Strongly Connected Components in graphs, formal proof of Tarjan1972 algorithm

jean-jacques.levy@inria.fr

Inria Sophia, 15-03-2017
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 (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 linear-time algorithm
The algorithm (1/3)

- 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 (2/3)
The algorithm (2/3)
The algorithm (2/3)
The algorithm (2/3)
The algorithm (3/3)

let rec printSCC (x: int) (s: Stack.t) (num: array int) (serialnb: ref int) =
  Stack.push x s;
  serialnb := !serialnb + 1;
  num[x] ← !serialnb;
  let min = ref num[x] in
  foreach y in (successors x) do
    let m = if num[y] = 0
      then printSCC y s num serialnb
      else num[y] in
    min := Math.min m !min
  done;

if !min = 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"
end;
return !min;

• print each component on a line
Proof in algorithms book (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 $w$.

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

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 $LOWLINK(v) = v$. 
Proof in algorithms book (2/2)

- give the program

- proof ↔ program

- that part of the proof is very informal
The algorithm (bis)

\[ \text{LOWLINK}(x) = \min \left( \{ \text{num}[x] \} \cup \{ \text{num}[y] \mid x \xrightarrow{*} y \} \right) \]

\[ \land x \text{ and } y \text{ are in same connected component} \]
The algorithm (ter)

\[ LOWLINK(x) = \min \left( \{ \text{rank}[y] \mid x \xrightarrow{*} y \text{ and } \text{y and } x \text{ are in same connected component} \} \right) \]
Our program (1/3)

• a functional version with lists and finite sets

• the working stack is a list

```plaintext
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
```
Our program (1/3)

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

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

```ml
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)
```
Our 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
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, _) → ∃s’. s = s’ ++ stack ∧ subset (elements s’) b} (* M1 *)
returns{(_, b, _, _) → subset blacks b} (* M2 *)
returns{(_, _, _, sccs_n) → subset sccs sccs_n} (* M3 *)
Pre/Post-conditions (2/3)

\[
\text{stack}
\]

\[
\begin{align*}
\text{sccs} & \subseteq \text{sccs}_n \\
\text{blacks} & \subseteq \text{b}
\end{align*}
\]

\[
\begin{align*}
\text{m} & \leq \text{rank y stack} \\
\text{m} & \leq \text{rank x stack}
\end{align*}
\]
Pre/Post-conditions (3/3)

with  dfs’ roots blacks (ghost grays) stack sccs =
requires{subset roots vertices} (* R1 *)
requires{∀x. mem x roots → access_to grays x} (* R2 *)

(* post conditions *)
returns{(_, b, _, _) → subset roots (union b grays)} (* E1 *)
returns{(m, _, s, _) → ∀x. mem x roots → m ≤ rank x s} (* E2 *)
returns{(m, _, s, _) → m = max_int() ∨ ∃x. mem x roots ∧ rank_of_reachable m x s} (* E3 *)
returns{(m, _, s, _) → ∀y. crossedgeto s y stack → m ≤ rank y stack} (* E4 *)
(* monotony *)
returns{(_, b, s, _) → ∃s’. s = s’ ++ stack ∧ subset (elements s’) b} (* M1 *)
returns{(_, b, _, _) → subset blacks b} (* M2 *)
returns{(_, _, _, sccs_n) → subset sccs sccs_n} (* M3 *)
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)
Paths

\textbf{inductive} path vertex \ (list vertex) \ vertex = \\
\quad \mid \text{Path}_{\text{empty}}: \\
\quad \quad \forall x: \text{vertex}. \ \text{path} \ x \ \text{Nil} \ x \\
\quad \mid \text{Path}_{\text{cons}}: \\
\quad \quad \forall x \ y \ z: \ \text{vertex}, \ l: \ \text{list} \ \text{vertex}. \\
\quad \quad \quad \text{edge} \ x \ y \ \rightarrow \ \text{path} \ y \ l \ z \ \rightarrow \ \text{path} \ x \ \text{(Cons} \ x \ l) \ z \\
\textbf{predicate} \ \text{reachable} \ (x \ z: \ \text{vertex}) = \\
\quad \exists l. \ \text{path} \ x \ l \ z \\

\textbf{predicate} \ \text{in\_same\_scc} \ (x \ z: \ \text{vertex}) = \\
\quad \text{reachable} \ x \ z \ \wedge \ \text{reachable} \ z \ x \\
\textbf{predicate} \ \text{is\_subsc} \ (s: \ \text{set} \ \text{vertex}) = \\
\quad \forall x \ z. \ \text{mem} \ x \ s \ \rightarrow \ \text{mem} \ z \ s \ \rightarrow \ \text{in\_same\_scc} \ x \ z \\
\textbf{predicate} \ \text{is\_scc} \ (s: \ \text{set} \ \text{vertex}) = \\
\quad \text{is\_subsc} \ s \ \wedge \ (\forall s'. \ \text{subset} \ s \ s' \ \rightarrow \ \text{is\_subsc} \ s' \ \rightarrow \ s = s')
Invariants (1/4)

predicate no_black_to_white (blacks grays: set vertex) =
\( \forall x \, x'. \text{ edge } x \, x' \rightarrow \text{mem } x \, \text{blacks} \rightarrow \text{mem } x' \, (\text{union } \text{blacks} \, \text{grays}) \)

