STRONGLY CONNECTED COMPONENTS OF DIRECTED HYPERGRAPHS

XAVIER ALLAMIGEON

Abstract. We study the problem of determining strongly connected components (SCCs) of directed hypergraphs. The main contribution is an algorithm computing the terminal strongly connected components (i.e., SCCs which do not reach any other components than themselves). The time complexity of the algorithm is almost linear, which is a significant improvement over the known methods which are quadratic time. This also proves that the problems of (i) testing strong connectivity, (ii) and determining the existence of a sink, can be both solved in almost linear time in directed hypergraphs. We also highlight an important discrepancy between the reachability relations in directed hypergraphs and graphs. We establish a superlinear lower bound on the size of the transitive reduction of the reachability relation in directed hypergraphs, showing that it is combinatorially more complex than in directed graphs. We also prove linear time reductions from combinatorial problems on the subset partial order, in particular from the well-studied problem of finding all minimal sets among a given family, to the problem of computing the SCCs in directed hypergraphs.

1. Introduction

Directed hypergraphs consist in a generalization of directed graphs, in which the tail and the head of the arcs are sets of vertices. Directed hypergraphs have a very large number of applications, since hyperarcs naturally provide a representation of implication dependencies. Among others, they are used to solve several problems related to satisfiability in propositional logic, in particular relative to Horn formulas, see for instance [AI91, AFFG97, GP95, GGPR98, Pre03]. They also appear in problems relative to network routing [Pre00], functional dependencies in database theory [ADS83], model checking [LS98], chemical reaction networks [¨Ozt08], transportation networks [NP89, NPG98], and more recently, algorithmics of convex polyhedra in tropical algebra [AGG10, All09a].

Many algorithmic aspects of directed hypergraphs have been studied, in particular optimization related ones, such as determining shortest paths [NP89, NPA06], maximum flows, minimum cardinality cuts, or minimum weighted hyperpaths (we refer to the surveys of Ausiello et al. [AFF01] and of Gallo et al. [GLPN93] for a comprehensive list of contributions). Naturally, some problems raised by the reachability relation in directed hypergraphs have also been studied. For instance, determining the set of the vertices reachable from a given vertex is known to be solvable in linear time in the size of the directed hypergraph (see for instance [GLPN93]).

In directed graphs, many other problems are known to be efficiently solvable, e.g. in linear time, such as testing acyclicity or strong connectivity, computing the

1In the sequel, the underlying model of computation is the Random Access Machine.
strongly connected components, determining a topological sorting among them, etc. Surprisingly, the analogues of these elementary problems in directed hypergraphs have not received any particular attention (as far as we know). Unfortunately, none of the direct graph algorithms can be straightforwardly extended to directed hypergraphs. The main reason is that the reachability relation of hypergraphs does not have the same structure: for instance, establishing that a given vertex \( u \) reaches another vertex \( v \) generally involves vertices which do not reach \( v \).

Contributions. In this paper, we tackle some reachability problems relative to directed hypergraphs and their strongly connected components.

Section 3 presents an almost linear time algorithm able to determine the terminal strongly connected components of a hypergraph (a component is said to be terminal when it reaches no other components than itself). As discussed below, this improves the existing quadratic approaches. This also shows that the following properties: (i) is a directed hypergraph strongly connected? (ii) does the hypergraph admit a sink (i.e. a vertex reachable from all vertices)? can also be determined in almost linear time. The algorithm proceeds by iterating two steps. The first one consists in finding some (terminal) SCCs of an underlying directed graph. In the second step, each of these components is collapsed to a single vertex, which makes appear new arcs in the digraph underlying to the hypergraph. The two steps are carefully combined to gain efficiency. Moreover, an elaborate instrumentation is settled to determine the new arising arcs, without sacrificing the time complexity. A complete example of execution trace of the algorithm is provided in Appendix A.

Unfortunately, this algorithm cannot be extended to determine all strongly connected components with the same complexity. In fact, the contributions presented in Section 4 strongly suggest that the problem of computing the entire set of SCCs is harder in directed hypergraphs than in directed graphs. In particular, we prove a lower bound result which shows that the size of the transitive reduction of the reachability relation may be superlinear in the size of the directed hypergraph (whereas this is linearly upper bounded in the setting of directed graphs). We deduce a linear time reduction from the minimal set problem to the problem of computing the strongly connected components. Given a family \( \mathcal{F} \) of sets over a certain domain, the minimal set problem consists in determining all the sets of \( \mathcal{F} \) which are minimal for the inclusion. While it has received much attention, the best known algorithms are only subquadratic time.

Related Work. Reachability in directed hypergraphs has been defined in different ways in the literature, depending on the context and the applications. The reachability relation which is discussed here is basically the same as in [AN90, AI91, AFF01], but is referred to as \( B \)-reachability in [GLPN93, GP95]. It precisely captures the logical implication dependencies in Horn propositional logic, and also the functional dependencies in the context of relational databases. Some variants of this reachability relation have been introduced, e.g. in which any hyperpath has to be provided with a linear order over the alternating sequence of its vertices and hyperarcs [TT09]. These variants are beyond the scope of the paper.

As mentioned above, determining the set of the reachable vertices from a given vertex has been thoroughly studied. Gallo et al. provide in [GLPN93] a linear time algorithm. In a series of works [AN90, AI91, AFF01], Ausiello et al. also introduce online algorithms maintaining the set of reachable vertices, or hyperpaths between vertices, under hyperarc insertions/deletions.
To our knowledge, other reachability problems, such as topological sorting, determining strongly connected components or terminal ones, have not been specifically studied so far. Naturally, they can be solved in polynomial time by using the algorithms previously mentioned. For instance, given a directed hypergraph $\mathcal{H}$ with $n$ vertices, the whole graph of the reachability relation can be determined in $O(n \text{size}(\mathcal{H}))$ by $n$ calls to the algorithm of [GLPN93]. Computing the (terminal) strongly connected components of this graph precisely yields the (terminal) components of $\mathcal{H}$. However, this approach is obviously not optimal: for instance, when $\mathcal{H}$ coincides with a directed graph, we know that the problem can be simply solved in linear time.

Computing the transitive closure and reduction of a directed hypergraph has also been studied by Ausiello et al. in [ADSS86]. In their work, reachability relations between sets of vertices are also taken into account, in contrast with our present contribution in which we restrict to reachability relations between vertices. The notion of transitive reduction in [ADSS86] is also different from the one discussed here (Section 4.1). More precisely, the transitive reduction of [ADSS86] rather corresponds to the transitive reduction in [ADS86] rather corresponds to the transitive reduction in [ADS86] is also different from the one discussed here (Section 4.1). More precisely, the transitive reduction of [ADSS86] rather corresponds to the transitive reduction in [ADS86] is also different from the one discussed here (Section 4.1). More precisely, the transitive reduction of [ADSS86] rather corresponds to the transitive reduction in [ADS86] is also different from the one discussed here (Section 4.1). More precisely, the transitive reduction of [ADSS86] rather corresponds to the transitive reduction in [ADS86] is also different from the one discussed here (Section 4.1). More precisely, the transitive reduction of [ADSS86] rather corresponds to the transitive closure of $\mathcal{H}$. This also leads to a notion of hyperpaths: a hyperarc $a$ is itself a pair $(T, H)$, of vertices $T$ and $H$ are both subsets of $V$. They respectively represent the tail and the head of $a$, and are also denoted by $T(a)$ and $H(a)$. Note that throughout this paper, the term hypergraph(s) will always refer to directed hypergraph(s).

The size of a directed hypergraph $\mathcal{H} = (V, A)$ is defined as $\text{size}(\mathcal{H}) = |V| + \sum_{(T, H) \in A}(|T| + |H|)$.

Given a directed hypergraph $\mathcal{H} = (V, A)$, and $u, v \in V$, then $v$ is said to be reachable from $u$ in $\mathcal{H}$, which will be denoted by $u \leadsto^\mathcal{H} v$, if $u = v$, or there exists a hyperarc $a = (T, H)$ such that $v \in H$ and all the elements of $T$ are reachable from $u$. This also leads to a notion of hyperpaths: a hyperpath from $u$ to $v$ in $\mathcal{H}$ is a sequence of $p$ hyperarcs $a_1, \ldots, a_p \in A$ satisfying $T(a_i) \subset \cup_{j=1}^{i-1} H(a_j)$ for all $i = 1, \ldots, p + 1$, with the conventions $H(a_0) = \{u\}$, and $T(a_{p+1}) = \{v\}$. The hyperpath is said to be minimal if none of its subsequences is a hyperpath from $u$ to $v$.

The strongly connected components (SCCs for short) of a directed hypergraph $\mathcal{H}$ are the equivalence classes of the relation $\equiv^\mathcal{H}$, defined by $u \equiv^\mathcal{H} v$ if $u \leadsto^\mathcal{H} v$ and $v \leadsto^\mathcal{H} u$.

If $f$ is a function from $V$ to an arbitrary set, the image of the directed hypergraph $\mathcal{H}$ by $f$ is the hypergraph, denoted $f(\mathcal{H})$, of vertices $f(V)$ and of hyperarcs $\{(f(T(a)), f(H(a))) \mid a \in A\}$.

Example 1. Consider the directed hypergraph depicted in Figure 1. Its vertices are $u, v, w, x, y, t$, and its hyperarcs $a_1 = (\{u\}, \{v\})$, $a_2 = (\{v\}, \{w\})$, $a_3 = (\{w\}, \{u\})$, $a_4 = (\{v, w\}, \{x, y\})$, and $a_5 = (\{w, y\}, \{t\})$. A hyperarc is represented as a bundle of arcs. It is decorated with a solid disk portion when its tail contain several vertices.

To our knowledge, other reachability problems, such as topological sorting, determining strongly connected components or terminal ones, have not been specifically studied so far. Naturally, they can be solved in polynomial time by using the algorithms previously mentioned. For instance, given a directed hypergraph $\mathcal{H}$ with $n$ vertices, the whole graph of the reachability relation can be determined in $O(n \text{size}(\mathcal{H}))$ by $n$ calls to the algorithm of [GLPN93]. Computing the (terminal) strongly connected components of this graph precisely yields the (terminal) components of $\mathcal{H}$. However, this approach is obviously not optimal: for instance, when $\mathcal{H}$ coincides with a directed graph, we know that the problem can be simply solved in linear time.

Computing the transitive closure and reduction of a directed hypergraph has also been studied by Ausiello et al. in [ADSS86]. In their work, reachability relations between sets of vertices are also taken into account, in contrast with our present contribution in which we restrict to reachability relations between vertices. The notion of transitive reduction in [ADSS86] is also different from the one discussed here (Section 4.1). More precisely, the transitive reduction of [ADSS86] rather corresponds to the transitive closure of $\mathcal{H}$. This also leads to a notion of hyperpaths: a hyperarc $a$ is itself a pair $(T, H)$, of vertices $T$ and $H$ are both subsets of $V$. They respectively represent the tail and the head of $a$, and are also denoted by $T(a)$ and $H(a)$. Note that throughout this paper, the term hypergraph(s) will always refer to directed hypergraph(s).

The size of a directed hypergraph $\mathcal{H} = (V, A)$ is defined as $\text{size}(\mathcal{H}) = |V| + \sum_{(T, H) \in A}(|T| + |H|)$.

Given a directed hypergraph $\mathcal{H} = (V, A)$, and $u, v \in V$, then $v$ is said to be reachable from $u$ in $\mathcal{H}$, which will be denoted by $u \leadsto^\mathcal{H} v$, if $u = v$, or there exists a hyperarc $a = (T, H)$ such that $v \in H$ and all the elements of $T$ are reachable from $u$. This also leads to a notion of hyperpaths: a hyperpath from $u$ to $v$ in $\mathcal{H}$ is a sequence of $p$ hyperarcs $a_1, \ldots, a_p \in A$ satisfying $T(a_i) \subset \cup_{j=1}^{i-1} H(a_j)$ for all $i = 1, \ldots, p + 1$, with the conventions $H(a_0) = \{u\}$, and $T(a_{p+1}) = \{v\}$. The hyperpath is said to be minimal if none of its subsequences is a hyperpath from $u$ to $v$.

The strongly connected components (SCCs for short) of a directed hypergraph $\mathcal{H}$ are the equivalence classes of the relation $\equiv^\mathcal{H}$, defined by $u \equiv^\mathcal{H} v$ if $u \leadsto^\mathcal{H} v$ and $v \leadsto^\mathcal{H} u$.

