Formal methods and assisted proofs: application to strongly connected components algorithms

Département Informatique — Parcours Recherche
Tuteur : Stephan Merz

Vincent TRÉLAT

April 1, 2022

***

Contents

1 Introduction

1.1 Academic context

This research work was carried out as part of my curriculum at the french École des Mines de Nancy. All documents such as codes or source papers are available on a GitHub repository.

1.2 Formal methods

Formal methods are a field of computer science related to mathematical logic and reasoning. The whole purpose of the discipline is to ensure by a logical proof that a given algorithm is not only correct on its domain of definition, but also to find – or define – that domain. Formal methods find applications in a variety of fields, both concrete, such as the railway industry or self-driving cars, and abstract, such as computational architecture.
Altough a formal proof lies first on paper, the real formalisation starts when proofs are mechanised in a proof assistant.

1.3 Isabelle (HOL)

> Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.

isabelle.in.tum.de

Isabelle is a really powerful low-level proof assistant coming with a higher order logic (HOL) proving environment making the proofs easily readible and comprehensible without adding any abstract overlay. The term "assistant" designates the fact that Isabelle has numerous tools allowing various automations in the proofs such as a theorem seeker or an automatic solver.

1.4 Isabelle by example

The following example is a good introduction to the use of Isabelle.

2 Models and representation

2.1 Nodes

Vertices of a graph can be represented as nodes. A node $\mathcal{N}$ is a simple data structure composed of an index, a boolean value telling if it has already been visited and two integer values *num* and *lowlink* whose role will be explained later. In the following, the aforesaid attributes will be referred to through the following notation:
Let $\mathcal{N}$ be a node. The attributes of $\mathcal{N}$ can be accessed via `N.index`, `N.visited`, `N.num` and `N.lowlink`. This notation will be applied to any object that lends itself to it.

2.2 Graphs and their representation

A graph $\mathcal{G}$ is the data $(\mathcal{V}, \mathcal{E})$ where:

- $\mathcal{V}$ is a set of vertices
- $\mathcal{E} \subseteq \{(x, y) \in V^2\}$ is a set of edges[1].

Vertices will often be called nodes and edges will be represented through adjency lists for each node.
Let us give an example. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the graph represented on figure 1. Thus, $\mathcal{V} = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and

$$\begin{aligned}
\mathcal{E} = \{&(0,0),(0,1),(0,2),(0,3),\\
&(1,4),(1,7),\\
&(3,0),(3,1),(3,2),(3,5),\\
&(4,3),(4,6),\\
&(5,6),\\
&(6,3),\\
&(7,6)\}
\end{aligned}$$

This representation being somewhat long, adjency lists can be used instead and therefore it gives:

```
G.adjency = [[0,1,2,3],[4,7],[0,1,2,5],[3,6],[6],[3],[6]]
```

Thus, for all $i \in \{0, \ldots, 7\}$, `G.adjency[i]` is the list of nodes to which node i is connected – *i.e.* there is an directed edge from node i to every node of `G.adjency[i]`.

[1] Note the use of the couple (x, y) and not the pair $\{x, y\}$ that makes the graph directed.

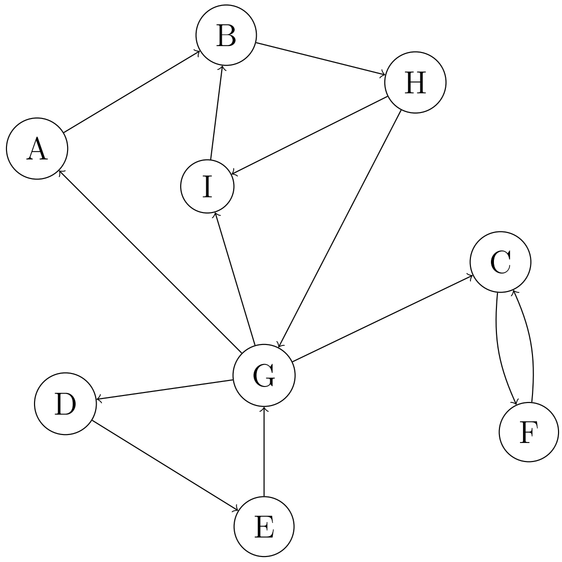


Figure 1: Representation of $\mathcal{G}$

3 Formalisation

3.1 Strongly connected components

3.1.1 Directed graphs

Definition 1. For two vertices x and y of $\mathcal{V}$, the relation "has an edge to" is noted "$\Rightarrow$" such that

$$(x, y) \in \mathcal{E} \iff x \Rightarrow y$$

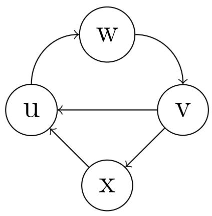
By extension, if there is a path from x to y with more than one edge, the same notation is kept for the sake of simplicity. The reflexive and transitive closure of the relation $\Rightarrow$ is noted $\Rightarrow^*$.