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

\text{elements } s = \text{grays} \bigcup \text{blacks} - (\text{set_of sccs})

(set_of sccs) \subseteq \text{blacks}
Invariants (2/4)

sccs

cc1  cc2  ccn

increasing rank

blacks  grays

s
Invariants (3/4)

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

  wff_color blacks grays s sccs ∧
  simplelist s ∧
  subset (elements s) vertices ∧

  (∀x y. mem x grays → lmem y s →
    rank x s ≤ rank y s → reachable x y) ∧

  (∀y. lmem y s → ∃x. mem x grays ∧
    rank x s ≤ rank y s ∧ reachable y x)```

Invariants (4/4)
let m = rank x (Cons x stack) in
let (m1, b1, s1, scs1) =
dfs’ (successors x) blacks (add x grays) (Cons x stack) scs 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_subsc (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) scs1) end
else begin

  (m1, add x b1, s1, scs1) end
Assertions

\[
\text{assert}\{\forall y. \text{in\_same\_scc} \ y \ x \rightarrow \text{mem} \ y \ (\text{elements} \ s2)\};
\]

- Coq proof: there exists \(x', y'\) with \(x' \in s2 \land y' \notin s2 \land \text{edge} \ x' \ y'\) and \(x', y'\) are in same strongly connected component as \(x\)

\[
y' \in s3 = \text{stack}
\]
- \(x' = x\) impossible because \(m1 \leq \text{rank} \ y' < \text{rank} \ x s1\)
- \(x' \neq x\) impossible because crossedge

\[
y' \in \text{scs} \quad \text{impossible because scs disjoint from stack}
\]

\[
y' \text{ is white}
\]
- \(x' = x\) impossible because successors are black
- \(x' \neq x\) impossible because no black to white
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} (* I1b *)
returns{(_, b, _, sccs_n) \rightarrow \forall cc. mem cc sccs_n \leftrightarrow subset cc b \land is_scc cc} (* I2b *)
(* 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() \lor 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 \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 *)
Full proof

- Full proof is at http://jeanjacqueslevy.net/why3
- See the file why3session.html
- Proof: 185 lines (38 lemmas) including the program texts.
- 82 proof obligations
  - All proved automatically by Alt-Ergo (1.30), CVC3 (2.4.1), CVC4 (1.4), Eprover (1.9), Spass (3.5), Yices (1.0.4)
  - Except 5 of manually checked by Coq (8.6)
- Coq proofs are 240 lines (25+20+119+32+44)
Towards imperative program
let rec dfs1 x blacks (ghost grays) stack sccs sn num =
requires{sn = cardinal (union grays blacks) \&\& 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 = \text{max\_int()} \lor
\exists y. \text{lmem} y s \&\& sn_n = \text{num\_n}[y] \lor m = \text{rank} y s} (* E5 *)

let m = \text{rank} x (Cons x stack) in
let (n1, m1, b1, s1, sccs1, sn1, num1) =
\text{dfs'}(\text{successors} x) blacks (add x grays) (Cons x stack) sccs (sn + 1) num[x \leftarrow sn] in
if n1 \geq sn then begin
  let (s2, s3) = \text{split} x s1 in
  (\text{max\_int}(), \text{max\_int}(), add x b1, s3, add (\text{elements} s2) sccs1, sn1, num1) end
else
(n1, m1, add x b1, s1, sccs1, sn1, num1)
Assertions

\[
\text{predicate wff_num (sn: int) (num: map vertex int) (s: list vertex) = } \\
(\forall x. \text{num}[x] < \text{sn} \leq \text{max_int}()) \land \\
(\forall x \ y. \text{lmem} x \ s \rightarrow \text{lmem} y \ s \rightarrow \text{num}[x] \leq \text{num}[y] \leftrightarrow \text{rank} x \ s \leq \text{rank} y \ s)
\]
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 ← n];
  let (n1, m1, b1, s1, sccs1) =
      dfs' (successors x) blacks (add x grays) (Cons x stack) sccs sn num in
  assert\{n1 ≥ n ≡ m1 ≥ m\}; (* *)
  if n1 ≥ n then begin
    let (s2, s3) = split x s1 num in
    assert\{s3 = stack\};
    assert\{subset (elements s2) (add x b1)\};
    assert\{is_subscce (elements s2) ∧ mem x (elements s2)\};
    assert\{∀y. in_same_scc y x → 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\{∃y. mem y grays ∧ rank y s1 < rank x s1 ∧ 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 (without automatic provers?)

• 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!