Depth-First Search and Strong Connectivity in Coq

François Pottier

INRIA

Abstract

Using Coq, we mechanize Wegener's proof of Kosaraju's linear-time algorithm for computing the strongly connected components of a directed graph. Furthermore, also in Coq, we define an executable and terminating depth-first search algorithm.

1. Introduction

The strongly connected components of a directed graph $(\mathcal{V}, E)$ can be computed in linear time. This striking fact, first established by Tarjan [7], is one of the fundamental results of algorithmic graph theory. Tarjan's algorithm is taught in many introductory algorithms courses. Unfortunately, this algorithm is rather difficult to explain, or to reconstruct, or to prove correct, in front of an audience composed of undergraduate students.

Instead, several textbooks [1, 2] present another linear-time algorithm, attributed to Kosaraju (unpublished) and Sharir [6]. This algorithm consists in one depth-first traversal of the graph E, followed with one depth-first traversal of the reverse graph $\bar{E}$. The first traversal influences the second, as follows: the order in which the vertices are examined during the second traversal is the reverse post-order of the forest f_1 produced by the first traversal. The forest f_2 produced by the second traversal has the property that every toplevel tree forms a component. (For the sake of brevity, from here on, we write "component" for "strongly connected component".)

As should be clear from this three-line description, Kosaraju's algorithm is significantly simpler than Tarjan's. Although the textbook proofs of this algorithm are still somewhat complex, Wegener [8] presents a beautiful reconstruction of the algorithm, which can be considered an explanation and a proof (albeit an informal one). It can be summarized as a sequence of remarks, or insights:

- A depth-first traversal of a graph produces a DFS forest (Figure 1). If one draws its trees from left to right and from least to most recently discovered, then such a forest has the property that there can be no graph edges "towards the right". That is, there cannot be a graph edge from a tree into some younger tree (Figure 2).

- Assume, as in our description of the algorithm, that a depth-first traversal of the graph E produces the forest f_1 (**H1**). Then, the previous remark implies that every component of E is contained within some tree of f_1.

- There is one specific component that can be easily found. Assuming the graph E and forest f_1 are non-empty, let r be the root of the rightmost tree in f_1 (Figure 3). By exploiting the previous remarks, it is not difficult to see that $scc(r)$ must be $\bar{E}^\star(r)$. That is, the component of r can be found by traversing the reverse graph $\bar{E}$, with starting point r.

- Assume, as in our description of the algorithm, that a depth-first traversal of the reverse graph $\bar{E}$ produces the DFS forest f_2 (**H2**), and that the sequence of the roots of f_2 respects the reverse post-order of the forest f_1 (**H3**). Then, r must be the vertex that appears first in the reverse

post-order of the forest f_1, hence must be the root of the first tree in the forest f_2. By the previous remark, this implies that the first tree in the forest f_2 is $scc(r)$ (Figure 3). Thus, this tree is a component.

- The last key insight is that the component $scc(r)$ must form a prefix of the forest f_1 (Figure 3). Thus, one may remove this component (by thought) from the graph E and forests f_1 and f_2, yielding a graph E' and two forests f'_1 and f'_2 which (it turns out) still satisfy the hypotheses **H1**, **H2** and **H3**. One can then repeat the whole argument, as many times as necessary, to conclude that every toplevel tree in the forest f_2 must be a component.

One attractive feature of this proof is that, in contrast with Cormen *et al.*'s [2], it never mentions the "times" at which vertices are visited. Instead, it is expressed purely in terms of graphs, forests, and their structural properties. This gives it a more "declarative", as opposed to "imperative", flavor.

In this paper, we make the following contributions:

1. We confirm the validity of Wegener's proof, as well as the elegance of its overall architecture, by mechanizing it in Coq. We find that it is indeed possible to structure the proof in the manner broadly suggested by Wegener. This puts the proof on a firm footing, and could be useful to someone who wishes to teach this material.
2. We define in Coq an executable and terminating depth-first search algorithm, which, given a (runtime representation of) a graph, constructs a DFS forest. The algorithm gives rise, by extraction, to satisfactory OCaml code for depth-first search. We report on our experience developing this code, and suggest several ways of improving it.

In the paper, we elide the details of our machine-checked version of Wegener's proof. We give only an outline of the proof, insofar as a (formalist!) teacher might explain it. We explicitly state several definitions and lemmas, but omit or inline a number of "trivial" auxiliary lemmas. We accompany each lemma with a hand-written proof, formulated in a clear but informal style, and collect these proofs in an appendix.

The paper is organized as follows. We define DFS forests and study their properties (§2). Then, we prove that Kosaraju's algorithm is correct: that is, the hypotheses **H1**, **H2** and **H3** above imply that every toplevel tree in the forest f_2 is a component of the graph E (§3). We develop an executable depth-first search algorithm (§4). We briefly review the related work (§5) and conclude.

2. Depth-First Search

2.1. Sets and relations

Throughout the paper, we fix a type $\mathcal{V}$ of **vertices**. We let r, v and w range over vertices.

We let V and W (in §2.1) as well as i, m, o, and M (further on in the paper) range over **sets of vertices**. (A set of vertices is represented in Coq as a predicate of type $\mathcal{V} \rightarrow \mathit{Prop}$.) We use the standard set-theoretic operations (union, intersection, inclusion, etc.). In particular, we write $\neg V$ for the complement of the set V.

A (directed) **graph** E is a binary relation over vertices. (Such a relation is represented in Coq as a predicate of type $\mathcal{V} \rightarrow \mathcal{V} \rightarrow \mathit{Prop}$.) We write $v\,E\,w$ to indicate that the vertices v and w are related by E, i.e., there is an edge from v to w.

We write $E(V)$ for the **image** of the set V under the relation E.

The assertion $E(V) \subseteq W$ means that every edge whose source vertex lies in V has a destination vertex in the set W: we say V **goes into** W. In particular, $E(V) \subseteq V$ means that the set V is **closed**: every edge that starts in V ends in V, or in other words, no edge leaves V.

The assertion $W \subseteq E(V)$ means that every member of W is the target of some edge whose source vertex lies in V: we say V **covers** W.

We write $E^\star$ for the reflexive, transitive **closure of a relation** E. If $v\,E\,w$ indicates the existence of an edge from v to w, then $v\,E^\star\,w$ indicates the existence of a path from v to w. The **closure of a set** of vertices V is the set $E^\star(V)$. It is the set of all vertices that can be reached from V by following some path in the graph E.

The assertion $W \subseteq E^\star(V)$ means that every vertex in W can be reached from some vertex in V by following some path: we say V **reaches** W.