Definition 2. Let $\mathcal{G} = (\mathcal{V}\ ,\ \mathcal{E}\ )$ be an directed graph. $\mathcal{C} \subseteq \mathcal{V}$ is a strongly connected component of $\mathcal{G}$ if:

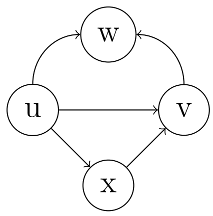
$$\forall x, y \in \mathcal{C}, (x \Rightarrow y) \wedge (y \Rightarrow x)$$

i.e. there is a path between every x and y in $\mathcal{C}$.

3.1.2 Examples



(a) Strongly connected graph



(b) Not strongly connected graph

Figure 2: Basic example of what is a small SCC

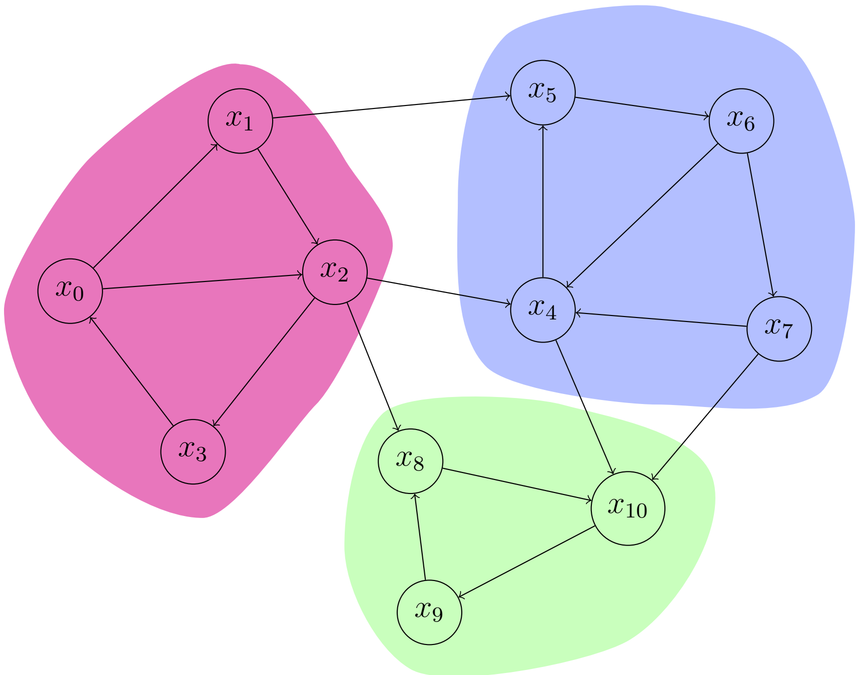


Figure 3: Example of a graph where each colored set of node is a maximal set of SCC

3.2 Order of traversal and backtracking edges

3.2.1 DFS and *num* value

Tarjan's SCC algorithm basically lies on a depth-first search. The figure 5 shows an example of a DFS traversal on a simple directed graph.

The previously mentioned figure also displays in red the `num` value which represents the order in which the nodes are visited in the graph during the DFS.