If $f$ is a function from $V$ to an arbitrary set, the image of the directed hypergraph $\mathcal{H}$ by $f$ is the hypergraph, denoted $f(\mathcal{H})$, of vertices $f(V)$ and of hyperarcs $\{(f(T(a)), f(H(a))) \mid a \in A\}$.

Example 1. Consider the directed hypergraph depicted in Figure 1. Its vertices are $u, v, w, x, y, t$, and its hyperarcs $a_1 = (\{u\}, \{v\})$, $a_2 = (\{v\}, \{w\})$, $a_3 = (\{w\}, \{u\})$, $a_4 = (\{v, w\}, \{x, y\})$, and $a_5 = (\{w, y\}, \{t\})$. A hyperarc is represented as a bundle of arcs. It is decorated with a solid disk portion when its tail contain several vertices.
Applying the recursive definition of reachability from $u$ discovers the vertex $v$, then $w$, which leads to the two vertices $x$ and $y$ through the hyperarc $a_4$, and finally $t$ through $a_5$. It can be checked that $t$ is reachable from $u$ through the hyperpath $a_1, a_2, a_4, a_5$ (which is minimal). As mentioned in Section 1, some vertices play the role of “auxiliary” vertices when determining reachability. In our example, establishing that $t$ is reachable from $u$ first requires to establish that $y$ is reachable from $u$, while $y$ does not reach $t$. This is an important difference with directed graphs, in which proving that $t$ is reachable from $u$ would only involve vertices both reachable from $u$ and reaching $t$.

Observe that all the notions introduced in this section are generalizations of their analogues on directed graphs. Indeed, any digraph $G = (V, A)$ can be equivalently seen as a directed hypergraph $H = (V, \{\{(u, v)\} \mid u, v \in A\})$. Then the reachability relations on $G$ and $H$ coincide, and $G$ and $H$ both have the same size.

3. Computing terminal strongly connected components

In this section, we describe an algorithm which determines all terminal SCCs of a directed hypergraph. Given a hypergraph $H$ of vertices $V$, a component $C$ is said to be terminal if for any $u \in C$ and $v \in V$, $u \rightarrow_H v$ implies $v \in C$. In other words, a SCC is terminal when it does not reach any component except itself.

3.1. Principle of the algorithm. First observe that a directed graph $\text{graph}(H) = (V, A')$ can be associated to any directed hypergraph $H = (V, A)$, by defining $A' = \{(t, h) \mid \{(t, H) \in A \text{ and } h \in H\}\}$. The directed graph $\text{graph}(H)$ is generated by the simple hyperarcs of $H$, i.e. the elements $a \in A$ such that $|T(a)| = 1$. We first point out a remarkable special case in which the terminal SCCs of $H$ and $\text{graph}(H)$ are equal:

**Proposition 1.** Let $H$ be a directed hypergraph such that each terminal SCC of $\text{graph}(H)$ is reduced to a singleton. Then $H$ and $\text{graph}(H)$ have the same terminal SCCs.

This statement is a consequence of the fact that any SCC of $H$ is precisely the union of some SCCs of $\text{graph}(H)$:

**Lemma 2.** Let $H$ be a directed hypergraph. Each strongly connected component $C$ of $H$ is of the form $\cup_i C'_i$ where the $C'_i$ are the SCCs of $\text{graph}(H)$ such that $C \cap C'_i \neq \emptyset$. 

![Figure 1. A directed hypergraph](image-url)
Proof. Consider \( u \in C \). Then there exists a component \( C' \) of \( \text{graph}(\mathcal{H}) \) such that \( u \in C' \) (since the Sccs of \( \text{graph}(\mathcal{H}) \) form a partition of the set of the vertices), and obviously \( C \cap C' \neq \emptyset \).

Conversely, suppose that \( C' \) is a SCC of \( \text{graph}(\mathcal{H}) \) such that \( C \cap C' \neq \emptyset \). Let \( u \in C \cap C' \). Then for any \( v \in C' \), we have \( u \twoheadrightarrow_{\text{graph}(\mathcal{H})} v \twoheadrightarrow_{\text{graph}(\mathcal{H})} u \), so that \( u \twoheadrightarrow_{\mathcal{H}} v \twoheadrightarrow_{\mathcal{H}} u \), hence \( v \in C \). \( \square \)

Proof (Proposition 3). First suppose that \( \{u\} \) is a terminal SCC of \( \text{graph}(\mathcal{H}) \). Suppose that there exists \( v \neq u \) such that \( u \twoheadrightarrow_{\mathcal{H}} v \). Consider a hyperarc \( a_1, \ldots, a_p \) from \( u \) to \( v \) in \( \mathcal{H} \). Then there must be a hyperarc \( a_i \) such that \( T(a_i) = \{u\} \) and \( H(a_i) \neq \{u\} \) (otherwise, the hyperpath is a cycle and \( v = u \)). Let \( w \in H(a_i) \setminus \{u\} \). Then \( (u, w) \) is an arc of \( \text{graph}(\mathcal{H}) \). Since \( \{u\} \) is a terminal SCC of \( \text{graph}(\mathcal{H}) \), this enforces \( w = u \), which is a contradiction. Hence \( \{u\} \) is a terminal SCC of \( \mathcal{H} \).

Conversely, consider a terminal SCC \( C \) of \( \mathcal{H} \). Let \( u \in C \), and let \( D \) be the SCC of \( \text{graph}(\mathcal{H}) \) containing \( u \). Consider \( D' \) a terminal SCC of \( \text{graph}(\mathcal{H}) \) such that \( D \twoheadrightarrow_{\text{graph}(\mathcal{H})} D' \), and let \( C' \) be a SCC of \( \mathcal{H} \) such that \( D' \cap C' \neq \emptyset \). By Lemma 2 we have \( D' \subseteq C' \). It follows that \( C \twoheadrightarrow_{\mathcal{H}} C' \), hence \( C = C' \) as \( C \) is terminal. Thus, \( D' \subseteq C \), and since \( D' \) is a singleton, it also forms a SCC of \( \mathcal{H} \) using the first part of the proof. This shows \( D' = C \) (since the Sccs of \( \mathcal{H} \) form a partition of the set of vertices), so that \( C \) is a terminal SCC of \( \text{graph}(\mathcal{H}) \). \( \square \)

The following proposition ensures that, in a directed hypergraph, merging two vertices of a same SCC does not alter the reachability relation:

**Proposition 3.** Let \( \mathcal{H} = (\mathcal{V}, \mathcal{A}) \) be a directed hypergraph, and let \( x, y \in \mathcal{V} \) such that \( x \equiv_{\mathcal{H}} y \). Consider the function \( f \) mapping any vertex distinct from \( x \) and \( y \) to itself, and both \( x \) and \( y \) to a same vertex \( z \) (with \( z \notin \mathcal{V} \setminus \{x, y\} \)). Then \( u \twoheadrightarrow_{\mathcal{H}} v \) if, and only if, \( f(u) \twoheadrightarrow_{f(\mathcal{H})} f(v) \).

**Proof.** Let \( \mathcal{H}' = f(\mathcal{H}) \). Suppose that \( s \twoheadrightarrow_{\mathcal{H}} t \). Observe that if \( X, Y \) are subsets of \( \mathcal{V} \), \( f(X) \subseteq f(Y) \) as soon as \( X \subseteq Y \), and \( f(X \cup Y) \subseteq f(X) \cup f(Y) \). Therefore, if \( a_1, \ldots, a_p \) is a hyperpath from \( s \) to \( t \), then:

\[
T(a_i) \subseteq \{s\} \cup H(a_1) \cup \cdots \cup H(a_{i-1}) \quad \text{for all } 1 \leq i \leq p
\]

\[
t \in H(a_p)
\]

so that:

\[
f(T(a_i)) \subseteq \{f(s)\} \cup f(H(a_1)) \cup \cdots \cup f(H(a_{i-1})) \quad \text{for all } 1 \leq i \leq p
\]

\[
f(t) \in f(H(a_p))
\]

It follows that \( f(s) \twoheadrightarrow_{f(\mathcal{H})} f(t) \).

Conversely, suppose that \( f(t) \) is reachable from \( f(s) \) in \( \mathcal{H}' \), and that \( f(t) \neq f(s) \) (the case \( f(t) = f(s) \) is trivial). Let \( H_0 = \{s\} \) and \( T_{p+1} = \{t\} \).

By definition, there exist \( a_1 = (T_1, H_1), \ldots, a_p = (T_p, H_p) \) in \( A \) such that for each \( i \in \{1, \ldots, p+1\} \), \( f(T_i) \subseteq f(H_0) \cup \cdots \cup f(H_{i-1}) \).

Also note that for any subset \( s \) of \( \mathcal{V} \), \( f(s) = s \) in \( s \cap \{x, y\} = \emptyset \) and \( f(s) = s \cup \{z\} \setminus \{x, y\} \) otherwise. In particular, as soon as \( z \notin f(s) \), \( f(s) \) coincides with \( s \).

Besides, \( f(s) \setminus \{z\} \subseteq s \subseteq f(s) \setminus \{z\} \cup \{x, y\} \).

Two cases can be distinguished:

(a) suppose that \( z \) does not belong to any \( f(H_j) \), so that \( f(H_j) = H_j \). Similarly, for each \( i \geq 1 \), \( f(T_i) \) does not contain \( z \), hence \( f(T_i) = T_i \). Besides, \( T_i \subseteq H_0 \cup \cdots \cup H_{i-1} \) for each \( i \), so that is is straightforward that \( f(s) \twoheadrightarrow_{\mathcal{H}'} f(t) \).
on a hypergraph $H$

merging several vertices of a same $Scc$
d
each time the $Scc$

$(b)$ now, if $z$ in one of the $f(H_j)$, let $k$ be the smallest integer such that $z \in f(H_k)$. Say for instance that $x \in H_k$. Let $(T'_1, H'_1), \ldots, (T'_q, H'_q)$ be taken from a hyperpath from $x$ to $y$ in $H$.

When $i \leq k$, $f(T_i)$ does not contain $z$, hence $f(T_i) = T_i$ and $T_i \subseteq f(H_0) \cup \cdots \cup f(H_{i-1}) = H_0 \cup \cdots \cup H_{i-1}$.

Besides, $T'_i = \{x\} \subseteq H_0 \cup \cdots \cup H_k$, and for each $i \in \{2, \ldots, q\}$, $T'_i \subseteq H_0 \cup \cdots \cup H_k \cup H'_1 \cup \cdots \cup H'_{i-1}$ since $x \in H_k$.

Finally, let us prove for $i \geq k + 1$ that $T_i \subseteq H_0 \cup \cdots \cup H_k \cup H'_1 \cup \cdots \cup H'_{q} \cup H_{k+1} \cup \cdots \cup H_{i-1}$. Clearly, $f(T_i) \setminus \{z\} \subseteq \bigcup_{j=0}^{i-1} f(H_j) \setminus \{z\}$. Besides, $x \in H_k$

and $y \in H'_q$, and since $T_i$ is included into $f(T_i) \setminus \{z\} \cup \{x, y\}$, then $T_i$ is also contained in $H_0 \cup \cdots \cup H_k \cup H'_1 \cup \cdots \cup H'_{q} \cup H_{k+1} \cup \cdots \cup H_{i-1}$.

It follows that $(T_i, H_i)_{i=1,\ldots,k}, (T'_i, H'_i)_{i=1,\ldots,q}, (T_i, H_i)_{i=k+1,\ldots,p}$ forms a hyperpath from $s$ to $t$ in $H$. \hfill \square

It follows that the terminal SCCs of $H$ and $f(H)$ are in one-to-one correspondence. These properties can be straightforwardly extended to the operation of merging several vertices of a same SCC simultaneously.

Using Propositions 1 and 3, we now sketch a method which computes the terminal SCCs in a directed hypergraph $H = (V, A)$. It performs several transformations on a hypergraph $H_{cur}$ whose vertices are labelled by subsets of $V$:

Starting from the hypergraph $H_{cur}$ image of $H$ by the map $u \mapsto \{u\}$,

(i) compute the terminal SCCs of the directed graph $\text{graph}(H_{cur})$.

(ii) if one of them, say $C$, is not reduced to a singleton, replace $H_{cur}$ by $f(H_{cur})$, where $f$ merges all the elements $U \subseteq C$ into the vertex $\bigcup_{U \subseteq C} U$. Then go back to Step 3.

(iii) otherwise, return the terminal SCCs of the directed graph $\text{graph}(H_{cur})$.