The **reverse graph** $\bar{E}$ is defined by letting $v\,\bar{E}\,w$ hold if and only if $w\,E\,v$ holds. The **strong connectivity** relation scc_E, or just scc, is defined as the intersection of $E^\star$ and $\bar{E}^\star$. Thus, $v\ scc\ w$ holds if and only if there is a path from v to w and back. The **component** $scc(v)$ of the vertex v is just the image of the vertex v under the relation scc.

2.2. Forests

We let f, $\vec{v}$ and $\vec{w}$ range over **forests**, which are inductively defined by $f, \vec{v}, \vec{w} ::= \epsilon \mid \frac{w}{\vec{w}} :: \vec{v}$. That is, a forest is either the empty forest ϵ, which consists of zero trees; or a non-empty forest (Figure 1), which consists of one tree (whose root is the vertex w and whose children form the forest $\vec{w}$) followed by zero or more trees (which form the forest $\vec{v}$).

By abuse of notation, we sometimes write $\frac{w}{\vec{w}}$ for a "tree", i.e., for the forest $\frac{w}{\vec{w}} :: \epsilon$.

Although we define forests as a stand-alone inductive type, one can think of a forest as a list of trees. This is why, in the paper, we choose to write $\frac{w}{\vec{w}} :: \vec{v}$ for a non-empty forest. This is an abuse of notation, as we also use the double-colon notation for prepending an element in front of a list, as in $r :: \vec{r}$. By a similar abuse of notation, we write ϵ for both the empty list and the empty forest.

A few standard functions over lists must be redefined for forests. In particular, the **concatenation** of two forests $\vec{v}$ and $\vec{w}$ is defined in the same manner as the concatenation of two lists. In the paper, we write $+\!\!+$ both for list concatenation and for forest concatenation.

The **roots** and **support** of a forest are sets of vertices, defined as follows:

$$
\begin{aligned}
roots(\epsilon) &= \emptyset & \qquad support(\epsilon) &= \emptyset \\
roots(\tfrac{w}{\vec{w}} :: \vec{v}) &= \{w\} \cup roots(\vec{v}) & \qquad support(\tfrac{w}{\vec{w}} :: \vec{v}) &= \{w\} \cup support(\vec{w}) \cup support(\vec{v})
\end{aligned}
$$

The **preorder** and **postorder** of a forest are lists of vertices, defined as follows:

$$
\begin{aligned}
pre(\epsilon) &= \epsilon & \qquad post(\epsilon) &= \epsilon \\
pre(\tfrac{w}{\vec{w}} :: \vec{v}) &= w :: pre(\vec{w}) +\!\!+ pre(\vec{v}) & \qquad post(\tfrac{w}{\vec{w}} :: \vec{v}) &= post(\vec{w}) +\!\!+ w :: post(\vec{v})
\end{aligned}
$$

We write $rev(\vec{r})$ for the reverse of the list $\vec{r}$. Thus, the **reverse postorder** of a forest $\vec{v}$ is $rev(post(\vec{v}))$.

2.3. DFS Forests

We now define the notion of a "DFS forest". This notion takes the form of an inductive predicate, *dfs* (§2.3.1), whose properties we study (§2.3.2–§2.3.5). Throughout §2.3, we fix a graph E.

2.3.1. Definition

In the following, i and o range over sets of vertices. They represent the sets of vertices that are marked at the beginning and at the end of a (partial) depth-first search, respectively. The mnemonic is as follows: i stands for "input" and o stands for "output". We also let m range over sets of vertices; it usually denotes the vertices that are marked somewhere in the "middle" of a depth-first search.

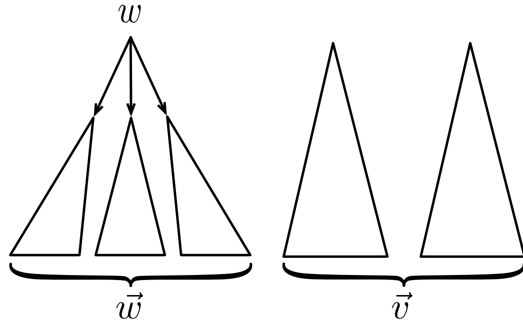

Figure 1: A non-empty forest $\frac{w}{\vec{w}} :: \vec{v}$. The sub-forest $\vec{w}$ represents the children of the vertex w. The sub-forest $\vec{v}$ represents the neighbors of the tree $\frac{w}{\vec{w}}$.

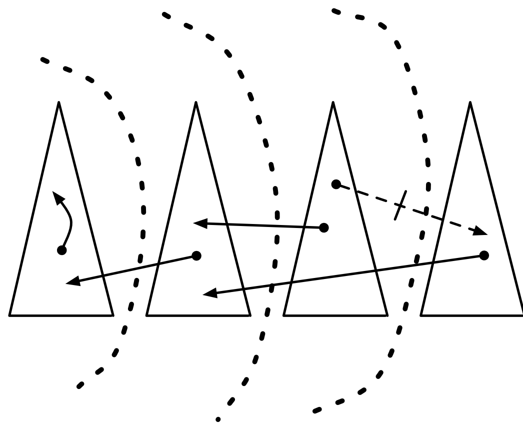

Figure 2: A DFS forest. The solid edges are examples of edges that may exist in the graph. The dashed, crossed-out edge is an example of an edge that cannot exist in the graph. The set of vertices on the left-hand side of every dashed boundary is closed.

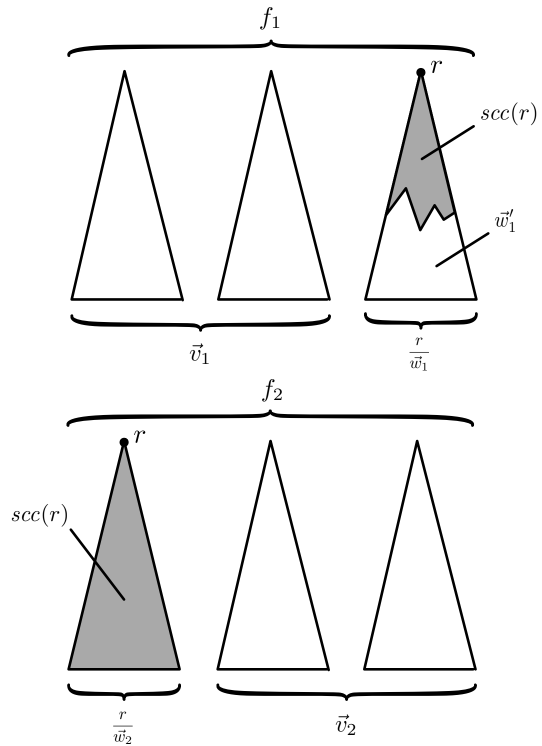

Figure 3: The principle of Kosaraju's algorithm. f_1 is a DFS forest for the graph E. f_2 is a DFS forest for the reverse graph $\bar{E}$. The vertex r is the root of the last tree in f_1 and the root of the first tree in f_2. The component $scc(r)$ forms a prefix of the last tree in f_1 and is exactly the first tree in f_2.