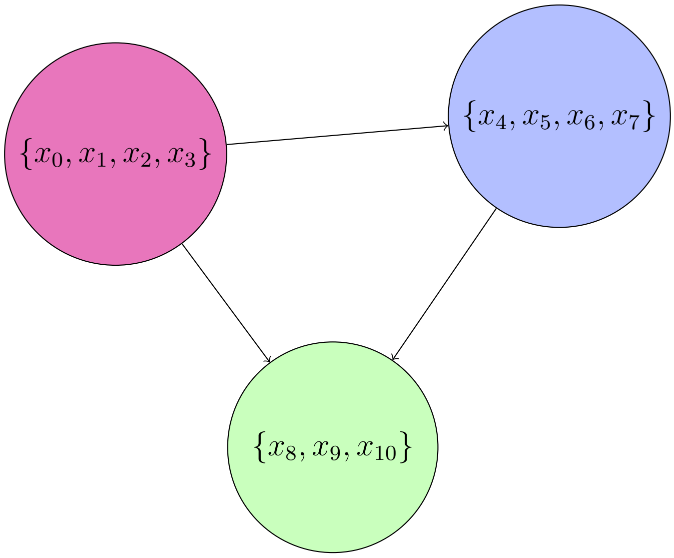


Figure 4: Reduced visualization of the graph represented if figure 3

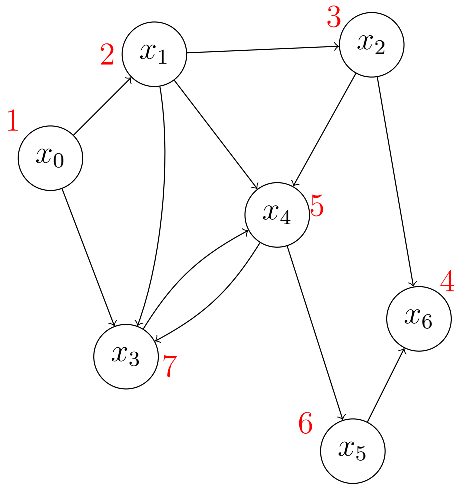


Figure 5: Example of a DFS

3.2.2 Backtracking edges

Definition 3. Given a graph $\mathcal{G}$ and an order of traversal in this graph, *i.e.* each node of $\mathcal{G}$ has a unique value $\texttt{num} \in [0,\ |\mathcal{V}|]$ and two nodes u and v, there is a backtracking edge from v to u if:

$$\begin{cases} \texttt{u.num} < \texttt{v.num} \\ v \ \Rightarrow\ u,\ \textit{i.e.}\ \ (v,u) \in \mathcal{E} \end{cases}$$

In this case, the backtracking edge from v to u is represented by $v \hookrightarrow u$.

3.3 Lowlink value

3.3.1 Definition

Informally, the `lowlink` value of a node represents the `num` value of the attachment node of their SCC, *i.e.* the `num` value of the entrance node in the corresponding SCC.

A more formal definition would be the following:

Definition 4. Let u be a node.

$$\texttt{u.lowlink} = \min\{\texttt{w.num} \mid \exists\ v \in \mathcal{V},\ u \Rightarrow v \hookrightarrow w\}$$

3.3.2 Example

Let $\mathcal{G}$ be the graph given in fig. 6. The order of traversal of the graph is given by the value *num* for each node of $\mathcal{G}$. The *lowlink* value is also displayed.

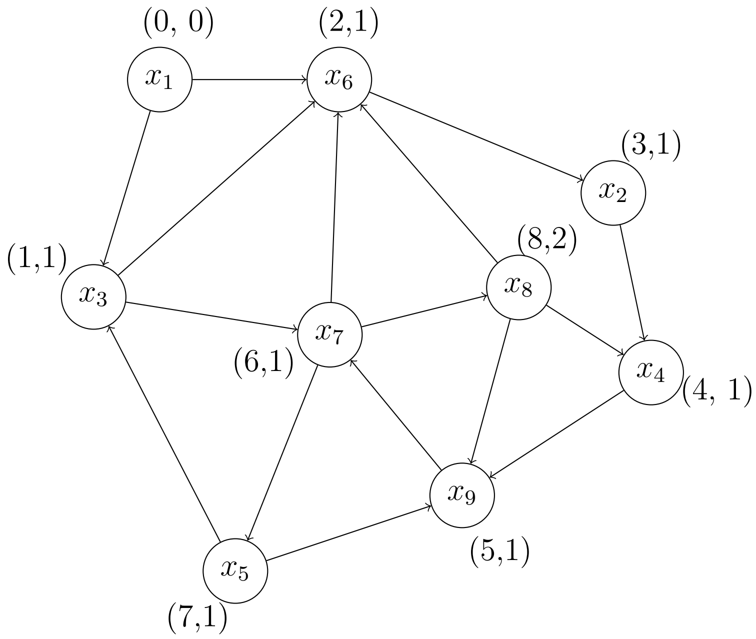


Figure 6: A DFS was performed through $\mathcal{G}$ from x_1 and next to each node is represented the couple of value (*num*, *lowlink*)

Now, backtracking edges can be highlighted w.r.t. the order of traversal. In fig. 7, they are represented as red dashed arrows.
Knowing the backtracking edges, all *lowlink* values can be computed[2].
Let us take x_8 as an example: its *lowlink* is equal to 2, which actually means that x_6[3] is its anchor – or attachment node – in their SCC, namely the green one. Indeed, x_8 is alone in its equivalence class[4], and from all nodes linked by one of the backtracking edges of x_8, x_6 has the minimum value *num*. Likewise, x_3 is its own attachment node in the green SCC since it is the first node visited when performing the DFS.