Each time the vertex merging step (Step 3) is executed, new arcs may appear in the directed graph $\text{graph}(H_{cur})$. This case is illustrated in Figure 2. In both sides, the arcs of $\text{graph}(H_{cur})$ are depicted in solid, and the non-simple arcs of $H_{cur}$ in dotted line. Note that the vertices of $H_{cur}$ contain subsets of $V$, but enclosing braces are omitted for readability. Applying Step 3 from vertex $u$ (left side) discovers a terminal SCC formed by $u, v$, and $w$ in the directed graph $\text{graph}(H_{cur})$. At Step 3 (right side), the vertices are merged, and the hyperarc $a_4$ is transformed into two graph arcs leaving the new vertex $\{u, v, w\}$.

The termination of this method is ensured by the fact that the number of vertices in $H_{cur}$ is strictly decreased each time Step 3 is applied. When the method is terminated, terminal SCCs of $H_{cur}$ are all reduced to single vertices, each of them labelled by subsets of $V$. Propositions 1 and 3 prove that these subsets are precisely the terminal SCCs of $H$.

3.2. Optimized algorithm. The sketch given in Section 3.1 is not optimal since a given vertex may be visited $O(|V|)$ times. To overcome this problem, we propose to incorporate the vertex merging step directly into an algorithm determining the terminal SCCs in directed graphs. The resulting algorithm on directed hypergraphs is given in Figure 3. Note that we suppose that the directed hypergraph $H$ is also provided with the lists $A_u$ of hyperarcs $a$ such that $u \in T(a)$, for each $u \in V^2$.

\footnote{These lists can be built in linear time in a preprocessing step.}
The algorithm consists of a main function TERMINALScc which initializes data, and then iteratively calls a visiting function VISIT on the vertices which have not been visited yet. Following the sketch given in Section 3.1 the function VISIT(u) repeats the following three tasks: (i) it recursively searches a terminal SCC in the underlying directed graph graph(H_{cur}), starting from the vertex u, (ii) once a terminal SCC is found, it performs a vertex merging step on it, (iii) and finally, it discovers the new graph arcs (if any) arising from the merging step.

Before discussing each of these three operations, we explain how the directed hypergraph \( H_{cur} \) is manipulated by the algorithm. First observe that the vertices of the hypergraph \( H_{cur} \) always form a partition of the initial set \( V \) of vertices. Instead of referring to them as subsets of \( V \), we use a union-find structure, which consists in three functions \{\textsc{Find}, \textsc{Merge}, \textsc{MakeSet}\} (see for instance [CSRL01, Chap. 21]):

- a call to \textsc{Find}(u) returns, for each original vertex \( u \in V \), the unique vertex of \( H_{cur} \) containing \( u \).
- two vertices \( U \) and \( V \) of \( H_{cur} \) can be merged by a call to \textsc{Merge}(U,V), which returns the new vertex.
- the “singleton” vertices \{u\} of the initial \( H_{cur} \) are created by the function \textsc{MakeSet}.

With this structure, each vertex of \( H_{cur} \) is represented by an element \( u \in V \), in which case it corresponds to the subset \{v \in V \mid \text{Find}(v) = u\}. Besides, the hypergraph \( H_{cur} \) is precisely the image of \( H \) by the function \text{Find}.

To avoid confusion, we denote the vertices of the hypergraph \( H \) by lower case letters, and the vertices of \( H_{cur} \) (and subsequently \text{graph}(H_{cur})) by capital ones. By convention, if \( u \in V \), \text{Find}(u) will correspond to the associated capital letter \( U \). Note that when an element \( u \in V \) has never been merged with another one, it satisfies \text{Find}(u) = u.

Discovering terminal SCCs in the directed graph \text{graph}(H_{cur}). This task is performed by the parts of the algorithm which are not shaded in gray. Similarly to Tarjan’s algorithm [Tar72], it uses a stack \( S \) and two arrays indexed by vertices, \text{index} and \text{low}. The stack \( S \) stores the vertices \( U \) of \text{graph}(H_{cur}) which are currently visited by VISIT. The array \text{index} tracks the order in which the vertices are visited, \text{i.e.} \text{index}[U] < \text{index}[V] if, and only if, \( U \) has been visited by VISIT before \( V \). The value \text{low}[U] is used to determine the minimal index of the visited vertices which are reachable from \( U \) in the digraph (see Line 44). A (not necessarily terminal) SCC \( C \) of \text{graph}(H_{cur}) is discovered when a vertex \( U \) satisfies \text{low}[U] = \text{index}[U]
Then $C$ consists of all the vertices stored in the stack $S$ above $U$. The vertex $U$ is the element of the SCC which has been visited first, and is called its root. Once the visit of the SCC is terminated, its vertices are collected into a set $\text{Finished}$ (Line 43).

Additionally, the algorithm uses an array $\text{is\_term}$ of booleans, allowing to track whether a SCC of $\text{graph}(\mathcal{H}_{\text{cur}})$ is terminal. A SCC will be terminal if, and only if, its root $U$ satisfies $\text{is\_term}[U] = \text{true}$. In particular, the boolean $\text{is\_term}[U]$ is set to $\text{false}$ as soon as $U$ is connected to a vertex $W$ located in a distinct SCC (Line 42) or satisfying $\text{is\_term}[W] = \text{false}$ (Line 45).

**Vertex merging step.** This step is performed from Lines 51 to 69, when it is discovered that the vertex $U = \text{FIND}(u)$ is the root of a terminal SCC in the digraph $\text{graph}(\mathcal{H}_{\text{cur}})$. All vertices $V$ which have been collected in that SCC are merged to $U$ (Line 60). Let $\mathcal{H}_{\text{new}}$ be the resulting hypergraph.

At Line 60, the stack $F$ is expected to contain the new arcs of $\text{graph}(\mathcal{H}_{\text{new}})$ leaving the newly “big” vertex $U$ (this point will be explained in the next paragraph). If it is empty, $\{U\}$ is a terminal SCC of $\text{graph}(\mathcal{H}_{\text{new}})$, hence also of $\mathcal{H}_{\text{new}}$ (Prop. 1). Otherwise, we go back to the beginning of Line 66 to discover terminal SCCs from the new vertex $U$ in the digraph $\text{graph}(\mathcal{H}_{\text{new}})$.
discovered, \textit{i.e.} without examining all the non-simple hyperarcs. The formal proof of this technique is provided in Appendix B.

During the execution of \texttt{Visit}(u), the local stack \( F \) is used to collect the hyperarcs which represent arcs leaving the vertex \texttt{Find}(u) in \texttt{graph}(H_{cur}).

Initially, when \texttt{Visit}(u) is called, the vertex \texttt{Find}(u) is still equal to \( u \). Then, the loop from Lines \( 23 \) to \( 35 \) iterates over the set \( A_u \) of the hyperarcs \( a \in A \) such that \( u \in T(a) \). At the end of the loop, it can be verified that \( F \) is indeed filled with all the simple hyperarcs leaving \( u = \texttt{Find}(u) \) in \( H_{cur} \), as expected.

Now the main difficulty is to collect in \( F \) the arcs which are added to the digraph \texttt{graph}(H_{cur}) after a vertex merging step. To overcome this problem, each non-simple hyperarc \( a \in A \) is provided with two auxiliary data:

\begin{itemize}
  \item a vertex \( r_a \), called the root of the hyperarc \( a \), and which is the first vertex of the tail \( T(a) \) to be visited by a call to \texttt{Visit},
  \item and a counter \( c_a \geq 0 \), which determines the number of vertices \( x \in T(a) \) which have been visited and such that \texttt{Find}(x) is reachable from \texttt{Find}(r_a) in the current digraph \texttt{graph}(H_{cur}).
\end{itemize}

These auxiliary data are maintained in the auxiliary data update step, from Lines \( 26 \) to \( 53 \). Initially, the root \( r_a \) of any hyperarc \( a \) is set to the special value \( \textit{undef} \).

The first time a vertex \( u \) such that \( a \in A_u \) is visited, it is assigned to \( u \) (see Line \( 26 \)). Besides, at the call to \texttt{Visit}(u), the counter \( c_a \) of each non-simple hyperarc \( a \in A_u \) is incremented, but only when \( R_a = \texttt{Find}(r_a) \) belongs to the stack \( S \) (see Line \( 35 \)). This is indeed a necessary and sufficient condition to the fact that \texttt{Find}(u) is reachable from \texttt{Find}(r_a) in the digraph \texttt{graph}(H_{cur}) (see Invariant \( B \) in Appendix B).

It follows from these invariants that, when the counter \( c_a \) reaches the threshold value \( |T(a)| \), all the vertices \( X = \texttt{Find}(x) \), for \( x \in T(a) \), are reachable from \( R_a \) in the digraph \texttt{graph}(H_{cur}) in a terminal \( \texttt{Scc} \) \( C \) of \texttt{graph}(H_{cur}). Then the aforementioned vertices \( X \) must all stand in the \( \texttt{Scc} \) \( C \) (since it is terminal). Therefore, when the vertex merging step is applied on this \( \texttt{Scc} \), the vertices \( X \) are merged into a single vertex \( U \). Hence, the hyperarc \( a \) necessarily generates new simple arcs leaving \( U \) in the new version of the digraph \texttt{graph}(H_{cur}).

Now let us verify that in this situation, \( a \) is correctly placed into \( F \) by our algorithm: as soon as \( c_a \) reaches the threshold \( |T(a)| \), \( a \) is placed into a temporary stack \( F_R \) associated to the vertex \( R_a \) (Line \( 31 \)). It is then emptied into \( F \) at Lines \( 52 \) or \( 55 \) during the vertex merging step.

\textbf{Example 2.} For example, in the left side of Figure \( 2 \) the execution of the loop from Lines \( 23 \) to \( 55 \) during the call to \texttt{Visit}(v) sets the root of the hyperarc \( a_4 \) to the vertex \( v \), and \( c_{a_4} \) to 1. Then, during \texttt{Visit}(w), \( c_{a_4} \) is incremented to 2 = \( |T(a_4)| \).

The hyperarc \( a_4 \) is therefore pushed on the stack \( F_v \) (because \( R_{a_4} = \texttt{Find}(r_{a_4}) = \texttt{Find}(v) = v \)). Once it is discovered that \( u, v, \) and \( w \) form a terminal \( \texttt{Scc} \) of \texttt{graph}(H_{cur}), \( a_4 \) is collected into \( F \) during the merging step. It then allows to visit the vertices \( x \) and \( y \) from the new vertex (rightmost hypergraph). A fully detailed execution trace is provided in Appendix A below.

Correctness and complexity. For sake of simplicity, we have not included in \texttt{TerminalScc} the step returning the terminal \( \texttt{Sccs} \). However, they can be easily built by examining each vertex (hence in time \( O(|V|) \)), as shown below:
Theorem 4. Let $\mathcal{H} = (\mathcal{V}, A)$ be a directed hypergraph. After the execution of $\textsc{TerminalScc}(\mathcal{H})$, the terminal SCCs are precisely formed by the sets $C_U = \{ v \in \mathcal{V} \mid \text{Find}(v) = U \text{ and is term}[U] = \text{true} \}$.

The proof of Th. 4, which is too long to be included here, is provided in Appendix B. It relies on successive transformations of intermediary algorithms to $\textsc{TerminalScc}$.

The complexity of $\textsc{TerminalScc}$ follows from the fact that we use disjoint-set forests with union by rank and path compression as union-find structure (CSRL01, Chapter 21). It allows to perform a sequence of $p$ operations $\text{MAKESET}$, $\text{FIND}$, or $\text{MERGE}$ in time $O(p \times \alpha(|\mathcal{V}|))$, where $\alpha$ is the very slowly growing inverse of the Ackermann function. For any practical value of $x$, $\alpha(x) \leq 4$. That is why the complexity of $\textsc{TerminalScc}$ is said to be almost linear in $\text{size}(\mathcal{H})$:

Theorem 5. Let $\mathcal{H} = (\mathcal{V}, A)$ be a directed hypergraph. Then the algorithm $\textsc{TerminalScc}(\mathcal{H})$ terminates in time $O(\text{size}(\mathcal{H}) \times \alpha(|\mathcal{V}|))$.