The predicate *dfs* $(i)\ \vec{v}\ (o)$ means that $\vec{v}$ **is a DFS forest** with respect to the sets i and o and (implicitly) with respect to the graph E. It is inductively defined by the two rules that follow. The pseudo-rule on the right-hand side is an informal reading of DFS-NonEmpty.

DFS-Empty

$$\frac{}{\mathit{dfs}\ (i)\ \epsilon\ (i)}$$

DFS-NonEmpty

$$\frac{\begin{array}{c} w \notin i \\ \mathit{dfs}\ (\{w\} \cup i)\ \vec{w}\ (m) \\ \mathit{roots}(\vec{w}) \subseteq E(\{w\}) \\ E(\{w\}) \subseteq m \\ \mathit{dfs}\ (m)\ \vec{v}\ (o) \end{array}}{\mathit{dfs}\ (i)\ \frac{w}{\vec{w}} :: \vec{v}\ (o)}$$

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

The predicate *dfs* $(i)\ \vec{v}\ (o)$ means that a depth-first search, beginning in a state where the marked vertices are i, can construct the forest $\vec{v}$ and end in a state where the marked vertices are o. Let us stress that this is not an executable description of the depth-first search algorithm. It is a declarative specification, which should be thought of as a property of the forest $\vec{v}$.

If one attempted to read the definition of *dfs* $(i)\ \vec{v}\ (o)$ as an executable specification, where i is an input parameter and $\vec{v}$ and o are output parameters, then one would find that this definition describes a non-deterministic process. Where the search should begin is not specified: which are the roots? in which order should they be visited? Technically, in DFS-NonEmpty, the first root w can be chosen in an arbitrary manner, as long as it is not marked already (premise #1). For the same reason, one level down, the order in which the as-yet-unmarked successors of w are visited is not specified; the second premise of DFS-NonEmpty does not impose a particular order. Where the search should stop is not specified either: at the top level, DFS-Empty can always be used to "stop early". In other words, a DFS forest is not required to cover the whole graph: it is perfectly acceptable for a DFS forest to cover only part of the graph.

We believe that, by defining *dfs* as a predicate, as opposed to a deterministic and terminating algorithm, we facilitate reasoning about the properties of DFS forests, independently of the effective method by which they are built.

Premises #2 and #5 in DFS-NonEmpty require the sub-forests $\vec{w}$ and $\vec{v}$ to be DFS forests. They also take care of augmenting and threading the set of marked vertices. The vertex w is marked immediately, that is, before constructing the sub-forest $\vec{w}$. After constructing this sub-forest, the marked vertices are m, and after constructing the sub-forest $\vec{v}$, they are o.

Premises #3 and #4 are used to control the sub-forest $\vec{w}$ that is constructed in premise #2. Premise #3 requires that the roots of $\vec{w}$ be successors of w. This means that every forest edge must be a graph edge, or in other words, that a DFS forest with respect to E is a sub-graph of E. Premise #4 states that, after $\vec{w}$ has been constructed, every successor of w must be marked. In other words, as long as there are unmarked successors of w, they must be examined; it is not permitted to stop early and declare that w has been fully processed.

2.3.2. Basic properties

DFS forests can be concatenated and split, as follows.

Lemma 1 (Concatenation; splitting) *dfs* $(i)\ \vec{v}\ (m)$ *and dfs* $(m)\ \vec{w}\ (o)$ *imply dfs* $(i)\ \vec{v} +\!\!+ \vec{w}\ (o)$. *Conversely, dfs* $(i)\ \vec{v} +\!\!+ \vec{w}\ (o)$ *implies dfs* $(i)\ \vec{v}\ (m)$ *and dfs* $(m)\ \vec{w}\ (o)$ *for some set of vertices* m. $\diamond$

The set of marked vertices grows with time. In other words, a vertex once marked remains marked.

Lemma 2 (Monotonicity) *dfs* $(i)\ \vec{v}\ (o)$ *implies* $i \subseteq o$. $\diamond$

By exploiting this property, we are able to establish a more precise statement:

Lemma 3 (Marking) *If dfs $(i)\ \vec{v}\ (o)$ holds, then the sets i and $support(\vec{v})$ are disjoint, and the set o is their union.* ◇

In light of the previous result, we note that *dfs* could have been defined as a predicate of two parameters i and $\vec{v}$, and the set o could then have been computed as $i \cup support(\vec{v})$. The choice between these presentations seems to be a matter of taste.

As a corollary of the previous results, we find that a vertex cannot appear twice in a DFS forest.

Lemma 4 (Unique visit) *dfs $(i)\ \vec{w} +\!\!+ \vec{v}\ (o)$ implies $support(\vec{v}) \subseteq \neg support(\vec{w})$.* ◇

2.3.3. Sound discovery

Quite obviously, because every edge in a DFS forest is a graph edge, the support of a DFS forest is reachable (via paths in the graph) from its roots. In other words, "only reachable vertices are discovered".

Lemma 5 (Sound discovery) *dfs $(i)\ \vec{v}\ (o)$ implies $support(\vec{v}) \subseteq E^{\star}(roots(\vec{v}))$.* ◇

2.3.4. Complete discovery

We now wish to prove the converse property, that is, "all reachable vertices are discovered". We do so in the following three lemmas. We begin with a simple statement: if $\vec{v}$ is a DFS forest, then, at the point in time where $\vec{v}$ has just been built, every successor of a vertex in $\vec{v}$ is marked.

Lemma 6 (Every successor is marked) *dfs $(i)\ \vec{v}\ (o)$ implies $E(support(\vec{v})) \subseteq o$.* ◇

This allows us to derive a (necessary and) sufficient condition for the set o to be closed.

Lemma 7 *dfs $(i)\ \vec{v}\ (o)$ and $E(i) \subseteq o$ imply $E(o) \subseteq o$.* ◇

As a special case of the previous lemma, we find that closedness is an invariant: if i is closed, then o is closed as well.

Lemma 8 (Complete discovery) *dfs $(i)\ \vec{v}\ (o)$ and $E(i) \subseteq i$ imply $E(o) \subseteq o$.* ◇

Naturally, the empty set is closed, so if one begins with an empty set of marked vertices, then the above lemma guarantees that, after each new toplevel DFS tree is produced, the set of marked vertices is closed (Figure 2).