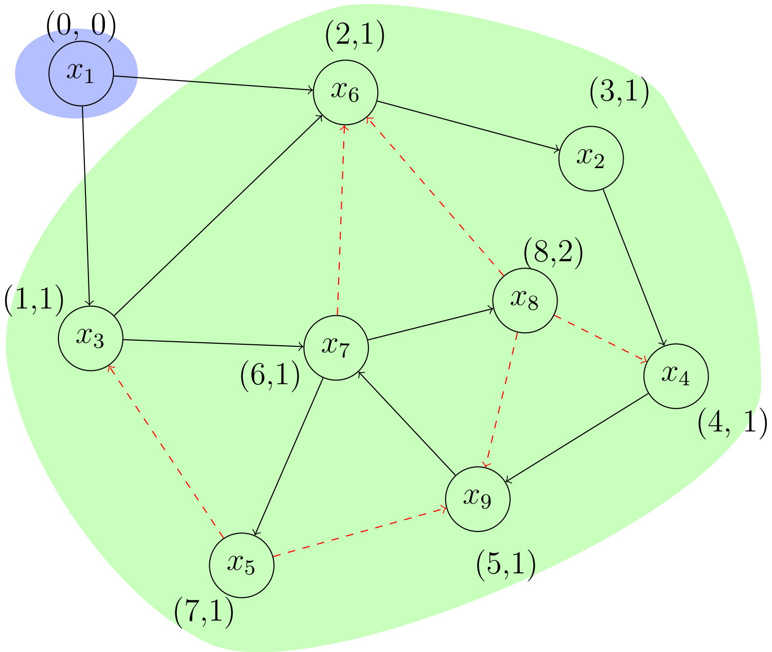


Figure 7: Same graph as in fig. 6 whose backtracking edges have been represented with red dashed arrows and SCCs have been highlighted

Then, SCCs can be easily found, namely $\{x_1\}$ and $\{x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$, as shown in fig. 7.

[2]In fact, they are refreshed during the DFS.
[3]Because `x6.num` $= 2$
[4]for the relation $\Rightarrow$

4 Tarjan's algorithm

Tarjan's algorithm is an efficient on-the-fly SCC computing algorithm [1]. It basically perfoms a DFS while updating the *num* and *lowlink* values. All nodes are stored in a stack during the traversal until a backtracking edge is found. In this case, the *lowlinks* are computed and all nodes are unstacked and saved in a SCC until a node verifying the equality between its *num* and its *lowlink* – which has to occur – is found. Then, the DFS goes on. The whole process is written in the following algorithm 1.

It can be shown that every node and edge are visited only once so the algorithm can achieve a linear complexity, *i.e.* $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

4.1 Description of Tarjan's algorithm

Algorithm 1: Tarjan's algorithm

```
Data: A graph G = (V, E)
Result: A partition SCCs of V where each element of SCCs is a SCC of G
Initialize an empty stack R;
Initialize an empty set SCCs;
Let num := 0;
forall v ∈ V do
    if v.num is unefined then
        SCC(v);
    end
end
function SCC: v ∈ V → None
    v.num = num;
    v.lowlink = num;
    increment num;
    Push v in R;
    v.onStack = true;
    forall w ∈ POST(v) do
        if w.num is undefined then
            v.lowlink = min(v.lowlink, w.lowlink);
        end
        else if w.onStack then
            v.lowlink = min(v.lowlink, w.num);
        end
    end
    if v.lowlink = v.num then
        Initialize an empty set currentSCC;
        repeat
            Let w := R.pop();
            w.onStack = false;
            currentSCC = currentSCC ∪ {w};
        until v.num ≠ w.num;
        SCCs = SCCs ∪ currentSCC;
    end
```

5 A sequential set-based algorithm

5.1 Formalisation

Definition 5 (SCC mapping)**.** In the following algorithm, the SCCs are progressively tracked in a collection of disjoint sets through a map $\mathcal{S} : \mathcal{V} \longrightarrow \mathcal{P}(\mathcal{V})$, where $\mathcal{P}(\mathcal{V})$ is the powerset of $\mathcal{V}$, s.t. the following invariant is maintained:

$$\forall v, w \in \mathcal{V}, w \in \mathcal{S}(v) \Longleftrightarrow \mathcal{S}(v) = \mathcal{S}(w) \tag{1}$$

Remark 1. In particular, $\forall v \in \mathcal{V}, v \in \mathcal{S}(v)$.

