requires{not mem x (union blacks grays)} (* R3 *)

(* monotony *) returns{(_, b, s, _) $\rightarrow \exists$ s'. s = s' ++ stack \land subset (elements s') b} (* M1 *) returns{(_, b, _, _) \rightarrow subset blacks b} (* M2 *) returns{(_, _, _, sccs_n) \rightarrow subset sccs sccs_n} (* M3 *)

Pre/Post-conditions (2/3)



Pre/Post-conditions (3/3)

with dfs' roots blacks (ghost grays) stack sccs = requires {subset roots vertices} (* R1 *) requires { $\forall x. mem x roots \rightarrow access_to grays x$ } (* R2 *)

```
(* post conditions *)

returns{(_, b, _, _) \rightarrow subset roots (union b grays)} (* E1 *)

returns{(m, _, s, _) \rightarrow \forall x. mem x roots \rightarrow m \leq rank x s} (* E2 *)

returns{(m, _, s, _) \rightarrow m = max_int() \lor \exists x. mem x roots \land rank_of_reachable m x s}

returns{(m, _, s, _) \rightarrow \forall y. crossedgeto s y stack \rightarrow m \leq rank y stack} (* E4 *)

(* monotony *)

returns{(_, b, s, _) \rightarrow \exists s'. s = s' ++ stack \land subset (elements s') b} (* M1 *)

returns{(_, b, _, _) <math>\rightarrow subset blacks b} (* M2 *)

returns{(_, _, _, sccs_n) \rightarrow subset sccs sccs_n} (* M3 *)
```

Graphs

type vertex constant vertices: set vertex function successors vertex : set vertex axiom successors_vertices: ∀x. mem x vertices → subset (successors x) vertices predicate edge (x y: vertex) = mem x vertices ∧ mem y (successors x)

Paths

```
inductive path vertex (list vertex) vertex =
  | Path_empty:
       \forall x: vertex. path x Nil x
  | Path_cons:
       \forall x \ y \ z: vertex, 1: list vertex.
       edge x y \rightarrow path y l z \rightarrow path x (Cons x l) z
predicate reachable (x z: vertex) =
  \exists 1. path x 1 z
predicate in_same_scc (x z: vertex) =
   reachable x z \wedge reachable z x
predicate is_subscc (s: set vertex) =
   \forall x z. mem x s \rightarrow mem z s \rightarrow in_same_scc x z
predicate is_scc (s: set vertex) =
   is_subscc s \land (\foralls'. subset s s' \rightarrow is_subscc s' \rightarrow s == s')
```

Invariants (1/4)

predicate no_black_to_white (blacks grays: set vertex) = $\forall x x'$. edge x x' \rightarrow mem x blacks \rightarrow mem x' (union blacks grays)

```
predicate wff_color (blacks grays: set vertex) (s: list vertex)
  (sccs: set (set vertex)) =
    inter blacks grays = empty ^
    (elements s) == union grays (diff blacks (set_of sccs)) ^
    (subset (set_of sccs) blacks) ^
    no_black_to_white blacks grays
```

```
blacks \bigcap grays = \emptyset
elements s = grays \bigcup blacks - (set_of sccs)
(set_of sccs) \subseteq blacks
```





Invariants (3/4)

```
predicate wff_stack (blacks grays: set vertex) (s: list vertex)
  (sccs: set (set vertex)) =
```

```
wff_color blacks grays s sccs \land
simplelist s \land
subset (elements s) vertices \land
```

```
(\forall x y. mem x grays \rightarrow lmem y s \rightarrow rank x s \leq rank y s \rightarrow reachable x y) \land
```

```
(\forall y. \text{lmem } y \ s \rightarrow \exists x. \text{ mem } x \ grays \land rank \ x \ s \le rank \ y \ s \land reachable \ y \ x)
```

Invariants (4/4)



```
let m = rank x (Cons x stack) in
let (m1, b1, s1, sccs1) =
dfs' (successors x) blacks (add x grays) (Cons x stack) sccs in
```

```
if m1 ≥ m then begin
  let (s2, s3) = split x s1 in
  assert{s3 = stack};
  assert{subset (elements s2) (add x b1)};
  assert{is_subscc (elements s2) ∧ mem x (elements s2)};
  assert{∀y. in_same_scc y x → mem y (elements s2)};
  assert{is_scc (elements s2)};
```

(max_int(), add x b1, s3, add (elements s2) sccs1) end
else begin

```
(m1, add x b1, s1, sccs1) end
```