The fact that o is closed means that, at the end, every vertex that is reachable from a marked vertex must be marked. Since at the end the roots of the forest $\vec{v}$ are marked (by Lemma 3, $roots(\vec{v})$ is a subset of o), there follows that every vertex that is reachable from some root is marked, that is, $E^{\star}(roots(\vec{v})) \subseteq o$. Thus, we have the desired property of "complete discovery". The formulation of Lemma 8 as the preservation of a closedness invariant was new to the author, and seems pleasing.

2.3.5. DFS forests and strongly connected components

Assume w is the root of a toplevel tree $\frac{w}{\vec{w}}$ in a DFS forest. If one starts at w and follows a path in the graph E, then one must remain in the tree $\frac{w}{\vec{w}}$ or in the trees towards the left (i.e., in i). A look at Figure 2 should reveal that this makes intuitive sense: there is no way to cross a dashed boundary towards the right.

Lemma 9 (Going left) *dfs $(i)\ \frac{w}{\vec{w}} :: \vec{v}\ (o)$ and $E(i) \subseteq i$ imply $E^{\star}(\{w\}) \subseteq i \cup support(\frac{w}{\vec{w}})$.* ◇

Symmetrically, if one starts at w and follows a path in the reverse graph $\bar{E}$, then one must remain in the tree $\frac{w}{\vec{w}}$ or in the trees towards the right (i.e., in $\vec{v}$).

Lemma 10 (Going right) *$dfs\ (i)\ \frac{w}{\vec{w}} :: \vec{v}\ (\mathcal{V})$ and $E(i) \subseteq i$ imply $\bar{E}^\star(\{w\}) \subseteq support(\frac{w}{\vec{w}} :: \vec{v})$.* $\diamond$

A slightly subtle detail in the previous statement is our use of $\mathcal{V}$, the set of all vertices, as opposed to an arbitrary set o. This ensures that the forest $\vec{v}$ represents all of the trees towards the right.

By intersection of the previous two results, one finds that, if w is the root of a toplevel tree $\frac{w}{\vec{w}}$ in a DFS forest, then the component of w must be contained in this tree.

Lemma 11 (Every component is contained within some tree) *$dfs\ (i)\ \frac{w}{\vec{w}} :: \vec{v}\ (\mathcal{V})$ and $E(i) \subseteq i$ imply $scc(w) \subseteq support(\frac{w}{\vec{w}})$.* $\diamond$

(The last three lemmas could easily be generalized to account for the case where w is an arbitrary vertex in the tree, not necessarily its root.)

Now, following Wegener's insight, let us focus on the last tree $\frac{w}{\vec{w}}$ in a DFS forest $\vec{v} +\!\!+ \frac{w}{\vec{w}}$. As a consequence of the previous results, we find that the component of w must be $\bar{E}^\star(\{w\})$, that is, the set of vertices that are reachable from w in the reverse graph $\bar{E}$.

Lemma 12 (Last tree) *$dfs\ (i)\ \vec{v} +\!\!+ \frac{w}{\vec{w}}\ (\mathcal{V})$ and $E(i) \subseteq i$ imply $scc(w) = \bar{E}^\star(\{w\})$.* $\diamond$

We also need a little fact about the first tree $\frac{w}{\vec{w}}$ in a DFS forest $\frac{w}{\vec{w}} :: \vec{v}$. Quite clearly, the set of vertices that are reachable from the root w is the whole tree $\frac{w}{\vec{w}}$, and nothing more. One could state this lemma under the assumption that i is empty, but we need a slightly more general statement, where i may be non-empty, but cannot interfere, i.e., both i and $\neg i$ are closed. This lemma is used in the proof of our main result (Lemma 17), where it is applied to the reverse graph.

Lemma 13 (First tree) *$dfs\ (i)\ \frac{w}{\vec{w}} :: \vec{v}\ (o)$ and $\left\{ \begin{array}{l} E(i) \subseteq i \\ E(\neg i) \subseteq \neg i \end{array} \right.$ imply $E^\star(\{w\}) = support(\frac{w}{\vec{w}})$.* $\diamond$

3. Strong Connectivity

In the following, we define the predicate *orders* (§3.1), which appears in our specification of Kosaraju's algorithm, in the hypothesis **H3** (see §1). Then, we define a suffix ordering on forests and study some of its properties (§3.2). Finally, we present the proof of Kosaraju's algorithm (§3.3).

3.1. Root Ordering

In the following, r stands for a vertex, and $\vec{r}$ stands for a list of vertices. The predicate $\vec{r}$ *orders* $\vec{v}$ is inductively defined as follows:

$$\textsc{Ordered-Nil}\quad \frac{}{\epsilon\ orders\ \epsilon} \qquad \textsc{Ordered-Skip}\quad \frac{r \notin support(\vec{v}) \quad \vec{r}\ orders\ \vec{v}}{r :: \vec{r}\ orders\ \vec{v}} \qquad \textsc{Ordered-Root}\quad \frac{\vec{r}\ orders\ \vec{v}}{r :: \vec{r}\ orders\ \frac{r}{\vec{w}} :: \vec{v}}$$

In the absence of the premise $r \notin support(\vec{v})$ in Ordered-Skip, the predicate $\vec{r}$ *orders* $\vec{v}$ would mean exactly that the roots of the forest $\vec{v}$ form a subsequence of the list $\vec{r}$. Due to this premise, however, the predicate $\vec{r}$ *orders* $\vec{v}$ is more restrictive. It means intuitively that a depth-first search algorithm which (as part of its toplevel loop) examines all vertices in the order prescribed by $\vec{r}$ can (plausibly) produce the forest $\vec{v}$. Whenever the algorithm examines a new vertex r, this vertex either becomes the root of the next toplevel tree, or is skipped. The latter case occurs if and only if r is already marked, i.e., if and only if r is *not* part of the forest $\vec{v}$ that *remains* to be constructed. Technically, the premise $r \notin support(\vec{v})$ in Ordered-Skip is exploited in the proof of Lemma 17.