Definition 6 (SCC union)**.** Let $\textsc{Unite}$ be the function taking as parameters a map $\mathcal{S}$ as defined previously and two vertices u and v of $\mathcal{V}$ such that $\textsc{Unite}(\mathcal{S}, u, v)$ merges the two mapped sets $\mathcal{S}(u)$ and $\mathcal{S}(v)$ and maintains the invariant (1) by updating the function $\mathcal{S}$.

Let us give an example:
Let $\mathcal{V} = \{u, v, w\}$ such that there is the following mapping: $\mathcal{S}(u) = \{u\}$ and $\mathcal{S}(v) = \mathcal{S}(w) = \{v, w\}$.
Then, $\textsc{Unite}(\mathcal{S}, u, v) = \mathcal{S}(u) = \mathcal{S}(v) = \mathcal{S}(w) = \{u, v, w\}$.

Definition 7 (Successors set for a node)**.** Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $v \in \mathcal{V}$. The set of successors of v in $\mathcal{G}$ is $\textsc{Post}(v)$ such that:

$$\forall w \in \textsc{Post}(v), (v, w) \in \mathcal{E}$$

5.2 The algorithm

See [3] for the original paper.

Algorithm 2: Sequential set-based SCC algorithm

Data: A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a starting node v_0;

Result: A partition `SCCs` of $\mathcal{V}$ where each element of `SCCs` is a maximal set of strongly connected components of $\mathcal{G}$;

```
Initialize an empty set EXPLORED;
Initialize an empty set VISITED;
Initialize an empty stack R;
setBased(v_0);
function setBased: v ∈ 𝒱 → None
    VISITED := VISITED ∪ {v};
    R.push(v);
    foreach w ∈ POST(v) do
        if w ∈ EXPLORED then
            continue;
        end
        else if w ∉ VISITED then
            setBased(w);
        end
        else
            while 𝒮(v) ≠ 𝒮(w) do
                r := R.pop();
                UNITE(𝒮, r, R.top());
            end
        end
    end
    if v = R.top() then
        report SCC 𝒮(v);
        EXPLORED := EXPLORED ∪ 𝒮(v);
        R.pop();
    end
```

5.3 Informal proof

Note that this proof is said informal only because it is not formally automated. Both logical and mathematical arguments developped below are absolutely relevant.

Lemma 1. (First invariant)

$$\forall x, y \in \mathtt{R}, x \neq y \implies \mathcal{S}(x) \cap \mathcal{S}(y) = \varnothing$$

Note the misuse of the set notation $x, y \in \mathtt{R}$ which just means that x and y are in the stack $\mathtt{R}$.

Proof. Let $x \in \mathcal{V}$ be the following node to be visited during the execution of the algorithm 2: x is pushed in $\mathtt{R}$. Let $y \in \textsc{Post}(x)$. There are two cases:

- y has not been visited yet, *i.e.* $y \notin \textsc{Visited}$. Thus, a DFS-like traversal is performed from y, so y is pushed in $\mathtt{R}$ and $\mathcal{S}(y) = \{y\}$ because y is alone in its equivalence class for the moment since it has not been visited yet. Therefore, $\mathcal{S}(x) \cap \mathcal{S}(y) = \varnothing$.

- y has already been visited, *i.e.* $y \in \textsc{Visited}$. Then, y was already pushed in $\mathtt{R}$ before x. Let $(x_i)_{1 \leq i \leq n}$ be the first nodes of the stack s.t. $x_0 = x$ and $x_n = y$.
  In order to avoid writing $\mathtt{R} = [\dots, y, \dots, x]$, let us define $\widetilde{R}$ the stack containing the first n nodes in $\mathtt{R}$, s.t. $\widetilde{R} = [y, \dots, x] = [x_n, \dots, x_0]$.
  Let us consider the worst case, *i.e.* when

  $$\forall\ 1 \leq i \leq n,\ \mathcal{S}(x_i) = \{x_i\}$$

  So, the while loop has to go down to y because all partial SCCs are disjoint. As the length of the stack $\mathtt{R}$ is bounded by $|\mathcal{V}|$, the algorithm terminates.
  x_0 is first unstacked and both $\mathcal{S}(x_0)$ and $\mathcal{S}(\mathtt{R.top()}) = \mathcal{S}(x_1)$ are then united. The current state of $\mathcal{S}$ and $\widetilde{R}$ is:

  $$\begin{cases} \mathcal{S} = \{\{x_0, x_1\}, \{x_2\}, \dots, \{x_n\}, \dots\} \\ \widetilde{R} = [x_n, \dots, x_1] \end{cases}$$