Proof. The analysis of the time complexity $\textsc{TerminalScc}$ depends on the kind of the instructions. We distinguish: (i) the operations on the global stacks $F_u$ and on the local stacks $F$, (ii) the call to the functions $\text{FIND}$, $\text{MERGE}$, and $\text{MAKESET}$, (iii) and the other operations, referred to as usual operations (by extension, their time complexity will be referred to as usual complexity). Also note that the function $\text{Visit}(u)$ is executed exactly once for each $u \in \mathcal{V}$ during the execution of $\textsc{TerminalScc}$. The complexity of each kind of operations is detailed thereafter:

(i) each operation on the stack (pop or push) is in $O(1)$. A given hyperarc is pushed on a stack of the form $F_u$ at most once during the whole execution of $\textsc{TerminalScc}$. Once it is popped from it, it will never be pushed on a stack of the form $F_v$ again. Similarly, a hyperarc is pushed on a local stack $F$ at most once, and after it is popped from it, it will never be pushed on any local stack $F'$ in the following states. Therefore, the total number of stack operations on the local and global stacks $F$ and $F_u$ is bounded by $4|\mathcal{V}|$. It follows that the corresponding complexity is $O(|\mathcal{V}|)$.

Consequently, the total number of iterations of the loop from Lines 35 to 47 occurring the whole execution of $\textsc{TerminalScc}$ is bounded by $\sum_{a \in A} |H(a)|$.

(ii) during the execution of $\textsc{TerminalScc}$, the function $\text{FIND}$ is called:

- exactly $|\mathcal{V}|$ times at Line 38,
- at most $\sum_{u \in \mathcal{V}} |A_u| = \sum_{a \in A} |T(a)|$ times at Line 27 (since during the call to $\text{Visit}(u)$, the loop from Lines 23 to 25 has exactly $|A_u|$ iterations),
- at most $\sum_{a \in A} |H(a)|$ at Line 29 (see above).

Hence it is called at most $\text{size}(\mathcal{H})$ times.

The function $\text{MERGE}$ is always called to merge two distinct vertices. Let $C_1, \ldots, C_p$ ($p \leq |\mathcal{V}|$) be the equivalence classes formed by the elements of $\mathcal{V}$ at the end of the execution of $\textsc{TerminalScc}$. Then $\text{MERGE}$ has been called at most $\sum_{i=1}^p (|C_i| - 1)$. Since $\sum_i |C_i| = |\mathcal{V}|$, $\text{MERGE}$ is executed at most $|\mathcal{V}| - 1$ times.

Finally, $\text{MAKESET}$ is called exactly $|\mathcal{V}|$ times. It follows that the total time complexity of the operations $\text{MAKESET}$, $\text{FIND}$ and $\text{MERGE}$ is $O(\text{size}(\mathcal{H}) \times \alpha(|\mathcal{V}|))$.

(iii) the analysis of the usual operations is split into several parts:
• the usual complexity TERMINALSCC without the calls to the function VISIT is clearly $O(|V| + |A|)$.

• during the execution of VISIT($u$), the usual complexity of the block from Lines 18 to 35 is $O(1) + O(|A_u|)$. Indeed, we suppose that the test at Line 28 can be performed in $O(1)$ by assuming that the stack $S$ is provided with an auxiliary array of booleans which determines, for each element of $V$, whether it is stored in $S$. Then the total usual complexity between Lines 18 and 35 is $O(\text{size}(\mathcal{H}))$ for a whole execution of TERMINALSCC.

• the usual complexity of the body of loop from Lines 38 to 47, without the recursive calls to VISIT, is clearly $O(1)$. As mentioned above, the total number of iterations of this loop is less than $\sum_{a \in A} |H(a)| \leq \text{size}(\mathcal{H})$. Therefore, the total usual complexity of the loop from Lines 36 to 48 is in $O(\text{size}(\mathcal{H}))$.

• the usual complexity of the loop between Lines 54 and 62 for a whole execution of TERMINALSCC is $O(|V|)$, since in total, it is iterated exactly the number of times the function MERGE is called.

• the usual complexity of the loop between Lines 52 and 56 for a whole execution of TERMINALSCC is $O(|V|)$, because a given element is placed at most once into Finished.

• if the two previous loops are not considered, less than 10 usual operations are executed in the block from Lines 49 to 66, all of complexity $O(1)$. The execution of this block either follows a call to VISIT or the execution of the goto statement (at Line 60). The latter is executed only if the stack $F$ is not empty. Since each hyperarc can be pushed on a local stack $F$ and then popped from it only once, it is executed $|A|$ in the worst case during the whole execution of TERMINALSCC. It follows that the usual complexity of the block from Lines 49 to 66 is $O(|V| + |A|)$ in total (excluding the loops previously discussed).

Summing all the complexities above proves that the time complexity of TERMINALSCC is $O(\text{size}(\mathcal{H}) \times \alpha(|V|))$. \qed

The space complexity of the algorithm TERMINALSCC is obviously linear in $\text{size}(\mathcal{H})$. An implementation is provided in the library TPLib \cite{All09b} (module Hypergraph), where the algorithm is used to efficiently characterize extreme points in tropical polyhedra \cite{AGG10}. It can be used independently of the rest of the library.\footnote{Obviously, the push and pop operations on the stack $S$ are still in $O(1)$ under this assumption.}

### 3.3. Determining some other properties in almost linear time.

Some properties can be directly determined from the terminal SCCs. Indeed, a directed hypergraph $\mathcal{H}$ admits a sink (i.e. a vertex reachable from all vertices) if, and only if, there it contains a unique terminal SCC. Besides, it is strongly connected when all vertices are contained in this latter component.

**Corollary 6.** Given a directed hypergraph $\mathcal{H}$, the following problems can be solved in almost linear time in $\text{size}(\mathcal{H})$: (i) is there a sink in $\mathcal{H}$? (ii) is $\mathcal{H}$ strongly connected?

\footnote{Note that in the source code, terminal SCCs are referred to as maximal SCCs.}
4. Combinatorial complexity of the reachability relation in directed hypergraphs

4.1. A lower bound on the size of the transitive reduction. Given a directed graph or a directed hypergraph, the reachability relation can be represented by the set of the couples \((x, y)\) such that \(x\) reaches \(y\). This is however a particularly redundant representation because of transitivity. Besides, in order to get a better idea of the intrinsic complexity of the reachability relation, we are rather interested in more economical representations. In fact, the reachability relation admits transitive reductions, which are defined as minimal binary relations having the same transitive closure.

In directed graphs, Aho et al. have shown in [AGU72] that all transitive reductions of the reachability relation have the same size (the size of a binary relation \(\mathcal{R}\) is the number of couples \((x, y)\) such that \(x \mathcal{R} y\)). This size is bounded by the size of the graph. Furthermore, a canonical transitive reduction can be defined, by choosing a total ordering over the vertices.

In directed hypergraphs, the existence of a canonical transitive reduction of the reachability relation can be similarly established, because reachability is still reflexive and transitive. However, we are going to show that its size may be superlinear in \(\text{size}(\mathcal{H})\) for some directed hypergraphs \(\mathcal{H}\).

These hypergraphs arise from the subset partial order. More specifically, given a family \(\mathcal{F}\) of distinct sets over a finite domain \(D\), the partial order induced by the relation \(\subseteq\) on \(\mathcal{F}\) is called the subset partial order over \(\mathcal{F}\). From this family, we build a corresponding directed hypergraph \(\mathcal{H}(\mathcal{F}, D)\). Each of its vertices is either associated to a set \(S \in \mathcal{F}\) or to a domain element \(x \in D\), and is denoted by \(v[S]\) or \(v[x]\) respectively. Besides, each set \(S\) is associated to two hyperarcs \(a[S]\) and \(a'[S]\). The hyperarc \(a[S]\) leaves the singleton \(\{v[S]\}\) and enters the set of the vertices \(v[x]\) such that \(x \in S\). The hyperarc \(a'[S]\) is defined inversely, leaving the latter set and entering \(\{v[S]\}\). An example is given in Figure 4.

---

5 Any finite reflexive and transitive relation \(\mathcal{R}\) can be seen as the reachability relation of a directed graph \(G\), whose arcs are the couples \((x, y)\) such that \(x \mathcal{R} y\). Then the transitive reduction of \(\mathcal{R}\) is defined as in [AGU72].
Lemma 7. Given \( S \in \mathcal{F} \), \( v \) is reachable from \( v(S) \) in \( \mathcal{H}(\mathcal{F},D) \) if, and only if, \( v = v(S') \) for some \( S' \in \mathcal{F} \) such that \( S' \subseteq S \), or \( v = v(x) \) for some \( x \in S \).

Proof. Clearly, any vertex \( v[x] \) is reachable from \( v[S] \) through the hyperarc \( a[S] \). Besides, assuming \( S \supseteq S' \), then \( v[S] \) reaches \( v[S'] \) through the hyperpath formed by the hyperarcs \( a[S] \) and \( a'[S'] \).

Now, let us prove by induction that these are the only vertices reachable from \( v[S] \). Let \( u \) be reachable from \( v[S] \). If \( u = v[S] \), then this is obvious. Otherwise, there exists a hyperarc \( a = (T,H) \) such that \( u \in H \) and \( T = \{ u_1, \ldots, u_q \} \) with each \( u_i \) being reachable from \( u \). We can distinguish two cases:

(i) either \( a \) is of the form \( a[S'] \) for some \( S' \in \mathcal{F} \), in which case the tail is reduced to the vertex \( v[S'] \), which is reachable from \( v[S] \). By induction, we know that \( S \supseteq S' \). Since \( u = v[x] \) for some \( x \in S' \), it follows that \( x \in S \).

(ii) or \( a \) is of the form \( a'[S'] \) for some \( S' \in \mathcal{F} \). Then its tail is the set of the \( v[x] \) for \( x \in S' \), and its head consists of the single vertex \( v[S'] \). Thus \( x \in S \) for all \( x \in S' \) by induction, which ensures that \( u = v[S'] \) with \( S' \subseteq S \). \( \square \)

Up to adding an extra element to the domain \( D \) and to each set \( S \in \mathcal{F} \), it can be assumed that \( |S| > 1 \) for all \( S \). In this case, the directed hypergraph \( \mathcal{H}(\mathcal{F},D) \) can be shown to be acyclic:

Lemma 8. Let \( \mathcal{F} \) be a family of distinct sets over \( D \). Assuming that \( |S| > 1 \) for all \( S \in \mathcal{F} \), the directed hypergraph \( \mathcal{H}(\mathcal{F},D) \) is acyclic.

Proof. Suppose that \( \mathcal{H}(\mathcal{F},D) \) contains a non-trivial cycle. If this cycle contains two distinct vertices \( v[S] \) and \( v[S'] \), then by Lemma 7 we should have \( S = S' \), which contradicts the distinctness assumption over the sets of \( \mathcal{F} \). Thus, the cycle should contain at least a vertex \( v[x] \) for some \( x \in D \). However, since \( |S| > 1 \) for all \( S \) containing \( x \), \( v[x] \) does not reach any vertices except itself. Therefore, \( v[x] \) cannot belong to any (non-trivial) cycle, which provides a contradiction. \( \square \)

Then the following proposition holds:

Proposition 9. The size of the transitive reduction of the reachability relation of \( \mathcal{H}(\mathcal{F},D) \) is lower bounded by the size of the transitive reduction of the subset partial order over the family \( \mathcal{F} \).

Proof. We are going to show by contrapositive that for any couple \((S,S')\) in the transitive reduction of the subset partial order over the family \( \mathcal{F} \), \((v[S'],v[S])\) must belong to the transitive reduction of the relation \( \sim_{\mathcal{H}(\mathcal{F},D)} \).

Suppose that the pair \((v[S'],v[S])\) is not in transitive reduction of \( \sim_{\mathcal{H}(\mathcal{F},D)} \). If \( S \not\subseteq S' \), then naturally, \((S,S')\) does not belong to the transitive reduction of the subset partial order over \( \mathcal{F} \). Now, let us assume \( S \subseteq S' \). Then there exists a sequence \( u_1, \ldots, u_p \) of \( p \) vertices of \( \mathcal{H}(\mathcal{F},D) \) \((p > 2)\) such that \( u_1 = v[S'] \), \( u_p = v[S] \), and \( u_1 \sim_{\mathcal{H}(\mathcal{F},D)} u_2 \sim_{\mathcal{H}(\mathcal{F},D)} \ldots \sim_{\mathcal{H}(\mathcal{F},D)} u_p \). Observe that any vertex reaching a vertex of the form \( v[T] \) \((T \in \mathcal{F})\) is necessarily of the form \( v[T'] \) for some \( T' \in \mathcal{F} \) (because of the assumption \(|T| > 1\)) which ensures that no vertex of the form \( v[x] \) for \( x \in D \) can reach \( v[T] \). Consequently, there exists \( S_1, \ldots, S_p \in \mathcal{F} \) such that \( u_i = v[S_i] \) for all \( 1 \leq i \leq p \). Following Lemma 7 this shows that \( S_1 \supseteq \cdots \supseteq S_p \). Since \( p > 2 \), \((S,S') = (S_p,S_1)\) cannot belong to the transitive reduction of the subset partial order over \( \mathcal{F} \). \( \square \)
The subset partial order have been well studied in the literature \cite{YJ93,Pri95,Pri99a,Pri99b,Elm09}. It has been proved in \cite{YJ93,Elm09} that the size of the transitive reduction of the subset partial order can be superlinear in the size of the input \((F,D)\) (defined as \(|D|+\sum_{S \in F}|S|\)). Combining this with Prop. 9 provides the following result:

**Theorem 10.** There is a directed hypergraph \(H\) such that the size of the transitive reduction of the reachability relation is in \(\Omega(\text{size}(H)^2/\log^2(\text{size}(H)))\).