3.2. Suffix Ordering

Throughout §3.2, let us fix a set M of so-called "masked" vertices. We define a predicate $\vec{v} \geq_M \vec{v}'$ which means that the set M forms a prefix of the forest $\vec{v}$ and that removing these vertices yields the forest $\vec{v}'$. (This implies that $\vec{v}'$ is a suffix of $\vec{v}$.)

$$\textsc{Filter-Empty}\quad \frac{}{\epsilon \geq_M \epsilon} \qquad \textsc{Filter-Masked}\quad \frac{w \in M \quad \vec{w} \geq_M \vec{w}' \quad \vec{v} \geq_M \vec{v}'}{\frac{w}{\vec{w}} :: \vec{v} \geq_M \vec{w}' +\!\!+ \vec{v}'} \qquad \textsc{Filter-Visible}\quad \frac{w \notin M \quad \mathit{support}(\vec{w}) \subseteq \neg M \quad \vec{v} \geq_M \vec{v}'}{\frac{w}{\vec{w}} :: \vec{v} \geq_M \frac{w}{\vec{w}} :: \vec{v}'}$$

If $\vec{v}$ is a DFS forest, and if we remove a prefix of it, then the remaining forest $\vec{v}'$ is still a DFS forest. This is stated as follows. We write E' for the graph E deprived of all edges whose source or destination lies in M. We note that the predicate *dfs* is now explicitly parameterized with a graph.

Lemma 14 (DFS Surgery) $\mathit{dfs}_E\ (i)\ \vec{w}\ (o)$ *and* $\vec{w} \geq_M \vec{w}'$ *imply* $\mathit{dfs}_{E'}\ (i \cup M)\ \vec{w}'\ (o \cup M)$. ◇

Note that, if $M \subseteq i$ holds, then the conclusion of this lemma boils down to $\mathit{dfs}_{E'}\ (i)\ \vec{w}'\ (o)$. This is exploited in the proof of Lemma 17.

Now and until the end of §3.2, let us fix a vertex r, and assume that the set of masked vertices is exactly the component of r, i.e., $M = \mathit{scc}(r)$.

If every root of a DFS forest $\vec{v}$ is reachable from r, then $\mathit{scc}(r)$ must form a prefix of $\vec{v}$.

Lemma 15 $\mathit{dfs}_E\ (i)\ \vec{v}\ (o)$ *and* $\mathit{support}(\vec{v}) \subseteq E^\star(\{r\})$ *imply* $\vec{v} \geq_M \vec{v}'$ *for some forest* $\vec{v}'$. ◇

As a corollary, if the distinguished vertex r is the root of the last tree in a DFS forest, then the component of r is a prefix of the last tree.

Lemma 16 $\mathit{dfs}_E\ (i)\ \vec{v} +\!\!+ \frac{r}{\vec{w}}\ (\mathcal{V})$ *and* $E(i) \subseteq i$ *imply* $\vec{v} +\!\!+ \frac{r}{\vec{w}} \geq_M \vec{v} +\!\!+ \vec{w}'$ *for some* $\vec{w}'$. ◇

3.3. Proof of Kosaraju's Algorithm

Let us say that a forest $\vec{v}$ is an **SCC forest** for some graph E if and only if every toplevel tree in $\vec{v}$ is a component of E. The main result of this paper is the following lemma:

Lemma 17 (Specification and soundness of Kosaraju's algorithm) *Let* $(\mathcal{V}, E)$ *be a directed graph. Assume that the forest* f_1 *is produced by a complete depth-first traversal of the graph:*

$$\mathit{dfs}_E\ (\emptyset)\ f_1\ (\mathcal{V})$$

Assume that the forest f_2 *is produced by a complete depth-first traversal of the reverse graph:*

$$\mathit{dfs}_{\bar{E}}\ (\emptyset)\ f_2\ (\mathcal{V})$$

Finally, assume that f_2 *obeys the reverse postorder of* f_1*:*

$$\mathit{rev}(\mathit{post}(f_1))\ \mathit{orders}\ f_2$$

Then, f_2 *is an SCC forest for the graph* E. ◇

4. A Depth-First Search Algorithm

We now describe an executable depth-first search algorithm, defined in Coq. This is exploratory work, which could be improved in several ways:

- We use a recursive formulation, because it seems more elegant. Yet, it occurred to us that, after the code is extracted to OCaml, this is likely to cause stack overflows when dealing with large graphs. A tail-recursive formulation, using an explicit stack, would be more robust.
- Building a proof of termination into the Coq definition of the algorithm is slightly tricky. The approach followed here leads to clean OCaml code, but causes us to jump through some hoops that involve dependent types and programming with tactics. A Coq expert would certainly find room for improvement here. More fundamentally, perhaps a tail-recursive formulation would not run into this issue (see §4.2).
- Our algorithm maintains a state of a fixed type, namely a pair of a set of marked vertices and a forest. Although this is good enough for some applications, including our intended application to Kosaraju's algorithm, it would be preferable to have a generic algorithm, in the style of Neumann's [4], which we discuss further on (§5).

4.1. Assumptions

When writing executable code in Coq, one must distinguish between mathematical objects (vertices; graphs; sets of vertices) and their runtime representations. A mathematical set of vertices, for instance, is just a predicate of type $\mathcal{V} \to \mathit{Prop}$ (§2.1), whereas the runtime representation of a set of vertices could be, say, a balanced binary search tree, provided by Coq's `FSets` library. Because we do not wish to impose a particular runtime representation, our code is parametric in the runtime representations of (1) graphs and (2) sets of vertices. We take a shortcut and assume that "mathematical vertices" and "runtime vertices" are the same thing; ideally, this assumption should be removed.

We assume a type `V` of vertices, and assume that `V` is finite. This assumption is exploited in the termination argument.

We assume a mathematical graph `E`. We assume a runtime representation of this graph, in the form of a function `successor`, which maps a vertex `v` to a higher-order iterator over the successors of `v`. Thus, `successor` has type `V -> FOREACH V`, where `FOREACH V` is a record type with one field, `foreach`, of type `forall A, A -> (A -> V -> A) -> A`. We assume that the graph `E` and its runtime representation `successor` are related, as follows. We require the property:

```
forall v, FOREACH_SET_SPEC (successor v) (image E (singleton v))
```

which means that invoking `foreach (successor v) a f` is equivalent to invoking `fold_left f vs a`, for some list of vertices `vs` whose members are the successors of `v` in the graph `E`. It may be worth noting that this specification allows a single successor to have multiple occurrences in the list `vs`. This flexibility is useful in applications where repeated edges may arise.

We assume a runtime representation of sets of vertices, equipped with three operations: the empty set, marking a vertex, and testing whether a vertex is marked. This leads to a record type with eight components, namely the four that we just listed, plus four assumptions that connect these four runtime objects with their mathematical counterparts.

```
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
}.
```

4.2. A Recursive Function

The type `state` is defined as an abbreviation for `repr * forest V`. The state of the algorithm is a pair of the set of marked vertices and the forest built so far.

Because `V` is finite, the strict superset ordering on `V -> Prop` is well-founded. Via the `meaning` function, this ordering can be transported to the type `repr` of runtime sets of vertices, and from there, to the type `state`.