 $assert{\forall y. in_same_scc y x \rightarrow mem y (elements s2)};$

- Coq proof: there exists x', y' with $x' \in s2 \land y' \not\in s2 \land$ edge x' y'
 - $y' \in s3 = \text{stack}$ x' = x impossible because $m1 \leq \text{rank}$ y' s1 < rank x s1 $x' \neq x$ impossible because crossedge
 - $y' \in \mathbf{sccs}$ impossible because sccs disjoint from stack

y' is white

x' = x impossible because successors are black $x' \neq x$ impossible because no black to white

Pre/Post-conditions (1/3)

```
let rec dfs1 x blacks (ghost grays) stack sccs =
requires {mem x vertices} (* R1 *)
requires {access_to grays x} (* R2 *)
requires {not mem x (union blacks grays)} (* R3 *)
(* invariants *)
requires {wff_stack blacks grays stack sccs} (* I1a *)
requires {\forall cc. mem cc sccs \leftrightarrow subset cc blacks \land is_scc cc} (* I2a *)
returns {(_, b, s, sccs_n) \rightarrow wff_stack b grays s sccs_n} (* 11b *)
returns {(_, b, _, sccs_n) \rightarrow \forall cc. mem cc sccs_n \leftrightarrow subset cc b \land is_scc cc} (* 12b *)
(* post conditions *)
returns {(_, b, _, _) \rightarrow mem x b} (* E1 *)
returns {(m, _, s, _) \rightarrow m \leq rank x s} (* E2 *)
returns {(m, _, s, _) \rightarrow m = max_int() \vee rank_of_reachable m x s} (* E3 *)
returns {(m, _, s, _) \rightarrow \forall y. crossedgeto s y stack \rightarrow m \leq rank y stack} (* E4 *)
(* monotony *)
returns {(_, b, s, _) \rightarrow \existss'. s = s' ++ stack \land subset (elements s') b} (* M1 *)
returns {(_, b, _, _) \rightarrow subset blacks b} (* M2 *)
returns {(_, _, _, sccs_n) \rightarrow subset sccs sccs_n} (* M3 *)
```

Towards imperative program

let rec dfs1 x blacks (ghost grays) stack sccs(sn num)= requires {sn = cardinal (union grays blacks) \land subset (union grays blacks) vertices} (* invariants *) requires {wff_num sn num stack} (* I3a *) $returns\{(_, _, _, s, _, sn_n, num_n) \rightarrow wff_num sn_n num_n s\} (* I3b *)$ (* post conditions *) returns {(sn_n, m, _, s, _, _, num_n) \rightarrow sn_n = m = max_int() \vee $\exists y$. lmem y s \land sn_n = num_n[y] \land m = rank y s} (* E5 *) let m = rank x (Cons x stack) in let (n1, m1, b1, s1, sccs1, sn1, num1) = dfs' (successors x) blacks (add x grays) (Cons x stack) sccs (sn + 1) num[x \leftarrow sn] in if $n1 \ge sn$ then begin let (s2, s3) = split x s1 in (max_int(), max_int(), add x b1, s3, add (elements s2) sccs1, sn1, num1) end else (n1, m1, add x b1, s1, sccs1, sn1, num1)

predicate wff_num (sn: int) (num: map vertex int) (s: list vertex) =
 ($\forall x. num[x] < sn \leq max_int()$) \lambda
 ($\forall x y. lmem x s \rightarrow lmem y s \rightarrow num[x] \leq num[y] \leftrightarrow rank x s \leq rank y s$)

```
let rec dfs1 x blacks (ghost grays) stack sccs sn num =
  let m = rank x (Cons x stack) in
  let n = !sn in
  incr sn; num := !num[x \leftarrow n];
  let (n1, m1, b1, s1, sccs1) =
    dfs' (successors x) blacks (add x grays) (Cons x stack) sccs sn num in
  \operatorname{assert} \{ n1 \geq n \leftrightarrow m1 \geq m \}; \quad (* \ *)
  if n1 \ge n then begin
    let (s2, s3) = split x s1 num in
    assert{s3 = stack};
    assert{subset (elements s2) (add x b1)};
    assert {is_subscc (elements s2) \land mem x (elements s2)};
    assert {\forall y. in_same_scc y x \rightarrow mem y (elements s2)};
    assert { is_scc (elements s2) };
     (max_int(), max_int(), add x b1, s3, add (elements s2) sccs1) end
  else begin
    assert \{\exists y. mem y grays \land rank y s1 < rank x s1 \land reachable x y\};
   (n1, m1, add x b1, s1, sccs1) end
```



- implementation of graphs
- vertices as integers in an array
- successors as lists for every vertex

• see http://jeanjacqueslevy.net/why3

Conclusion

Conclusion

- readable proofs ?
- simple algorithms should have simple proofs
 - to be shown with a good formal precision
- compare with other proof systems
- further algorithms (in next talks ?)
 - graphs represented with arrays + lists
 - topological sort, articulation points, sccK, sscT
- Why3 is a beautiful system but not so easy to use !