**Proof.** We use the construction given in \cite{Elm09} in which \(F\) consists of two disjoint families \(F_1\) and \(F_2\) of sets over the domain \(D = \{x_1, \ldots, x_n\}\) (where \(n\) is supposed to be divisible by 4). The first family is formed by the subsets containing all the elements \(x_1, \ldots, x_{n/2}\), and precisely \(n/4\) elements among \(x_{n/2+1}, \ldots, x_n\). The second family consists of the subsets having \(n/4\) elements among \(x_1, \ldots, x_{n/2}\).

Clearly, the transitive reduction of the subset partial order over \(F\) coincides with the cartesian product \(F_2 \times F_1\). Each \(F_i\) precisely contains \(\binom{n/2}{n/4}\) = \(\Theta(2^{n/2}/\sqrt{n})\) sets, so that the size of the transitive reduction of the subset partial order is \(\Theta(2^{n/2}/\sqrt{n})\).

Proposition 9 shows that the size of the transitive reduction of the reachability relation \(\rightarrow_{H(F,D)}\) is in \(\Omega(2^{n/2}/n)\). Now, the size of the directed hypergraph \(H(F,D)\) is equal to:

\[
\text{size}(H(F,D)) = n + 2\left(\frac{n/2}{n/4}\right) + 2\frac{n}{4}\left(\frac{n/2}{n/4}\right) + 2\frac{n}{4}\left(\frac{n/2}{n/4}\right),
\]

so that \(\text{size}(H(F,D)) = \Theta(\sqrt{n}2^{n/2})\). This provides the expected result. \(\square\)

Theorem 10 highlights an important difference between directed graphs and hypergraphs. Unlike graphs, hypergraphs do not admit any economical representation of the reachability relation having a size in \(O(\text{size}(H))\). As a consequence, the reachability relation embedded in directed hypergraphs is combinatorially more complex than in directed graphs.

### 4.2. Reachability problems in directed hypergraphs and combinatorial problems on sets.

The lower bound provided by Theorem 10 suggests that solving some reachability related problems in directed hypergraphs may be not as easy as in digraphs. This is confirmed by the results of this section, in which we exhibit linear time reductions of problems on the subset partial order to such reachability problems.

Topological sort and linear extension. The topological sort of an acyclic directed hypergraph \(H\) refers to a total ordering \(\preceq\) of the vertices such that \(u \preceq v\) as soon as \(u \rightarrow_H v\). Using the hypergraphs \(H(F,D)\) built from families of sets introduced in Section 4.1, we can establish the following result:

**Proposition 11.** There is a linear time reduction from the problem of determining a linear extension of the subset partial order over a family of sets, to the problem of topologically sorting the vertices of an acyclic directed hypergraph.

**Proposition 11.** Consider a family \(F\) of sets over a domain \(D\). The directed hypergraph \(H(F,D)\) can be built in linear time in the size of \((F,D)\) (i.e. \(|D|+\sum_{S \in F}|S|\)). Suppose that we now have a topological ordering \(\preceq\) over the vertices. Without loss of generality, it can be supposed that it is given by a real-valued function \(f\) such that \(u \preceq v\) if, and only if, \(f(u) \leq f(v)\). By Lemma 7, for any two sets \(S, S' \in F\) such
that $S' \subseteq S$, we have $f(v[S]) \leq f(v[S'])$. It follows that the function $g : \mathcal{F} \to \mathbb{R}$ defined by $g(S) = -f(v[S])$ yields a linear extension of the partial order over $\mathcal{F}$. \qed

To our knowledge, the problem of determining a linear extension of the subset partial order over a family $\mathcal{F}$ of sets has not been particularly studied. It is probably not obvious to solve this problem without examining a significant part of the subset partial order (or at least of a sparse representation such as its transitive reduction). The best known methods to compute the subset partial order have a complexity in $O(N^{2}/\log^{2}N)$ in the dense case [Elm09], and $O(N^{2}/\log N)$ in general (e.g. [Pri95]), where $N$ is the size of the input. In comparison, topologically sorting directed graphs can be solved in linear time.

Strongly connected components and the minimal set problem. Given a family $\mathcal{F}$ of distinct sets as above, the minimal set problem consists in finding the minimal sets $S \in \mathcal{F}$ for the subset partial order $\subseteq$. This problem has received much attention [Pri91], [Y92], [Y93], [Pri95], [Pri99], [Elm99], [BP11]. It has important applications in propositional logic [Pri91] or data mining [BP11]. It can also be seen as a boolean case of the problem of finding maximal vectors among a given family [KLP75], [KS85], [GSG05].

We establish a linear time reduction from the minimal set problem to the problem of determining the strongly connected components in a directed hypergraph. Given a family $\mathcal{F}$ of sets over the domain $D$, we build a directed hypergraph $\overline{\mathcal{H}}(\mathcal{F}, D)$ starting from the hypergraph $\mathcal{H}(\mathcal{F}, D)$. On top of the vertices of the latter, $\overline{\mathcal{H}}(\mathcal{F}, D)$ has the following vertices: (i) for each $S \in \mathcal{F}$, an additional vertex $w[S]$, (ii) $(|D| + 1)$ vertices labelled by $c_{0}, \ldots, c_{|D|}$, (iii) and a special vertex labelled by $\text{superset}$. Besides, we add the following hyperarcs: (i) for each $S \in \mathcal{F}$, a hyperarc leaving $\{v[S]\}$ and entering the singleton $\{c_{|S| - 1}\}$, (ii) for every $0 \leq i \leq |D|$, a hyperarc leaving $\{c_{i}\}$ and entering the set of the vertices $w[S]$ such that $i = |S|$, (iii) for each $i > 0$, a hyperarc from $\{c_{i}\}$ to $\{c_{i-1}\}$, (iv) for each $S \in \mathcal{F}$, a hyperarc leaving the set $\{v[S], w[S]\}$ and entering the singleton $\{\text{superset}\}$, (v) for every $S \in \mathcal{F}$, a hyperarc from $\{\text{superset}\}$ to $\{v[S]\}$. This construction is illustrated in Figure 5.

Every vertex $v[S]$ is reachable from $\text{superset}$. Conversely, it can be shown that $v[S]$ reaches $\text{superset}$ if, and only if, it is not minimal, meaning that there exists $S' \in \mathcal{F}$ such that $S \supseteq S'$.

\textbf{Proposition 12.} For any $S \in \mathcal{F}$, $S$ is not minimal in $\mathcal{F}$ if, and only if, $\text{superset}$ is reachable from $v[S]$ in $\overline{\mathcal{H}}(\mathcal{F}, D)$.

\textit{Proof.} Assume that $S$ is not minimal in $\mathcal{F}$, and let $S' \in \mathcal{F}$ satisfying $S' \supseteq S$. Then by Lemma 3, $v[S']$ is reachable from $v[S]$ in $\mathcal{H}(\mathcal{F}, D)$, and hence in $\overline{\mathcal{H}}(\mathcal{F}, D)$. Besides, since $|S'| = j < |S| = i$, then $w[S']$ is reachable from $v[S]$ through the hyperarc traversing the vertices $c_{j}, c_{j+1}, \ldots, c_{i}$. Finally, the vertex $\text{superset}$ is reachable through the hyperarc from $\{v[S'], w[S']\}$.

Conversely, suppose that $v[S]$ reaches $\text{superset}$ in $\overline{\mathcal{H}}(\mathcal{F}, D)$. Consider a minimal hyperpath $a_{1}, \ldots, a_{p}$ from $v[S]$ to $\text{superset}$. Necessarily, $a_{p}$ is a hyperarc of the form $(\{v[S'], w[S']\}, \{\text{superset}\})$ for some $S' \in \mathcal{F}$. Consequently, both vertices $v[S']$ and $w[S']$ are reachable from $v[S]$. Besides, to each of the two vertices, there exists a hyperarc from $v[S]$ which does not contain the vertex $\text{superset}$ (meaning that the latter does not appear in any tail or head of the hyperarcs of the hyperpath). These two hyperarcs are subsequences of $a_{1}, \ldots, a_{p}$.
Figure 5. The hypergraph $H(F, D)$, where $D = \{x_1, \ldots, x_4\}$ and $F$ consists of $S_1 = \{x_1, x_2, x_4\}$, $S_2 = \{x_1, x_2, x_3\}$, and $S_3 = \{x_1, x_2\}$. The hyperarcs of $H(F, D)$ are depicted in gray.

Thus, suppose that $a'_1, \ldots, a'_q$ is a minimal hyperpath from $v[S]$ to $v[S']$ not containing \textit{superset}. In this case, no vertex of the form $w[T]$ for $T \in F$ appears in the hyperpath, since otherwise, the vertex \textit{superset} should also appear (the only hyperarc from $w[T]$ enters \textit{superset}), or the hyperpath would not be minimal (we could remove the hyperarc leading to $\{w[T]\}$). Similarly, no vertex of the form $c_i$ belongs to the hyperpath, since otherwise, it should also contain a vertex of the form $w[T]$ (or the hyperpath would not be minimal). It follows that the hyperpath $a'_1, \ldots, a'_q$ is also a hyperpath in the hypergraph $H(F, D)$. Applying Lemma 7 then shows that $S' \subseteq S$.

It remains to show that the latter inclusion is strict. Similarly, let $a''_1, \ldots, a''_r$ be a minimal hyperpath from $v[S]$ to $w[S']$ not containing \textit{superset}. Then the tail of $a''_i$ is necessarily reduced to the vertex $c_i$, where $i = |S'|$, and its head is $\{w[S']\}$. It follows that $a''_1, \ldots, a''_{r-1}$ is a hyperpath from $v[S]$ to $c_i$ not containing \textit{superset}. Now suppose that $i \geq |S|$. Let $j \geq i$ the greatest integer such that $c_j$ appears in the hyperpath $a''_1, \ldots, a''_{r-1}$. Necessarily, one of the hyperarc in the hyperpath is of the form $(\{v[T]\}, \{c_j\})$, so that $v[T]$ is reachable from $v[S]$ through a hyperpath not passing through the vertex \textit{superset}. It follows from the previous discussion that $T \subseteq S$. But $|T| = j + 1 > i$, which is a contradiction. This shows that $i = |S'| < |S|$, hence $S' \subset S$.  

As a consequence, minimal sets of the family $F$ are precisely given by the vertices of the form $v[S]$ which do not belong to the SCC of the vertex \textit{superset}. This proves the following complexity reduction:
Theorem 13. The minimal set problem can be reduced in linear time to the problem of determining the SCCs in a directed hypergraph.

Proof. We assume the existence of an oracle providing the SCCs of any directed hypergraph.

Consider an instance $(F, D)$ of the minimal set problem. The hypergraph $H(F, D)$ can be built in linear time in the size of the input. Calling the oracle on $H(F, D)$ yields its SCCs. Then, by examining each SCC and its content, we collect the $S \in F$ such that $v[S]$ does not belong to the same component as the vertex superset. We finally return these sets. By Proposition 12 they are precisely the minimal sets in the family $F$. □

No algorithm is known to solve the minimal set problem in linear time. Surprisingly, the most efficient algorithms addressing the problem compute the whole subset partial order [YJ93, Elm09], so that the best known time complexity is in $O(N^2/\log^k N)$ ($k = 1$ or 2).

Remark 3. Another interesting combinatorial problem is to decide whether a collection of sets is a Sperner family, i.e. the sets are not pairwise comparable. As a consequence of Theorem 13, it can be shown that the problem of deciding whether a collection of sets is a Sperner family can be reduced in linear time to the problem of determining the SCCs in a directed hypergraph. The Sperner family problem can be indeed reduced in linear time to the minimal set problem, by examining whether the number of minimal sets of $F$ is equal to the cardinality of $F$.

5. Conclusion

In this paper, we have studied several aspects relative to reachability and strongly connected components in directed hypergraphs. We have defined an algorithm which allows to determine all terminal SCCs in almost linear time. In comparison, the previous approaches run in quadratic time. As a consequence, two other important problems, testing strong connectivity and the existence of a sink, can be solved in almost linear time.