The recursive function `visitf_dep` is defined by well-founded recursion over this ordering. Ideally, we would like this function to have type `state -> V -> state`, as it visits one vertex and updates the current state. Unfortunately, a moment's thought reveals that this cannot work. When we perform a recursive call, we must prove that there are strictly more marked vertices now than at entry into the function. Because we perform several recursive calls in a sequence (as we iterate over the successors), we must establish and exploit the fact that a recursive call causes the set of marked vertices to grow (non-strictly). This leads us to build this information into the type of `visitf_dep`, which becomes a dependent type:

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

The postcondition `lift le s0 s1` means that there are at least as many marked vertices in the final state `s1` as in the initial state `s0`. The code is written in the form of a proof script, which is somewhat unnatural but tolerable. Perhaps Coq's `Program` mode would help avoid this.

It seems that we are paying for our choice of a recursive formulation. In a tail-recursive formulation, with an explicit stack, we would have to exhibit a more complex well-founded ordering, but we would not need to assign the function a dependent type, as the termination argument would not require a comparison of the pre- and post-states.

4.3. A Specification

In order to prove that `visitf_dep` is correct, we must provide a specification for it. More generally, we need a specification for the inner loop of `visitf_dep`, which invokes `visitf_dep` on every successor. This leads us to defining a predicate `visitf_spec s vs s'`, where `s` and `s'` are the pre- and post-states, and `vs` is a set of vertices that must be visited.

```
Definition visitf_spec (s : state) (vs : V -> Prop) (s' : state) :=
  let (marks,  forest)  := s  in
  let (marks', forest') := s' in
  (* 0 *) le marks marks' /\
  (* 1 *) subset vs (meaning marks') /\
  (* 2 *) subset (roots forest') (union vs (roots forest)) /\
  (* 3 *) forall init,
          dfs E init (meaning marks ) (revf forest ) ->
          dfs E init (meaning marks') (revf forest').
```

Conjunct #0 states that at least as many vertices are marked at the end as at the start. Conjunct #1 states that every vertex in `vs` is marked at the end. Conjunct #2 states that every root of the forest at the end either was already a root at the beginning or is a member of `vs`. Conjunct #3 states that, if the forest was DFS at the beginning, then it still is DFS at the end. For efficiency reasons, we build

recursively-reversed forests, where younger trees appear first; hence the use of the auxiliary function `revf`, whose definition we omit.

We prove three key properties of `visitf_spec`. First, if the vertices `vs` are already marked in the initial state `s`, then the specification is satisfied by doing nothing, that is, `visitf_spec s vs s` holds. Second, to visit the union of the sets `vs` and `ws`, it suffices to visit each of these sets in succession, that is, `visitf_spec s0 vs s1` and `visitf_spec s1 ws s2` imply `visitf_spec s0 (union vs ws) s2`. Third, in order to visit a single unmarked vertex `w`, it suffices to (a) mark `w`; (b) visit its successors, yielding a sub-forest `ws`; (c) prepend the tree formed by `w` and `ws` in front of the current forest. (We omit the Coq version of this statement.)

A pleasant aspect is that these properties are proved without any reference to `visitf_dep` (§4.2). They represent the essence of the argument that "recursive depth-first search is correct", without any reference to actual code.

4.4. Wrapping Up

There remains to: (1) prove that the function `visitf_dep` satisfies the specification `visitf_spec` and (2) re-package the algorithm as a function `visitf` of non-dependent type `state -> V -> state`.

Item (1) is roughly the process of extracting proof obligations out of the code of §4.2 and using the three key properties of §4.3 to check that these obligations are met. Using a tool such as Why3, this would come almost for free. In Coq, the process is fairly difficult: the inner loop requires an auxiliary lemma, which is proved by induction; and the fact that `visitf_dep` has a dependent type complicates everything. Item (2) is straightforward.

In the end, we find that `visitf` is correct in the following sense:

```
forall s v s',
visitf s v = s' ->
visitf_spec s (singleton v) s'.
```

A similar result is proved about `fold_left visitf`, which visits a list of vertices in succession.

The OCaml code obtained by extraction for `visitf_dep` is 11 lines and is identical to what an OCaml programmer would write by hand.

5. Related Work

Neumann [4] presents an executable DFS algorithm, expressed in Isabelle/HOL, in iterative style. The algorithm is generic: it is parameterized by a state, whose type is user-defined, and by three functions that allow the user to specify how the state should be altered when a vertex is first discovered, re-discovered, and fully processed. An abortion mechanism is built into the algorithm. The algorithm assigns a pair of timestamps to every vertex. The properties satisfied by these timestamps are established, in the style of Cormen *et al.* [2], and the algorithm is proved correct, in the sense that it discovers all (and only) reachable vertices. (This seems to be a limited specification, which says nothing of the user-defined state computed by the algorithm.) On top of this generic algorithm, Neumann develops several variants of Nested DFS, an algorithm that detects the existence of a cycle that contains at least one "accepting" vertex, and proves their correctness. In its present state, our work in §4 is more modest.

Lammich [3] presents an Isabelle/HOL formalization of Gabow's algorithm for computing the strongly connected components. He extends it to an algorithm for the emptiness check of generalized Büchi automata: this requires checking (on the fly) whether there exists a reachable, non-trivial component that contains at least one vertex of every acceptance class. From a pedagogical point of

view, Kosaraju's algorithm seems simpler and more modular than Gabow's, as it is built on top of depth-first search. Presumably, Lammich's choice of Gabow's algorithm is motivated by performance considerations, although this is not discussed.

6. Conclusion

In summary, the main contributions of this paper are (1) a machine-checked version of Wegener's proof of Kosaraju's algorithm, together with a hand-written account (§2, §3) and (2) a clean separation between the mathematics of depth-first search (§2, §3) and its Coq implementation (§4).

Within §4, we have isolated the specification `visitf_spec` and established several of its properties independently of the code of `visitf_dep`. This separates the areas where Coq shines (mathematical reasoning) and those where it seems clumsy (defining general recursive functions).

The Coq code is available online [5]. The material in §2 and §3 represents about 1000 non-blank, non-comment lines, not counting a library of basic facts and tactics about sets and relations. The algorithm of §4 represents about 300 non-blank, non-comment lines.

There are many directions for future work. (1) Prove a stronger result about our depth-first search algorithm: show that the call `fold_left visitf rs (void, Empty _)` produces a forest `vs` such that the predicate `rs` *orders* `vs` (§3.1) holds. (2) Define and verify an executable version of Kosaraju's algorithm. This requires (1) above. (3) Study a tail-recursive version of the depth-first search algorithm. (4) Define a generic depth-first search algorithm, and apply it to a number of simple problems, such as cycle detection, topological sort, graph reversal, etc.

