Depth-first search and strong connectivity in Coq

François Pottier

January 9, 2015

The problem

Finding the strongly connected components of a directed graph.

- *Pedagogical* value:

  The first nontrivial graph algorithm.

- *Practical* value:

  Applications in program analysis, constraint solving, model-checking, etc.

Known algorithms

Several known algorithms run in linear time:

- *Tarjan* (1972).
  - One pass. Maintains auxiliary data ("lowpoint values", etc.).
- *Kosaraju* (unpublished, 1978) and Sharir (1981).
  - Two passes. Maintains no auxiliary data.
  - Described in Cormen/Leiserson/Rivest's textbook.
  - Explained by Wegener (2002).
- *Gabow* (2000), improving on Purdom (1968) and Munro (1971).
  - One pass. Maintains a union-find data structure.

Kosaraju's algorithm

The algorithm is as follows:

1. Perform a DFS traversal of the *graph* E, producing a forest f_1.
2. Perform a DFS traversal of the *reverse graph* $\bar{E}$, visiting the roots in the *reverse post-order* of f_1, producing a forest f_2.

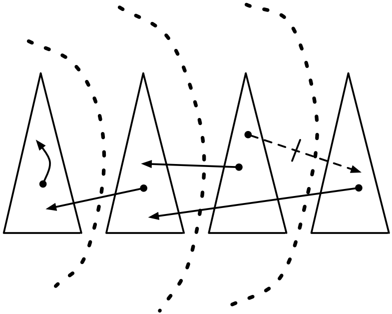
Then, f_2 is a list of the strongly connected components. *Magic!*

– Note: the second traversal does not have to be depth-first.

Really Easy to implement if you have done DFS already.

Why does this work?

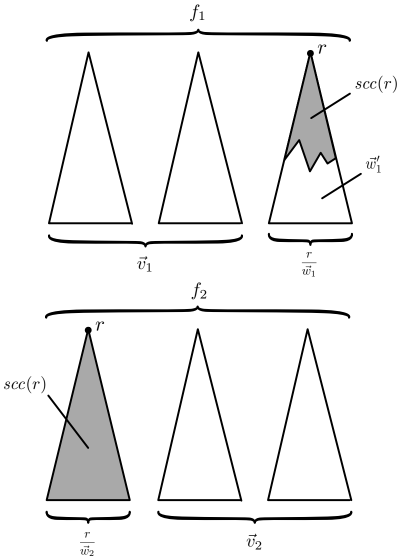
Complete discovery



The left side of every dashed boundary is closed w.r.t. E.

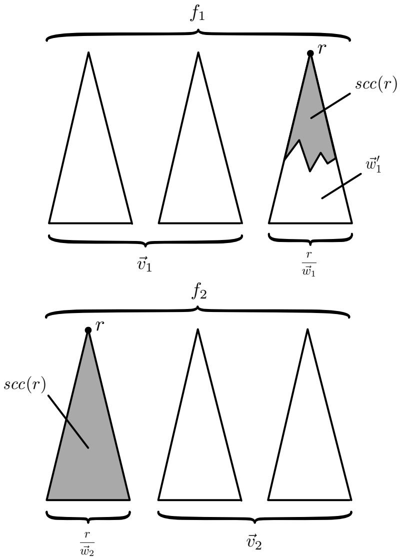
The right side of every dashed boundary is closed w.r.t. $\bar{E}$.

Every component is contained within some tree.



Let r be the root of the *last* tree in f_1.

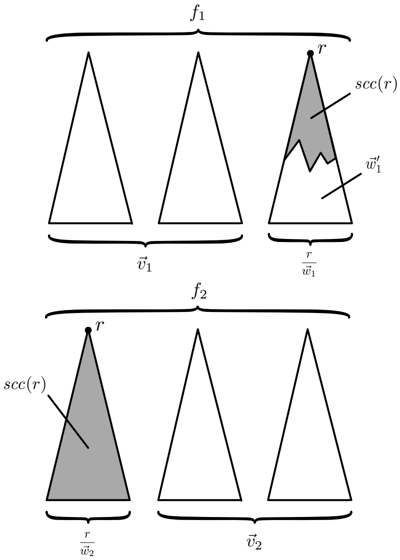
The component of r must be $\bar{E}^\star(r)$.