  Then, x_1 is unstacked and $\mathcal{S}(x_1)$ and $\mathcal{S}(x_2)$ are then united, so that:

  $$\begin{cases} \mathcal{S} = \{\{x_0, x_1, x_2\}, \{x_3\}, \dots, \{x_n\}, \dots\} \\ \widetilde{R} = [x_n, \dots, x_2] \end{cases}$$

  Finally (by induction), $\mathcal{S} = \{x_0, \dots, x_n\}$ and $\widetilde{R} = [y]$, *i.e.* $\mathcal{S}(x) = \mathcal{S}(y)$. It is important to notice that $x = x_0, x_1, \dots, x_{n-1}$ are no longer in the stack, so this operation kept the invariant true.

$\blacksquare$

Lemma 2.

$$\biguplus_{v \in \mathtt{R}} \mathcal{S}(v) = \textsc{Live} := \textsc{Visited} \setminus \textsc{Explored}$$

Proof. The disjointness of all on-stack partial SCCs is given by lemma 1. Nodes from $\textsc{Visited} \setminus \textsc{Eexplored}$ are in $\mathtt{R}$ because they are being processed. So, $\textsc{Live} \subseteq \mathtt{R}$.
By L.6-7 of algorithm 2, $\textsc{Visited} \subseteq \mathtt{R}$.
L.9-10 ensure that no explored node is pushed in $\mathtt{R}$.
L.24-25 keep the invariant by unstacking explored nodes from $\mathtt{R}$, so $\mathtt{R} \cap \textsc{Explored} = \varnothing$. Thus, $\mathtt{R} = \textsc{Visited} \setminus \textsc{Explored} = \textsc{Live}$. $\blacksquare$

Corollary 2.1.

$$\forall v \in \textsc{Live}, \exists!\ r \in \mathtt{R} \cap \mathcal{S}(v), \mathcal{S}(v) = \mathcal{S}(r)$$

Proof. Let $v \in \textsc{Live} = \biguplus_{v \in \mathtt{R}} \mathcal{S}(v)$. v is in a unique partial SCC $\mathscr{S} := \mathcal{S}(v)$. Because of lemma 1, there cannot exist $x \neq y \in \mathtt{R}$ s.t. $\mathcal{S}(x) = \mathcal{S}(y) = \mathscr{S}$. Thus, there exists a unique $x \in \mathtt{R}$ s.t. $\mathcal{S}(x) = \mathscr{S}$ (and $x \in \mathtt{R} \cap \mathscr{S}$). $\blacksquare$

Corollary 2.2.

$$\forall v \in \mathcal{V}, \forall w \in \textsc{Post}(v), w \in \textsc{Live} \implies \exists w' \in \mathtt{R}, \mathcal{S}(w') = \mathcal{S}(w)$$

Proof. Holds because of corollary 2.1. $\blacksquare$

Remark 2. In the algorithm 2, this property is held by L.16-18. These lines also illustrate how the algorithm "reads" the SCCs. Corollary 2.2 shows that when the mapped representatives of the top two nodes of $\mathtt{R}$ are united (until $\mathcal{S}(w') = \mathcal{S}(v) = \mathcal{S}(w)$ since w' has a path to v), then all united components are in the same SCC.

Remark 3. Because $\mathtt{R}$ only contains exactly one representative for each partial SCC (corollary 2.1), after each step of the main loop – *i.e.* the DFS – every partial SCC is actually maximal in the current set of visited nodes.

Theorem 1. The sequential algorithm 2 is correct, *i.e.* it returns a set of maximal SCCs.

Proof. Holds by remark 3. $\blacksquare$

5.4 Formal proof

Since the informal proof seems to be consistant, the formal – automated – proof can be written in Isabelle (HOL) based on the basis of the reasonning developped above.