In the long run, we would like to see Coq equipped with a library of elementary graph algorithms, in the style of `ocamlgraph`. They would be valuable building blocks in the development of future verified applications.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009.

[3] P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. volume 8558, pages 325–340, July 2014.

[4] R. Neumann. A framework for verified depth-first algorithms. In *ATx/WInG: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation*, volume 17 of *EPiC Series*, pages 36–45. EasyChair, June 2012.

[5] F. Pottier. Depth-first search and strong connectivity in Coq. `http://gallium.inria.fr/~fpottier/dfs/dfs.tar.gz`, Sept. 2014.

[6] M. Sharir. A strong connectivity algorithm and its applications to data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[7] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[8] I. Wegener. A simplified correctness proof for a well-known algorithm computing strongly connected components. *Information Processing Letters*, 83(1):17–19, 2002.

7. Proofs

Proof of Lemma 1. By immediate inductions. □

Proof of Lemma 2. By an immediate induction. □

Proof of Lemma 3. By immediate inductions, using Lemma 2. □

Proof of Lemma 4. By Lemmas 1 and 3. □

Proof of Lemma 5. By induction over the *dfs* hypothesis. □

Proof of Lemma 6. By induction over the *dfs* assumption. The base case is immediate; the inductive case relies on Lemma 2. □

Proof of Lemma 7. By Lemma 3, o is $i \cup \mathit{support}(\vec{v})$. Thus, in order to establish the goal, it is (necessary and) sufficient to prove separately $E(i) \subseteq o$ and $E(\mathit{support}(\vec{v})) \subseteq o$. The former holds by hypothesis; the latter follows from Lemma 6. □

Proof of Lemma 8. By Lemmas 2 and 7. □

Proof of Lemma 9. Clearly, w is a member of the set $i \cup \mathit{support}(\frac{w}{\vec{w}})$ Thus, in order to establish the goal, it suffices to show that this set is closed. This is easy: by Lemma 3, this set is exactly the set of marked vertices after constructing the tree $\frac{w}{\vec{w}}$; thus, by Lemma 8, it is closed. □

Proof of Lemma 10. As in the proof of the previous lemma, it suffices to show that the set $\mathit{support}(\frac{w}{\vec{w}} :: \vec{v})$ is closed, this time with respect to $\bar{E}$. By Lemma 3, this set is just $\neg i$. (We rely on a slightly subtle detail: the use of $\mathcal{V}$ in the hypothesis $\mathit{dfs}\ (i)\ \frac{w}{\vec{w}} :: \vec{v}\ (\mathcal{V})$ means that all vertices are marked at the end. So, any vertex that is not in $\mathit{support}(\frac{w}{\vec{w}} :: \vec{v})$ must be in i.) Thus, we have to prove that $\neg i$ is closed with respect to $\bar{E}$, or in other words, that i is closed with respect to E. This is true by hypothesis. □

Proof of Lemma 11. By definition, $\mathit{scc}(w)$ is the intersection of the set of vertices that w can reach and the set of vertices that can reach w. Thus, the goal is:

$$E^\star(\{w\}) \cap \bar{E}^\star(\{w\}) \subseteq \mathit{support}(\frac{w}{\vec{w}})$$

This follows immediately from Lemmas 9 and 10, by intersection. □

Proof of Lemma 12. Because $\mathit{scc}(w)$ is defined as $E^\star(\{w\}) \cap \bar{E}^\star(\{w\})$, the goal boils down to $\bar{E}^\star(\{w\}) \subseteq E^\star(\{w\})$. By Lemma 10, $\bar{E}^\star(\{w\})$ is a subset of $\mathit{support}(\frac{w}{\vec{w}})$. Furthermore, by Lemma 5, the vertices in a tree are reachable from its root, so $\mathit{support}(\frac{w}{\vec{w}})$ is a subset of $E^\star(\{w\})$. □

Proof of Lemma 13. We prove the inclusion $E^\star(\{w\}) \subseteq \mathit{support}(\frac{w}{\vec{w}})$. (The reverse inclusion follows from Lemma 5.) Lemma 9 yields $E^\star(\{w\}) \subseteq i \cup \mathit{support}(\frac{w}{\vec{w}})$, so the goal boils down to proving that the sets $E^\star(\{w\})$ and i are disjoint. This follows from the fact that w lies outside i (Lemma 3) and from the hypothesis that $\neg i$ is closed. □

Proof of Lemma 14. This proof relies on two auxiliary lemmas:

1. If in the graph E, $\vec{v}$ goes into $\vec{w}$, then in the graph E', the same holds, and furthermore, in E', no edge goes to a masked vertex. Formally, $E(\vec{v}) \subseteq \vec{w}$ implies $E'(\vec{v}) \subseteq \vec{w} \setminus M$.

2. If no vertex in the tree $\frac{w}{\vec{w}}$ is masked, and if in the graph E, w covers $\vec{w}$, then in the graph E', the same holds. Formally, $\{w\} \cup \mathit{support}(\vec{w}) \subseteq \neg M$ and $\mathit{roots}(\vec{w}) \subseteq E(\{w\})$ imply $\mathit{roots}(\vec{w}) \subseteq E'(\{w\})$.

The proof of the main result can then be carried out by induction, using these auxiliary lemmas as well as Lemma 1 where needed. □

Proof of Lemma 15. By induction over the *dfs* hypothesis. The one non-trivial case arises when the forest $\vec{v}$ begins with a tree $\frac{w}{\vec{w}}$ and w is not masked, i.e., $w \notin M$. In that case, in order to apply FILTER-VISIBLE, we must prove that no descendant of w can be masked, i.e., $\mathit{support}(\vec{w}) \subseteq \neg M$. We assume, by way of contradiction, that some vertex x is in $\mathit{support}(\vec{w})$ and in M. Then, we find that there is a path $r\ E^\star\ w\ E^\star\ x\ E^\star\ r$. Indeed,

- $r\ E^\star\ w$ follows from the fact that r reaches the roots of $\vec{v}$, i.e., $\mathit{support}(\vec{v}) \subseteq E^\star(\{r\})$.
- $w\ E^\star\ x$ follows from the fact that x is a descendant of w, i.e., $x \in \mathit{support}(\vec{w})$, and Lemma 5.
- $x\ E^\star\ r$ follows from the fact that x is masked, i.e. $x \in M$, and our assumption that M is $\mathit{scc}(r)$.

