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.).
  - 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 \textit{last} tree in \( f_1 \).

The component of \( r \) must be \( \overline{E}^*(r) \).
Let $r$ be the root of the \textit{last} tree in $f_1$.

The component of $r$ must be $\overline{E^*(r)}$.

So it is exactly the \textit{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!)
Let $r$ be the root of the \textit{last} tree in $f_1$.  
The component of $r$ must be $\bar{E}^*(r)$.  
So it is exactly the \textit{first} tree in $f_2$.  
Furthermore, 
it is a \textit{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 $\overline{E^*}(r)$.

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:

```
 washed

 washed

 washed
```

Forests form an inductive type:

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

\[
\text{DFS-EMPTY} \\
\frac{\text{dfs} (i) \in (i)}{}
\]
\textbf{DFS-NONEMPTY}
\begin{align*}
& w \not\in i \\
& dfs \ (\{w\} \cup i) \ \vec{w} \ (m) \\
& roots(\vec{w}) \subseteq E(\{w\}) \\
& E(\{w\}) \subseteq m \\
& dfs \ (m) \ \vec{v} \ (o) \\
\hline
& dfs \ (i) \ \frac{w}{\vec{w}} :: \vec{v} \ (o)
\end{align*}

\begin{itemize}
  \item $w$ was not initially marked
  \item after marking $w$, the DFS forest $\vec{w}$ was built
  \item every root of $\vec{w}$ is a successor of $w$
  \item every successor of $w$ was marked at this point
  \item then, the DFS forest $\vec{v}$ was built
\end{itemize}

\begin{itemize}
  \item the DFS forest $\vec{w} :: \vec{v}$ was built
\end{itemize}
Complete discovery
Complete discovery

Lemma (Complete discovery)

dfs \( (i) \overset{\theta}{\rightarrow} (o) \) and \( E(i) \subseteq i \) imply \( E(o) \subseteq o \).

Easy. (The paper summary of the proof is a few lines long.)
François Pottier  Depth-first search and strong connectivity in Coq
Kosaraju’s algorithm

Theorem (Kosaraju’s algorithm is correct)

Let \((V, E)\) be a directed graph. If the following hypotheses hold,

\[
\begin{align*}
\text{dfs}_E (\emptyset) & \quad f_1 (V) \\
\text{dfs}_{\overline{E}} (\emptyset) & \quad f_2 (V) \\
\text{rev} (\text{post}(f_1)) & \quad \text{orders} \ f_2
\end{align*}
\]

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.
A mathematical graph \( E \).

A runtime function \texttt{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 $s_1$ has at least as many marked vertices as $s_0$:

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: \textit{Loop}.)

Parameterized by user hooks (on\_entry, on\_exit, on\_rediscovery).

Nice (?) most general (?) specification:

Theorem dfs\_main\_spec:
  \[
  \exists vs, \quad \text{rev roots} = \text{rrootsl} vs \land \\
  \text{rdfs E (marked base) (marked dfs\_main) vs} \land \\
  \text{dfs\_main} = \text{rfold dfs\_init\_spec} vs.
  \]

Running the iterative DFS algorithm is equivalent to \textit{guessing} a DFS forest and recursively \textit{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!