5.4.1 Environment setup

The first definitions should be the different structures used in the algorithm. In particular, a record containing all the sets needed and described in the pseudo-code of algorithm 2 :

```
record 'v env =
  𝓢 ::  "'v ⇒ 'v set"
  explored ::  "'v set"
  visited ::  "'v set"
  sccs ::  "'v set set"
  stack ::  "'v list"
```

The following lines define a graph structure and some useful natural relations :

```
locale graph =
  fixes vertices ::  "'v set" and successors ::  "'v ⇒ 'v set"
  assumes vfin:  "finite vertices"
  and sclosed:  "∀x ∈ vertices.  successors x ⊆ vertices"

abbreviation edge where
  "edge x y ≡ y ∈ successors x"

inductive reachable where
  reachable_refl[iff]:  "reachable x x"
| reachable_succ[elim]:  "⟦edge x y; reachable y z⟧ ⟹ reachable x z"
```

Those two relations are `edge`, which simply translates the property for two nodes of being linked by an edge and `reachable`, which is the binary relation $\Rightarrow^*$ defined in section 3.1.1.
In order to be able to use those relations in the proofs later, it is essential to prove a list of lemmas, namely all the different natural properties that Isabelle cannot deduce[5] from nothing[6]. For instance, the following lemmas are essential.

[5]That is an abuse of language. The idea is for example that for the moment, there is no formal link between `edge` and `reachable`. The goal is to formalize it so Isabelle is logically able to both use and simplify some results in the proofs.

[6]There is actually a theorem fetcher that is particularly useful to find a basic set of lemmas.

```
lemma reachable_edge:  "edge x y ⟹ reachable x y"
    by auto
```
Mathematical writing: $\forall x, \forall y, x \Rightarrow y \Longrightarrow x \Rightarrow^* y$

```
lemma succ_reachable:
    assumes "reachable x y" and "edge y z"
    shows"reachable x z"
    using assms by induct auto
```
Mathematical writing: $\forall x, \forall y, \forall z, (x \Rightarrow^* y \wedge y \Rightarrow z) \Longrightarrow x \Rightarrow^* z$

```
lemma reachable_trans:
    assumes y:  "reachable x y" and z:  "reachable y z"
    shows "reachable x z"
    using assms by induct auto
```
Mathematical writing: $\forall x, \forall y, \forall z, (x \Rightarrow^* y \wedge y \Rightarrow^* z) \Longrightarrow x \Rightarrow^* z$

As the formal proofs will enventually deal with strongly connected components, it is also essential to formally define SCCs. For the purpose of the proof, the property of being a SCC is called `sub_scc` and being a *maximal* SCC is called `is_scc` :

```
definition is_subscc where
    "is_subscc S ≡∀ x ∈ S. ∀ y ∈ S. reachable x y"
```
Mathematical writing: A set S is a SCC if $\forall x \in S, \forall y \in S, x \Rightarrow^* y$

```
definition is_scc where
    "is_scc S ≡ S ≠ {} ∧ is_subscc S
    ∧ (∀ S'. S ⊆ S' ∧ is_subscc S' ⟶ S' = S)"
```
Mathematical writing: A non-empty SCC S is maximal if for all SCC S', $S \subseteq S' \Longrightarrow S' = S$

Once again, there are some lemmas to prove, such as telling Isabelle when an element can be added to a SCC, or that two vertices that are reachable from each other are in the same SCC, or that two SCCs having a common element are identical.

5.4.2 Ordering relation

In the proof, a ordering relation[7] noted $\bullet \preceq \bullet$ in $\bullet$ will be needed on the stack. Let x and y be two nodes and R be a stack. Informally, x precedes y in R if x was pushed in R before y (see figure 8).

[7] In fact, a total order is being defined on stacks.

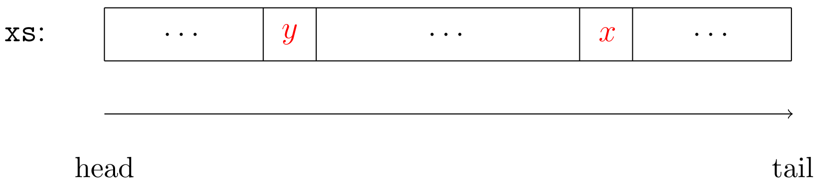


Figure 8: The ordering relation on stacks

Definition 8 (Ordering relation)**.** Let x and y be two nodes and xs be a stack.

$$x \preceq y \text{ in } xs \equiv \exists\ h,\ \exists\ r,\ (xs = h@[x]@r) \wedge (y \in [x]@r)$$

The idea is to later use the following property: if $x \preceq y$ in xs, then $x \Rightarrow^* y$. It is defined in Isabelle as follows:

```
definition precedes ("_ ≼ _ in _" [100,100,100] 39) where
    "x ≼ y in xs ≡ ∃h r.  xs = h @ (x # r) ∧ y ∈ set (x # r)"
```