Let r be the root of the *last* tree in f_1.

The component of r must be $\bar{E}^\star(r)$.

So it is exactly the *first* tree in f_2.



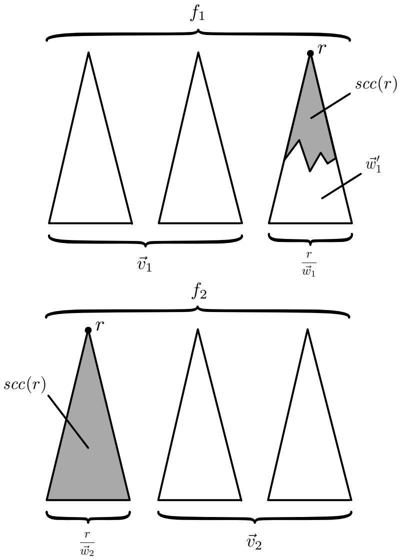
Let r be the root of the *last* tree in f_1.

The component of r must be $\bar{E}^{\star}(r)$.

So it is exactly the *first* tree in f_2.

Furthermore,

it is a *prefix* of the last tree in f_1.



Let r be the root of the *last* tree in f_1.

The component of r must be $\bar{E}^\star(r)$.

So it is exactly the *first* tree in f_2.

Furthermore,

it is a *prefix* of the last tree in f_1.

So, if we *remove* it by thought...

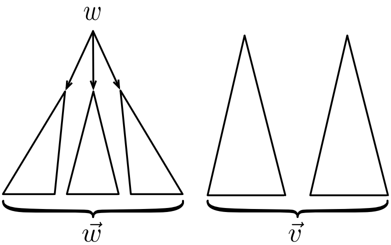
we end up where we started, ...

only with a *smaller* graph. (Induction!)

Now, in Coq (briefly)

Forests

A non-empty forest:



Forests form an inductive type:

$$f, \vec{v}, \vec{w} ::= \epsilon \mid \frac{w}{\vec{w}} :: \vec{v}$$

DFS forests

We define an inductive predicate $dfs\ (i)\ \vec{v}\ (o)$.

- It has a certain *declarative* flavor:

  $\vec{v}$ *is a DFS forest.*

- It still has a certain *imperative* flavor:

  if the vertices in i *are marked at the beginning,*
  then a DFS algorithm may construct $\vec{v}$*,*
  and the vertices in o *are marked at the end.*

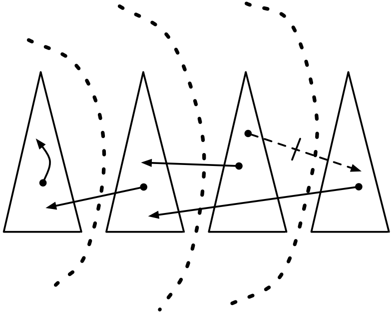
DFS forests

$$\frac{\textsc{DFS-Empty}}{dfs\ (i)\ \epsilon\ (i)}$$

DFS forests

DFS-NonEmpty	
$w \notin i$	w was not initially marked
$dfs\ (\{w\} \cup i)\ \vec{w}\ (m)$	after marking w, the DFS forest $\vec{w}$ was built
$roots(\vec{w}) \subseteq E(\{w\})$	every root of $\vec{w}$ is a successor of w
$E(\{w\}) \subseteq m$	every successor of w was marked at this point
$dfs\ (m)\ \vec{v}\ (o)$	then, the DFS forest $\vec{v}$ was built
$dfs\ (i)\ \dfrac{w}{\vec{w}} :: \vec{v}\ (o)$	the DFS forest $\dfrac{w}{\vec{w}} :: \vec{v}$ was built