The existence of this path implies $w \in \mathit{scc}(r)$, that is, $w \in M$. Contradiction. □

Proof of Lemma 16. The relation $\cdot \geq_M \cdot$ is compatible with forest concatenation. Thus, the goal boils down to proving $\vec{v} \geq_M \vec{v}$ and $\frac{r}{\vec{w}} \geq_M \vec{w}'$. The former sub-goal follows from $\mathit{support}(\vec{v}) \subseteq \neg M$, which itself follows from the fact that M is contained in the tree $\frac{r}{\vec{w}}$ (Lemma 11). The latter sub-goal follows from Lemma 15, using Lemmas 1 and 5 along the way. □

Proof of Lemma 17. In order to establish the desired statement, we formulate a strengthened statement, which is amenable to a proof by induction. This strengthened statement allows for the existence of a set M of so-called "masked" vertices, which is closed with respect to both E and $\bar{E}$, i.e., no edge leaves or enters this set. Intuitively, M represents the components that have been already identified and removed.

The strengthened statement is as follows: under the following five hypotheses,

$$E(M) \subseteq M \tag{1}$$

$$\bar{E}(M) \subseteq M \tag{2}$$

$$\mathit{dfs}_E\ (M)\ f_1\ (\mathcal{V}) \tag{3}$$

$$\mathit{dfs}_{\bar{E}}\ (M)\ f_2\ (\mathcal{V}) \tag{4}$$

$$\mathit{rev}(\mathit{post}(f_1))\ \mathit{orders}\ f_2 \tag{5}$$

the forest f_2 is an SCC forest for the graph E.

It is clear that the original statement follows from the strengthened statement by letting $M = \emptyset$.

As suggested by Wegener [8], the proof of the strengthened statement is by induction. We use structural induction over the forest f_2. In the base case, f_2 is empty and the result is immediate. So, we focus on the inductive case, where f_2 is a non-empty forest. Throughout this proof, the reader is encouraged to refer to Figure 3. For some r, $\vec{w}_2$, and $\vec{v}_2$, we have:

$$f_2 = \frac{r}{\vec{w}_2} :: \vec{v}_2 \tag{6}$$

According to (3) and (4), the forests f_1 and f_2 have a common non-empty support, namely $\neg M$. This implies that the hypothesis (5) cannot follow from the rules ORDERED-NIL or ORDERED-SKIP. (The premise $r \notin \mathit{support}(\vec{v})$ in ORDERED-SKIP is exploited here.) Thus, it must follow from ORDERED-ROOT. This means that the root r of the first tree in f_2 is the head of the reverse post-order of f_1. For some $\vec{r}$, we have:

$$\mathit{rev}(\mathit{post}(f_1)) = r :: \vec{r} \tag{7}$$

$$\vec{r}\ \mathit{orders}\ \vec{v}_2 \tag{8}$$

Equation (7) implies that the forest f_1 is non-empty (this, we knew already) and that the vertex r is the root of its last tree. Thus, for some $\vec{v}_1$ and $\vec{w}_1$, we have:

$$f_1 = \vec{v}_1 +\!\!+ \frac{r}{\vec{w}_1} \tag{9}$$

As r is the root of the last tree in the forest f_1, Lemma 12 states that the component of r is exactly its reverse closure:

$$scc(r) = \bar{E}^\star(\{r\}) \tag{10}$$

As r is the root of the first tree in the forest f_2, Lemma 13 states that the reverse closure of r is exactly (the support of) the tree $\frac{r}{\vec{w}_2}$:

$$\bar{E}^\star(\{r\}) = support(\frac{r}{\vec{w}_2}) \tag{11}$$

By combining equations (10) and (11), we find that we have established an important intermediate result: the first tree in the forest f_2 is indeed a component of the graph E. There remains to prove that the remainder of this forest, namely $\vec{v}_2$, is an SCC forest.

In order to do this, the idea is to remove the component $scc(r)$ from the graph, as well as from the forests f_1 and f_2, and argue that the smaller graph and forests thus obtained are suitable arguments for the main induction hypothesis.

Let us write E' for the graph E deprived of all edges whose source or destination lies in $scc(r)$. Let us write M' for the set $M \cup scc(r)$. We prove that $\vec{v}_2$ is an SCC forest with respect to E'; this clearly implies that it is also an SCC forest with respect to E.

By Lemma 16, the component $scc(r)$ forms a prefix of (the last tree of) the forest f_1, which means that it can be removed, yielding a well-formed forest. Thus, for some $\vec{w}'_1$, we have:

$$\begin{gathered} f_1 = \\ \vec{v}_1 +\!\!+ \frac{r}{\vec{w}_1} \geq_{scc(r)} \vec{v}_1 +\!\!+ \vec{w}'_1 \end{gathered} \tag{12}$$

By (12) and by Lemma 14, our diminished forest is a DFS forest with respect to our diminished graph:

$$dfs_{E'}\ (M \cup scc(r))\ \vec{v}_1 +\!\!+ \vec{w}'_1\ (\mathcal{V}) \tag{13}$$

Now, in order to establish our goal, namely that $\vec{v}_2$ is an SCC forest with respect to E', we apply the main induction hypothesis to the graph E', to the set of masked vertices M', and to the forests $\vec{v}_1 +\!\!+ \vec{w}'_1$ and $\vec{v}_2$. We must now check that the five requirements of the induction hypothesis hold:

1. The set M' is closed with respect to the graph E', i.e., $E'(M') \subseteq M'$. This is trivial, as the vertices in $scc(r)$ carry no edges.
2. M' is reverse-closed with respect to E', i.e., $\bar{E}'(M') \subseteq M'$. This is again trivial.
3. This requirement is exactly (13).
4. $\vec{v}_2$ is a valid DFS forest with respect to E' and M', i.e., $dfs_{\bar{E}'}\ (M')\ \vec{v}_2\ (\mathcal{V})$. This follows by Lemma 14 from $dfs_{\bar{E}}\ (M')\ \vec{v}_2\ (\mathcal{V})$, which itself (in light of the fact that $M' = M \cup support(\frac{r}{\vec{w}_2}))$ follows directly from our initial assumption about $\vec{v}_2$.
5. The last requirement is $rev(post(\vec{v}_1 +\!\!+ \vec{w}'_1))$ *orders* $\vec{v}_2$. As the predicate "$\vec{r}$ *orders* $\vec{v}$" is insensitive to the presence in the list $\vec{r}$ of vertices that are not in the support of $\vec{v}$, and in the light of (12), this goal is equivalent to $rev(post(f_1))$ *orders* $\vec{v}_2$, which follows from (7), (8), and Ordered-Skip.

This concludes the proof. □