We have also shown that the reachability relation in directed hypergraphs is more complex than in directed graphs, by proving a superlinear lower bound on the size of its transitive reduction (Th. 10). We have defined linear time reductions from combinatorial problems on set families to reachability problems in directed hypergraphs, in particular from the minimal set problem to the problem of determining the SCCs of a directed hypergraph (Th. 13). This strongly suggests that the latter may be not solvable in linear time as in directed graphs. These reductions also strengthen the interest for finding efficient algorithms to determine all SCCs in directed hypergraphs.

For future work, we consequently plan to study how to generalize the algorithm introduced in Section 3 to find all SCCs, hopefully improving the existing complexity bounds on the minimal set problem. In parallel, it would be interesting to study complexity lower bounds (most likely superlinear ones) on the problem of computing the strongly connected components. We think that the reduction from combinatorial problems on sets could be helpful to derive such bounds.
References

[ADS83] Giorgio Ausiello, Alessandro D’Atri, and Domenico Saccà. Graph algorithms for functional dependency manipulation. *J. ACM*, 30:752–766, October 1983.

[ADS86] G Ausiello, A D’Atri, and D Saccá. Minimal representation of directed hypergraphs. *SIAM J. Comput.*, 15:418–431, May 1986.

[AFF01] Giorgio Ausiello, Paolo Giulio Franciosa, and Daniele Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Proceedings*, volume 2202 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2001.

[AFFG97] Giorgio Ausiello, Paolo Giulio Franciosa, Daniele Frigioni, and Roberto Giaccio. Decremental maintenance of reachability in hypergraphs and minimum models of horn formulae. In Hon Wai Leong, Hiroshi Imai, and Sanjay Jain, editors, *Algorithms and Computation, 8th International Symposium, ISAAC ’97, Singapore, December 17-19, 1997, Proceedings*, volume 1350 of *Lecture Notes in Computer Science*, pages 122–131. Springer, 1997.

[AGG10] Xavier Allamigeon, Stéphane Gaubert, and Éric Goubault. The tropical double description method. In J.-Y. Marion and Th. Schwentick, editors, *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS 2010)*, volume 5 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47–58, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[AGU72] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[A91] Giorgio Ausiello and Giuseppe F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *J. Log. Program.*, 10(1/2/3&4):69–90, 1991.

[All09a] Xavier Allamigeon. Static analysis of memory manipulations by abstract interpretation — Algorithmics of tropical polyhedra, and application to abstract interpretation. PhD thesis, École Polytechnique, Palaiseau, France, November 2009.

[All09b] Xavier Allamigeon. TPLib: Tropical polyhedra library, 2009. Distributed under LGPL, available at [https://gforge.inria.fr/projects/tplib](https://gforge.inria.fr/projects/tplib).

[ANI90] Giorgio Ausiello, Umberto Nanni, and Giuseppe F. Italiano. Dynamic maintenance of directed hypergraphs. *Theoretical Computer Science*, 72(2-3):97 – 117, 1990.

[BP11] Roberto J. Bayardo and Biswanath Panda. Fast algorithms for finding extremal sets. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2011*. SIAM, 2011. To appear.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[Elm09] Amr Elmasry. Computing the subset partial order for dense families of sets. *Information Processing Letters*, 109(18):1082 – 1086, 2009.

[GGPR98] Giorgio Gallo, Claudio Gentile, Daniele Pretolani, and Gabriella Rago. Max horn sat and the minimum cut problem in directed hypergraphs. *Math. Program.*, 80:213–237, 1998.

[GLPN93] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Appl. Math.*, 42(2-3):177–201, 1993.

[GP95] Giorgio Gallo and Daniele Pretolani. A new algorithm for the propositional satisfiability problem. *Discrete Applied Mathematics*, 60(1-3):159–179, 1995.

[GSG05] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31st international conference on Very large data bases*, VLDB ’05, pages 229–240. VLDB Endowment, 2005.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22:469–476, October 1975.

[KS85] David G. Kirkpatrick and Raimund Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Proceedings of the first annual symposium on Computational geometry*, SCG ’85, pages 89–96, New York, NY, USA, 1985. ACM.

[LS98] Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In Kim Gulstrand Larsen, Sven Skyum, and Glynn
Appendix A. An Example of Complete Execution Trace of the Algorithm of Section 3

We give the main steps of the execution of the Algorithm TERMINAL_SCC on the directed hypergraph depicted in Figure 1.
Vertices are depicted by solid circles if their index is defined, and by dashed circles otherwise. Once a vertex is placed into $\text{Finished}$, it is depicted in gray. Similarly, a hyperarc which has never been placed into a local stack $F$ is represented by dotted lines. Once it is pushed into $F$, it becomes solid, and when it is popped from $F$, it is colored in gray (note that for the sake of readability, gray hyperarcs mapped to cycles after a vertex merging step will be removed). The stack $F$ which is mentioned always corresponds to the stack local to the last non-terminated call of the function $\text{Visit}$.

Initially, $\text{Find}(z) = z$ for all $z \in \{u,v,w,x,y,t\}$. We suppose that $\text{Visit}(u)$ is called first. After the execution of the block from Lines 18 to 35 the current state is:

Upon visiting $u$, the root of the hyperarc $a_1$ is set to $u$. Then, $a_2$ is pushed on the stack $F$.

Similarly, the function $\text{Visit}(w)$ is called during the execution of the loop from Lines 36 to 48 in $\text{Visit}(v)$. After Line 45 in $\text{Visit}(w)$, the root of the hyperarc $a_5$ is set to $w$, and the counter $c_{a_5}$ is incremented to 1 since $w \in S$. Besides, $c_{a_4}$ is incremented to 2 since $\text{Find}(r_{a_4}) = \text{Find}(v) = v \in S$, so that $a_4$ is pushed on the stack $F_v$. The state is:
The execution of the loop from Lines 36 to 48 of VISIT(w) discovers that index[u] is defined but $u \notin \text{Finished}$, so that low[w] is set to min(low[w], low[u]) = 0 and is_term[w] to is_term[w] && is_term[u] = true. At the end of the loop, the state is therefore:

Since low[w] $\neq$ index[w], the block from Lines 49 to 56 is not executed, and VISIT(w) terminates. Back to the loop from Lines 36 to 48 in VISIT(v), low[v] is assigned to the value min(low[v], low[w]) = 0, and is_term[v] to is_term[v] && is_term[w] = true:

Since low[v] $\neq$ index[v], the block from Lines 49 to 56 is not executed, and VISIT(v) terminates. Back to the loop from Lines 36 to 48 in VISIT(u), low[u] is assigned to the value min(low[u], low[w]) = 0, and is_term[u] to is_term[w] && is_term[w] = true. Therefore, at Line 49 the conditions low[u] = index[u] and is_term[u] = true hold,
so that a vertex merging step is executed. At that point, the stack \( F \) is empty. After that, \( i \) is set to \( index[u] = 0 \) (Line 54), and \( F_u = [] \) is emptied to \( F \) (Line 52), so that \( F \) is still empty. Then \( w \) is popped from \( S \), and since \( index[w] = 2 > i = 0 \), the loop from Lines 54 to 58 is iterated. Then the stack \( F_w = [] \) is emptied in \( F \).

At Line 56, \( \text{Merge}(u, w) \) is called. The result is denoted by \( U \) (in practice, either \( U = u \) or \( U = w \)). The state is:

\[
\begin{align*}
\text{index}[v] &= 1 \\
\text{low}[v] &= 0 \\
\text{is_term}[v] &= \text{true}
\end{align*}
\]

Then \( v \) is popped from \( S \), and since \( index[v] = 1 > i = 0 \), the loop Lines 54 to 58 is iterated again. Then the stack \( F_v = [a_4] \) is emptied in \( F \). At Line 56, \( \text{MERGE}(U, v) \) is called. The result is set to \( U \) (in practice, \( U \) is one of the vertices \( u, v, w \)). The state is:

\[
\begin{align*}
\text{index}[U] &= 0, 1, \text{ or 2} \\
\text{low}[U] &= 0 \\
\text{is_term}[U] &= \text{true}
\end{align*}
\]

After that, \( u \) is popped from \( S \), and as \( index[u] = 0 = i \), the loop is terminated. At Line 59, \( index[U] \) is set to \( i \), and \( U \) is pushed on \( S \). Since \( F \neq \emptyset \), we go back to Line 56 in the state:

\[
\begin{align*}
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{is_term}[U] &= \text{true}
\end{align*}
\]

Then \( a_4 \) is popped from \( F \), and the loop from Lines 48 to 47 iterates over \( H(a_4) = \{x, y\} \). Suppose that \( x \) is treated first. Then \( \text{VISIT}(x) \) is called. During its execution, at Line 55, the state is:
Since $F$ is empty, the loop from Lines 46 to 58 is not executed. At Line 49, $\text{low}[x] = \text{index}[x]$ and $\text{is_term}[x] = \text{true}$, so that a trivial vertex merging step is performed, only on $x$, since it is the top element of $S$. At Line 50, it can be verified that $S = [x; U]$, $\text{index}[x] = 3$ and $F = \emptyset$. Therefore, the goto statement at Line 59 is not executed. It follows that the loop from Lines 58 to 60 is executed, and after that, the state is:

\[
\begin{align*}
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{is_term}[x] &= \text{true}
\end{align*}
\]

\[
\begin{align*}
S &= [x; U] \\
n &= 4 \\
F &= \emptyset \\
U &= \text{Find}(u) = \text{Find}(v) \\
&= \text{Find}(w)
\end{align*}
\]

\[
\text{Finished} = \{x\}
\]

After the termination of \text{Visit}(x), since $x \in \text{Finished}$, $\text{is_term}[U]$ is set to $false$. After that, \text{Visit}(y) is called, and at Line 56, it can be checked that $c_{a_5}$ has been incremented to $2 = |T(a_5)|$ because $R_{a_5} = \text{Find}(r_{a_5}) = \text{Find}(w) = U$ and $U \in S$. Therefore, $a_5$ is pushed to $F_U$, and the state is:

\[
\begin{align*}
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{is_term}[x] &= \text{true}
\end{align*}
\]

\[
\begin{align*}
S &= [U] \\
n &= 5 \\
F &= \emptyset \\
U &= \text{Find}(u) = \text{Find}(v) \\
&= \text{Find}(w)
\end{align*}
\]

\[
\text{Finished} = \{x\}
\]
As for the vertex $x$, $\text{Visit}(y)$ terminates by popping $y$ from $S$ and adding it to $\text{Finished}$. Back to the execution of $\text{Visit}(U)$, at Line 49, the state is:

$$
\begin{align*}
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{is\_term}[x] &= \text{true} \\
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{is\_term}[U] &= \text{false}
\end{align*}
$$

While $\text{low}[U] = \text{index}[U]$, $\text{is\_term}[U]$ is equal to $\text{false}$, so that no vertex merging loop is performed on $U$. Therefore, $a_5$ is not popped from $F_U$. Nevertheless, the loop from Lines 62 to 64 is executed, and after that, $\text{Visit}(u)$ is terminated in the state:

$$
\begin{align*}
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{is\_term}[x] &= \text{true} \\
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{is\_term}[U] &= \text{false}
\end{align*}
$$

Finally, $\text{Visit}(t)$ is called from $\text{TerminalScc}$ at Line 13. It can be verified that a trivial vertex merging loop is performed on $t$ only. After that, $t$ is placed into $\text{Finished}$. Therefore, the final state of $\text{TerminalScc}$ is:

$$
\begin{align*}
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{is\_term}[x] &= \text{true} \\
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{is\_term}[U] &= \text{false}
\end{align*}
$$
As \( \text{is\_term}[x] = \text{is\_term}[y] = \text{is\_term}[t] = \text{true} \) and \( \text{is\_term}[\text{Find}(z)] = \text{false} \) for \( z = u, v, w \), there are three terminal SCCs, given by the sets:

\[
\{ z \mid \text{Find}(z) = x \} = \{ x \},
\{ z \mid \text{Find}(z) = y \} = \{ y \},
\{ z \mid \text{Find}(z) = t \} = \{ t \}.
\]

**Appendix B. Proof of Theorem 4**

The correctness proof of the algorithm TERMINALSCC turns out to be harder than for algorithms on directed graphs such as Tarjan’s one [Tar72], due to the complexity of the invariants which arise in the former algorithm. That is why we propose to show the correctness of two intermediary algorithms, named TERMINALSCC2 (Figure 6) and TERMINALSCC3 (Figure 7), and then to prove that they are equivalent to TERMINALSCC.