All the different properties (*i.e.* lemmas) which follow this definition in the Isabelle implementation are detailed in the natural mathematical writing below:

Let x, y, z be three nodes, and let xs, ys, zs be three lists of nodes representing stacks. By abuse of language, if an element is on a stack, it is in the set of elements contained in the stack so the following statement can be written: x is on $xs \iff x \in xs$. However, xs in not seen as the set representing xs since an element may occur several times in a stack. The operator @ denotes the concatenation and operates on two lists: $[x_0, \ldots, x_n]@[y_0, \ldots, y_m] = [x_1, \ldots, x_n, y_1, \ldots, y_m]$.

(i) $x \preceq y \text{ in } xs \Longrightarrow (x \in xs) \wedge (y \in xs)$

(ii) $y \in [x@xs \Longrightarrow (x \in xs) \wedge (y \in xs)$

(iii) $x \neq z \Longrightarrow (x \preceq y \text{ in } ([z]@zs) \Longrightarrow x \preceq y \text{ in } zs)$

(iv) $(y \preceq x \text{ in } ([x]@xs)) \wedge (x \notin xs) \Longrightarrow (x = y)$

(v) $y \in (ys@[x]) \Longrightarrow y \preceq x \text{ in } (ys@[x]@xs)$

(vi) $(x \preceq x \text{ in } xs) = (x \in xs)$

(vii) $x \preceq y \text{ in } xs \Longrightarrow x \preceq y \text{ in } (ys@xs)$

(viii) $x \notin ys \Longrightarrow (x \preceq y \text{ in } (ys@xs) \iff x \preceq y \text{ in } xs)$

(ix) $x \preceq y$ in $xs \Longrightarrow x \preceq y$ in $(xs@ys)$

(x) $y \notin ys \Longrightarrow x \preceq y$ in $(xs@ys) \Longleftrightarrow x \preceq y$ in xs

(xi)
$(x \preceq y \text{ in } xs) \wedge (y \preceq z \text{ in } xs) \wedge \underbrace{(\forall\, 0 \leq i < j \leq \text{length}(xs), xs[i] \neq xs[j])}_{\text{all elements on } xs \text{ are distinct}} \Longrightarrow x \preceq z \text{ in } xs$

5.4.3 Implementation of the algorithm

Now that the environment is set up, the actual algorithm – seen as a function – can be implemented.
Since Isabelle does not support loops, the implementation will be split into two mutually recursive functions. The main function is called `dfs` and takes its name after the Depth First Search algorithm because the algorithm 2 roughly consists in a deep traversal of a graph. The second function is called `dfss` and represents the *while* loop of the algorithm 2. The two functions are mutually recursive because they recursively call each other. In particular, `dfss` will call either itself or `dfs`, depending on the case.

```
function dfs ::  "'v ⇒ 'v env ⇒ 'v env" and
         dfss::  "'v ⇒ 'v set ⇒ 'v env ⇒ 'v env" where
"dfs v e =
    (let e1 = e⦇visited := visited e ∪ {v}, stack := (v # stack e)⦈;
         e' = dfss v (successors v) e1
    in if v = hd(stack e')
         then e'⦇sccs:=sccs e' ∪ 𝒮 e' v, explored:=explored e' ∪ (𝒮 e' v),
stack:=tl(stack e')⦈
         else e')"
| "dfss v vs e =
    (if vs = {} then e
    else (let w = SOME x.  x ∈ vs
         in (let e' = (if w ∈ explored e then e
                  else if w ∉ visited e then dfs w e
                  else unite v w e)
              in dfss v (v - {w}) e')))"
  by pat_completeness (force+)
```

The two last keywords require explanations as well : `pat_completeness` stands for *pattern completeness* and ensures that there is no missing patterns. The keyword `force` is used[8] to help Isabelle know – by proving it – that both `dfs` and `dfss` are actually functions and that those functions are well defined with respect to the usual logical and mathematical meaning.

[8]Here, `auto` cannot terminate because of the mutual recursion.

References

[1] R. Chen, C. Cohen, J.-J. Lévy, S. Merz, L. Théry, *Formal Proofs of Tarjan's Strongly Connected 2 Components Algorithm in Why3, Coq and Isabelle*, 2019

[2] V. Bloemen, A. Laarman, J. van de Pol, *Multi-Core On-The-Fly SCC Decomposition*, 2016

[3] V. Bloemen, *Strong Connectivity and Shortest Paths for Checking Models*, 2019