Complete discovery

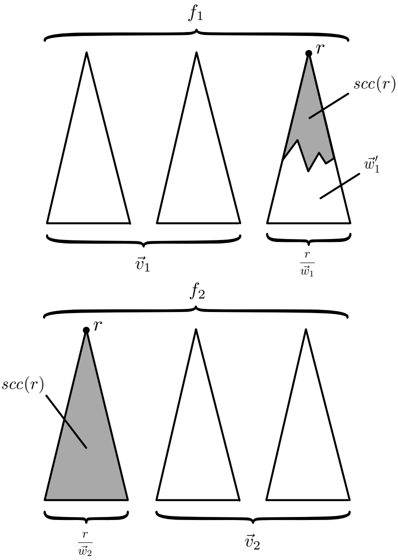


Complete discovery

Lemma (Complete discovery)

$\mathit{dfs}\ (i)\ \vec{v}\ (o)$ *and* $E(i) \subseteq i$ *imply* $E(o) \subseteq o$.

Easy. (The paper summary of the proof is a few lines long.)



f_1
r
$scc(r)$
$\vec{w}'_1$
$\vec{v}_1$
$\frac{r}{\vec{w}_1}$
f_2
r
$scc(r)$
$\frac{r}{\vec{w}_2}$
$\vec{v}_2$

Kosaraju's algorithm

Theorem (Kosaraju's algorithm is correct)

Let $(\mathcal{V}, E)$ *be a directed graph. If the following hypotheses hold,*

$$\begin{array}{c} dfs_{E}\ (\emptyset)\ f_1\ (\mathcal{V}) \\ dfs_{\bar{E}}\ (\emptyset)\ f_2\ (\mathcal{V}) \\ rev(post(f_1))\ orders\ f_2 \end{array}$$

then the toplevel trees of f_2 *are the components of the graph* E*.*

Slightly involved. (The paper summary of the proof is two pages.)

Towards an executable* DFS in Coq

(*executable = extractible)

Parameters

A set `V` of vertices.

`V` must be finite.

– Slightly too strong an assumption, but OK for now.

Parameters

A mathematical graph `E`.

A runtime function `successor v`
producing an iterator on the successors of `v`.

Parameters

A runtime representation of sets of vertices.

```
Record SET (V : Type) := MkSET {
  repr    : Type;
  meaning : repr -> (V -> Prop);
  void    : repr;
  mark    : V -> repr -> repr;
  marked  : V -> repr -> bool;
  ...    // 3 more hypotheses about void, mark, marked
}.
```

A recursive formulation

```
Notation state := (repr * forest V)%type.
```

A state records the marked vertices and the forest built so far.

A recursive formulation

One would like to write something like this:

```
Definition visitf : state -> V -> state := ...
```

This *cannot work*, though.

Because the recursive call sits in a loop, the proof of termination must use the fact that *a vertex, once marked, remains marked*.

So, we must build this information into the postcondition...

A recursive formulation

This states that s1 has at least as many marked vertices as s0:

```
Definition visitf_dep:
  forall s0 : state, V -> { s1 | lift le s0 s1 }.
Proof.
  eapply (Fix (...) (...)).
  ...
Defined.
```

Works. Unpleasant.

A tail-recursive formulation

Work in progress.

Termination is relatively easy to prove. (Generic library: *Loop*.)

Parameterized by user hooks (`on_entry`, `on_exit`, `on_rediscovery`).

Nice (?) most general (?) specification:

```
Theorem dfs_main_spec:
  exists vs,
  rev roots = rrootsl vs /\
  rdfs E (marked base) (marked dfs_main) vs /\
  dfs_main = rfold dfs_init_spec vs.
```

Running the iterative DFS algorithm is equivalent to *guessing* a DFS forest and recursively *folding* over this forest.

Conclusion

Conclusion

Contributions:

- Proofs of basic properties of DFS.
- A proof of (the principle of) Kosaraju's algorithm.
- Embryo of a certified DFS library. (More to come.)

Lessons:

- Separation between *mathematics* and *code* is desirable, and quite easy to achieve in Coq.
- Writing, specifying, proving *generic* executable code is a lot of work!
- We need a certified library of basic graph algorithms!