The main difference between the first intermediary form and TERMINALSCC is that it does not use auxiliary data associated to the hyperarcs to determine which ones are added to the digraph \( \text{graph}(\mathcal{H}_{\text{cur}}) \) after a vertex merging step. Instead, the stack \( F \) is directly filled with the right hyperarcs (Lines 22 and 49). Besides, a boolean \( \text{no\_merge} \) is used to determine whether a vertex merging step has been executed. The notion of vertex merging step is refined: it now refers to the execution of the instructions between Lines 41 and 50 in which the boolean \( \text{no\_merge} \) is set to \( \text{false} \).

For the sake of simplicity, we will suppose that sequences of assignment or stack manipulations are executed atomically. For instance, the sequences of instructions located in the blocks from Lines 16 and 25, or from Lines 41 and 50, and at from Lines 56 to 58 are considered as elementary instructions. Under this assumption, intermediate complex invariants do not have to be considered.

We first begin with very simple invariants:

**Invariant 1.** Let \( U \) be a vertex of the current hypergraph \( \mathcal{H}_{\text{cur}} \). Then \( \text{index}[U] \) is defined if, and only if, \( \text{index}[u] \) is defined for all \( u \in \mathcal{V} \) such that \( \text{Find}(u) = U \).

**Proof.** It can be shown by induction on the number of vertex merging steps which has been performed on \( U \).
Invariant 2. Let \( u \in \mathcal{V} \). When \( \text{index}[u] \) is defined, then \( \text{Find}(u) \) belongs either to the stack \( S \), or to the set \( \text{Finished} \) (both cases cannot happen simultaneously).

Proof. Initially, \( \text{Find}(u) = u \), and once \( \text{index}[u] \) is defined, \( \text{Find}(u) \) is pushed on \( S \) (Line 20). Naturally, \( u \not\in \text{Finished} \), because otherwise, \( \text{index}[u] \) would have been defined before (see the condition Line 58). After that, \( U = \text{Find}(u) \) can be popped from \( S \) at three possible locations:

- at Lines 14 or 46 in which case \( U \) is transformed into a vertex \( U' \) which is immediately pushed on the stack \( S \) at Line 48. Since after that, \( \text{Find}(u) = U' \), the property \( \text{Find}(u) \in S \) still holds.
- at Line 50 in which case it is directly appended to the set \( \text{Finished} \).

\[
\begin{align*}
\text{Invariant 3. The set } \text{Finished is always growing.}
\end{align*}
\]
Proof. Once an element is added to Finished, it is never removed from it nor merged into another vertex (the function MERGE is always called on elements immediately popped from the stack $S$).

**Proposition 14.** After the algorithm $\text{TerminalScc2}(\mathcal{H})$ terminates, the sets \( \{ v \in \mathcal{V} \mid \text{Find}(v) = U \text{ and } \text{is\_term}[U] = \text{true} \} \) are precisely the terminal SCCs of $\mathcal{H}$.

Proof. We prove the whole statement by induction on the number of vertex merging steps.

**Basis Case.** First, suppose that the hypergraph $\mathcal{H}$ is such that no vertices are merged during the execution of $\text{TerminalScc2}(\mathcal{H})$, i.e. the vertex merging loop (from Lines 43 to 47) is never executed. Then the boolean $\text{no\_merge}$ is always set to $\text{true}$, so that $n$ is never redefined to $i + 1$ (Line 52), and there is no back edge to Line 26 in the control-flow graph. It follows that removing all the lines between Lines 43 to 53 does not change the behavior of the algorithm. Besides, since the function MERGE is never called, $\text{Find}(u)$ always coincides with $u$. Finally, at Line 27, $F$ is precisely assigned to the set of simple hyperarcs leaving $u \in \mathcal{H}$, so that the loop from Lines 26 to 38 iterates on the successors of $u$ in $\text{graph}(\mathcal{H})$. As a consequence, the algorithm $\text{TerminalScc2}(\mathcal{H})$ behaves exactly like $\text{TerminalScc}(\text{graph}(\mathcal{H}))$. Moreover, under our assumption, the terminal SCCs of $\text{graph}(\mathcal{H})$ are all reduced to singletons (otherwise, the loop from Lines 43 to 47 would be executed, and some vertices would be merged). Therefore, by Proposition 11, the statement in Proposition 14 holds.

**Inductive Case.** Suppose that the vertex merging loop is executed at least once, and that its first execution happens during the execution of, say, $\text{Visit2}(x)$. Consider the state of the algorithm at Line 41 just before the execution of the first occurrence of the vertex merging step. Until that point, $\text{Find}(v)$ is still equal to $v$ for all vertices $v \in \mathcal{V}$, so that the execution of $\text{TerminalScc2}(\mathcal{H})$ coincides with the execution of $\text{TerminalScc}(\text{graph}(\mathcal{H}))$. Consequently, if $C$ is the set formed by the vertices $y$ located above $x$ in the stack $S$ (including $x$), $C$ forms a terminal SCC of $\text{graph}(\mathcal{H})$. In particular, the elements of $C$ are located in a same SCC of the hypergraph $\mathcal{H}$.

Consider the hypergraph $\mathcal{H}'$ obtained by merging the elements of $C$ in the hypergraph $(\mathcal{V}, \mathcal{A} \setminus \{ a \mid \exists y \in C \text{ s.t. } T(a) = \{ y \} \})$, and let $X$ be the resulting vertex. For now, we may add a hypergraph as last argument of the functions $\text{Visit2}$, $\text{Find}$, \ldots to distinguish their execution in the context of the call to $\text{TerminalScc2}(\mathcal{H})$ or $\text{TerminalScc2}(\mathcal{H}')$. We make the following observations:

- the vertex $x$ is the first element of the component $C$ to be visited during the execution of $\text{TerminalScc2}(\mathcal{H})$. It follows that the execution of $\text{TerminalScc2}(\mathcal{H})$ until the call to $\text{Visit2}(x, \mathcal{H})$ coincides with the execution of $\text{TerminalScc2}(\mathcal{H}')$ until the call to $\text{Visit2}(X, \mathcal{H}')$.
- besides, during the execution of $\text{Visit2}(x, \mathcal{H})$, the execution of the loop from Lines 26 to 38 only has a local impact, i.e. on the is\_term[$y$], index[$y$], or low[$y$] for $y \in C$, and not on any information relative to other vertices. Indeed, we claim that the set of the vertices $y$ on which $\text{Visit2}$ is called during the execution of the loop is exactly $C \setminus \{ x \}$. First, for all $y \in C \setminus \{ x \}$, $\text{Visit2}(y)$ has necessarily been executed after Line 26 (otherwise, by Invariant 2, $y$ would be either below $x$ in the stack $S$, or in $\text{Finished}$). Conversely, suppose that after Line 26, there is a call to $\text{Visit2}(t)$ with $t \notin C$. By Invariant 2, $t$ belongs to $\text{Finished}$, so that for one of
the vertices \( w \) examined in the loop, either \( w \in \text{Finished} \) or \( \text{is\_term}[w] = \text{false} \) after the call to \( \text{Visit2}(w) \). Hence \( \text{is\_term}[x] \) should be \( \text{false} \), which contradicts our assumptions.

- finally, from the execution of Line 63 during the call to \( \text{Visit2}(x, \mathcal{H}) \), our algorithm behaves exactly as \( \text{TerminalScc2}(\mathcal{H}') \) from the execution of Line 63 in \( \text{Visit2}(X, \mathcal{H}') \). Indeed, \( \text{index}[X] \) is equal to \( i \), and the latter is equal to \( n - 1 \). Similarly, for all \( y \in C \), \( \text{low}[y] = i \) and \( \text{is\_term}[y] = \text{true} \). The vertex \( X \) being equal to one of the \( y \in C \), we also have \( \text{low}[X] = i \) and \( \text{is\_term}[X] = \text{true} \). Moreover, \( X \) is the top element of \( S \).

Furthermore, it can be verified that at Line 69, the set \( F \) contains exactly all the hyperarcs of \( A \) which generate the simple hyperarcs leaving \( X \) in \( \mathcal{H}' \): they are exactly characterized by

\[
\text{Find}(z, \mathcal{H}) = X \text{ for all } z \in T(a), \text{ and } T(a) \neq \{y\} \text{ for all } y \in C \\
\iff \text{Find}(z, \mathcal{H}) = X \text{ for all } z \in T(a), \text{ and } \text{collected}_a = \text{false}
\]

since at that Line 69 a hyperarc \( a \) satisfies \( \text{collected}_a = \text{true} \) if, and only if, \( T(a) \) is reduced to a singleton \( \{l\} \) such that \( \text{index}[l] \) is defined.

Finally, for all vertices \( y \in C \), \( \text{Find}(y, \mathcal{H}) \) can be equivalently replaced by \( \text{Find}(X, \mathcal{H}') \).

As a consequence, \( \text{TerminalScc2}(\mathcal{H}) \) and \( \text{TerminalScc2}(\mathcal{H}') \) return the same result. Both functions perform the same union-find operations, except the first the vertex merging step executed by \( \text{TerminalScc2}(\mathcal{H}) \) on \( C \).

Let \( f \) be the function which maps all vertices \( y \in C \) to \( X \), and any other vertex to itself. We claim that \( \mathcal{H}' \) and \( f(\mathcal{H}) \) have the same reachability graph, i.e. \( \sim_{\mathcal{H}'} \) and \( \sim_{f(\mathcal{H})} \) are identical relations. Indeed, the two hypergraphs only differ on the images of the hyperarcs \( a \in A \) such that \( T(a) = \{y\} \) for some \( y \in C \). For such hyperarcs, we have \( H(a) \subseteq C \), because otherwise, \( \text{is\_term}[x] \) would have been set to \( \text{false} \) (i.e. the \( \text{Scc} \ C \) would not be terminal). It follows that their are mapped to the cycle \( \{(X), (X)\} \) by \( f \), so that \( \mathcal{H}' \) and \( f(\mathcal{H}) \) clearly have the same reachability graph. In particular, they have the same terminal \( \text{Sccs} \).

Finally, since the elements of \( C \) are in a same \( \text{Scc} \) of \( \mathcal{H} \), Proposition 8 shows that the function \( f \) induces a one-to-one correspondence between the \( \text{Sccs} \) of \( \mathcal{H} \) and the \( \text{Sccs} \) of \( f(\mathcal{H}) \):

\[
D \mapsto f(D) \\
(D' \setminus \{X\}) \cup C \leftarrow D' \quad \text{if } X \in D' \\
D' \leftarrow D' \quad \text{otherwise.}
\]

The action of the function \( f \) exactly corresponds to the vertex merging step performed on \( C \). Since by induction hypothesis, \( \text{TerminalScc2}(\mathcal{H}') \) determines the terminal \( \text{Sccs} \) in \( f(\mathcal{H}) \), it follows that Proposition 14 holds. \( \square \)

The second intermediary version of our algorithm, \( \text{TerminalScc3} \), is based on the first one, but it performs the same computations on the auxiliary data \( r_a \) and \( c_a \) as in \( \text{TerminalScc} \). However, the latter are never used, because at Line 62 \( F \) is re-assigned to the value provided in \( \text{TerminalScc2} \). It follows that for now, the parts in gray can be ignored. The following lemma states that \( \text{TerminalScc2} \) and \( \text{TerminalScc3} \) are equivalent:
Figure 7. Second intermediary form of our linear algorithm on hypergraphs

Proposition 15. Let $H$ be a directed hypergraph. After the execution of the algorithm TERMINALSOC3($H$), the sets \{$v \in V \mid \text{Find}(v) = U$ and $\text{is}_\text{term}[U] = true$\} precisely correspond to the terminal SCCs of $H$.

Proof. When VISIT3($u$) is executed, the local stack $F$ is not directly assigned to the set \{$a \in A \mid T(a) = \{u\}$\} (see Line 22 in Figure 6), but built by several iterations on the set $A_u$ (Line 23). Since $u \in T(a)$ and $|T(a)| = 1$ holds if, and only if, $T(a)$ is reduced to \{$u$\}, VISIT3($u$) initially fills $F$ with the same hyperarcs as VISIT2($u$).

Besides, the condition no_merge = false in VISIT2 (Line 53) is replaced by $F \neq \emptyset$ (Line 64). We claim that the condition $F \neq \emptyset$ can be safely used in VISIT2 as well. Indeed, in VISIT2, $F \neq \emptyset$ implies no_merge = false. Conversely, suppose that in VISIT2, no_merge = false and $F = \emptyset$, so that the algorithm goes back to Line 53 after having no_merge to true. The loop from Lines 56 to 58 is not executed since $F = \emptyset$, and it directly leads to a new execution of Lines 59 to 63 with no_merge = true. Therefore, going back to Line 53 was useless.

Finally, during the vertex merging step in VISIT3, $n$ keeps its value, which is greater than or equal to $i + 1$, but is not necessarily equal to $i + 1$ like in VISIT2 (just after Line 52). This is safe because the whole algorithm only need that $n$ take increasing values, and not necessarily consecutive ones.

We conclude by applying Proposition 14.
We make similar assumptions on the atomicity of the sequences of instructions. Note that Invariant 1, 2, and 3 still holds in VISIT3.

\textbf{Invariant 4.} Let \( a \in A \) such that \( |T(a)| > 1 \). If for all \( x \in T(a) \), \( \text{index}[x] \) is defined, then the root \( r_a \) is defined.

\textit{Proof.} For all \( x \in T(a) \), VISIT3\((x)\) has been called. The root \( r_a \) has necessarily been defined at the first of these calls (remember that the block from Lines 15 to 17 is supposed to be executed atomically). \( \square \)

\textbf{Invariant 5.} Consider a state \( \text{cur} \) of the algorithm in which \( U \in \text{Finished} \). Then any vertex reachable from \( U \) in \( \text{graph}(H_{\text{cur}}) \) is also in \( \text{Finished} \).

\textit{Proof.} The invariant clearly holds when \( U \) is placed in \( \text{Finished} \). Using the atomicity assumptions, the call to VISIT3\((u)\) is necessarily terminated. Let \( \text{old} \) be the state of the algorithm at that point, and \( H_{\text{old}} \) and \( \text{Finished}_{\text{old}} \) the corresponding hypergraph and set of terminated vertices at that state respectively. Since VISIT3\((u)\) has performed a depth-first search from the vertex \( U \) in \( \text{graph}(H_{\text{old}}) \), all the vertices reachable from \( U \) in \( H_{\text{old}} \) stand in \( \text{Finished}_{\text{old}} \).

We claim that the invariant is then preserved by the following vertex merging steps. The graph arcs which may be added by the latter leave vertices in \( S \), and consequently not from elements in \( \text{Finished} \) (by Invariant 2). It follows that the set of reachable vertices from elements of \( \text{Finished}_{\text{old}} \) is not changed by future vertex merging steps. As a result, all the vertices reachable from \( U \) in \( \text{graph}(H_{\text{cur}}) \) are elements of \( \text{Finished}_{\text{old}} \). Since by Invariant 5 \( \text{Finished}_{\text{old}} \subseteq \text{Finished} \), this proves the whole invariant in the state \( \text{cur} \). \( \square \)

\textbf{Invariant 6.} In the digraph \( \text{graph}(H_{\text{cur}}) \), at the call to VISIT3\((u)\), \( u \) is reachable from a vertex \( W \) such that \( \text{index}[W] \) is defined if, and only if, \( W \) belongs to the stack \( S \).

\textit{Proof.} The “if” part can be shown by induction. When the function VISIT3\((u)\) is called from Line 28, the stack \( S \) is empty, so that this is obvious. Otherwise, it is called from Line 43 during the execution of VISIT3\((x)\). Then \( X = \text{FIND}(x) \) is reachable from any vertex in the stack, since \( x \) was itself reachable from any vertex in the stack at the call to \( \text{FIND}(X) \) (inductive hypothesis) and that this reachability property is preserved by potential vertex merging steps (Proposition 3). As \( u \) is obviously reachable from \( X \), this shows the statement.

Conversely, suppose that \( \text{index}[W] \) is defined, and \( W \) is not in the stack. According to Invariant 2 \( W \) is necessarily an element of \( \text{Finished} \). Hence \( u \) also belongs to \( \text{Finished} \) by Invariant 5 which is a contradiction since this cannot hold at the call to VISIT\((u)\). \( \square \)

\textbf{Invariant 7.} Let \( a \in A \) such that \( |T(a)| > 1 \). Consider a state \( \text{cur} \) of the algorithm TERMINAL.SCC3 in which \( r_a \) is defined.

Then \( c_a \) is equal to the number of elements \( x \in T(a) \) such that \( \text{index}[x] \) is defined and \( \text{FIND}(x) \) is reachable from \( \text{FIND}(r_a) \) in \( \text{graph}(H_{\text{cur}}) \).

\textit{Proof.} Since at Line 28, \( c_a \) is incremented only if \( R_a = \text{FIND}(r_a) \) belongs to \( S \), we already know using Invariant 5 that \( c_a \) is equal to the number of elements \( x \in T(a) \) such that, at the call to VISIT3\((x)\), \( x \) was reachable from \( \text{FIND}(r_a) \).

Now, let \( x \in V \), and consider a state \( \text{cur} \) of the algorithm in which \( r_a \) and \( \text{index}[x] \) are both defined, and \( \text{FIND}(r_a) \) appears in the stack \( S \). Since \( \text{index}[x] \)
is defined, \textsc{Visit3} has been called on \( x \), and let \( \text{old} \) be the state of the algorithm at that point. Let us denote by \( \mathcal{H}_{\text{old}} \) and \( \mathcal{H}_{\text{cur}} \) the current hypergraphs at the states \( \text{old} \) and \( \text{cur} \) respectively. Like previously, we may add a hypergraph as last argument of the function \textsc{Find} to distinguish its execution in the states \( \text{old} \) and \( \text{cur} \). We claim that \( \textsc{Find}(r_a, \mathcal{H}_{\text{cur}}) \prec_{\text{graph}(\mathcal{H}_{\text{cur}})} \textsc{Find}(x, \mathcal{H}_{\text{cur}}) \) if, and only if, \( \textsc{Find}(r_a, \mathcal{H}_{\text{old}}) \prec_{\text{graph}(\mathcal{H}_{\text{old}})} x \). The “if” part is due to the fact that reachability in \( \text{graph}(\mathcal{H}_{\text{old}}) \) is not altered by the vertex merging steps (Proposition 3). Conversely, if \( x \) is not reachable from \( \textsc{Find}(r_a, \mathcal{H}_{\text{old}}) \) in \( \mathcal{H}_{\text{old}} \), then \( \textsc{Find}(r_a, \mathcal{H}_{\text{old}}) \) is not in the call stack \( S_{\text{old}} \) (Invariant 6), so that it is an element of \( \text{Finished}_{\text{old}} \). But \( \text{Finished}_{\text{old}} \subseteq \text{Finished}_{\text{cur}} \), which contradicts our assumption since by Invariant 2 an element cannot be stored in \( \text{Finished}_{\text{cur}} \) and \( S_{\text{cur}} \) at the same time. It follows that if \( r_a \) is defined and \( \textsc{Find}(r_a) \) appears in the stack \( S \), \( \text{col}_a \) is equal to the number of elements \( x \in T(a) \) such that \( \text{index}[x] \) is defined and \( \textsc{Find}(r_a) \prec_{\text{graph}(\mathcal{H}_{\text{cur}})} \textsc{Find}(x) \).

Let \( \text{cur} \) be the state of the algorithm when \( \textsc{Find}(r_a) \) is moved from \( S \) to \( \text{Finished} \). The invariant still holds. Besides, in the future states \( \text{new} \), \( \text{col}_a \) is not incremented because \( \textsc{Find}(r_a, \mathcal{H}_{\text{cur}}) \in \text{Finished}_{\text{cur}} \subseteq \text{Finished}_{\text{new}} \) (Invariant 3), so that \( \textsc{Find}(r_a, \mathcal{H}_{\text{new}}) = \textsc{Find}(r_a, \mathcal{H}_{\text{cur}}) \), and the latter cannot appear in the stack \( S_{\text{new}} \) (Invariant 2). Furthermore, any vertex reachable from \( R_a = \textsc{Find}(r_a, \mathcal{H}_{\text{new}}) \) in \( \text{graph}(\mathcal{H}_{\text{new}}) \) belongs to \( \text{Finished}_{\text{new}} \) (Invariant 3). It even belongs to \( \text{Finished}_{\text{cur}} \), as shown in the second part of the proof of Invariant 3 (emphasized sentence). It follows that the number of reachable vertices from \( \textsc{Find}(r_a) \) has not changed between states \( \text{cur} \) and \( \text{new} \). Therefore, the invariant on \( \text{col}_a \) will be preserved, which completes the proof.

}\hfill \square

**Proposition 16.** In \textsc{Visit3}, the assignment at Line 62 does not change the value of \( \text{F} \).

**Proof.** It can be shown by strong induction on the number \( p \) of times that this line has been executed. Suppose that we are currently at Line 53, and let \( X_1, \ldots, X_q \) be the elements of the stack located above the root \( U = X_1 \) of the terminal SCC of \( \text{graph}(\mathcal{H}_{\text{cur}}) \). Any arc \( a \) which will be transferred to \( F \) from Line 53 to Line 60 satisfies \( \text{col}_a = |T(a)| > 1 \) and \( \textsc{Find}(r_a) = X_i \) for some \( 1 \leq i \leq q \) (since at 53 \( F \) is initially empty). Invariant 7 implies that for all elements \( x \in T(a) \), \( \text{Find}(x) \) is reachable from \( X_i \) in \( \text{graph}(\mathcal{H}_{\text{cur}}) \), so that by terminality of the SCC \( C = \{X_1, \ldots, X_q\} \), \( \text{Find}(x) \) belongs to \( C \), i.e., there exists \( j \) such that \( \text{Find}(x) = X_j \). It follows that at Line 60 \( \text{Find}(x) = U \) for all \( x \in T(a) \). Then, we claim that \( \text{collected}_a = \text{false} \) at Line 60. Indeed, \( a' \in A \) satisfies \( \text{collected}_{a'} = \text{true} \) if, and only if:

- either it has been copied to \( F \) at Line 23 in which case \( |T(a')| = 1 \),
- or it has been copied to \( F \) at the \( r \)-th execution of Line 62 with \( r < p \). By induction hypothesis, this means that \( a' \) has been pushed on a stack \( F_X \) and then popped from it strictly before the \( r \)-th execution of Line 62.

Observe that a given hyperarc can be popped from a stack \( F_x \) at most once during the whole execution of \textsc{TerminalSCC3}. Here, \( a \) has been popped from \( F_X \) after the \( p \)-th execution of Line 62 and \( |T(a)| > 1 \). It follows that \( \text{collected}_a = \text{false} \).

Conversely, suppose for that, at Line 62 \( \text{collected}_a = \text{false} \), and all the \( x \in T(a) \) satisfies \( \text{Find}(x) = U \). Clearly, \( |T(a)| > 1 \) (otherwise, \( a \) would have been placed into \( F \) at Line 23 and \( \text{collected}_a \) would be equal to \( \text{true} \)). Few steps before, at Line 53 \( \text{Find}(x) \) is equal to one of \( X_j \), \( 1 \leq j \leq q \). Since \( \text{index}[X_j] \) is defined (\( X_j \) is an element of the stack \( S \)), by Invariant 1 \( \text{index}[x] \) is also defined for all
$x \in T(a)$, hence, the root $r_a$ is defined by Invariant 4. Besides, $\text{Find}(r_a)$ is equal to one of the $X_j$, say $X_k$ (since $r_a \in T(a)$). As all the $\text{Find}(x)$ are reachable from $\text{Find}(r_a)$ in $\text{graph}(H_{cur})$, then $c_a = |T(a)|$ using Invariant 7. It follows that $a$ has been pushed on the stack $F_{Ra}$, where $R_a = \text{Find}(r_a, H_{old})$ in an previous state old of the algorithm. As $\text{collected}_a = false$, $a$ has not been popped from $F_{Ra}$, and consequently, the vertex $R_a$ of $H_{old}$ has not involved in a vertx merging step. Therefore, $R_a$ is still equal to $\text{Find}((r_a, H_{cur}) = X_k$. It follows that at Line 53, $a$ is stored in $FX_k$, and thus it is copied to $F$ between Lines 54 and 60. This completes the proof.

We now can prove the correctness of $\text{TerminalScc}$.

**Theorem 4.** By Proposition 16, Line 62 can be safely removed in $\text{Visit3}$. It follows that the booleans $\text{collected}_a$ are now useless, so that Line 5, the loop from Lines 35 to 37, and Line 63 can be also removed. After that, we precisely obtain the algorithm $\text{TerminalScc}$. Proposition 15 completes the proof.

INRIA Saclay – Ile-de-France and CMAP, Ecole Polytechnique, France
E-mail address: xavier.allamigeon@inria.fr