Computing the Extreme Points of Tropical Polyhedra*

Xavier Allamigeon\textsuperscript{1,3}, Stéphane Gaubert\textsuperscript{2}, and Éric Goubault\textsuperscript{3}

\textsuperscript{1} EADS Innovation Works, SE/IS – Suresnes, France
\textsuperscript{2} INRIA Saclay and CMAP, Ecole Polytechnique, France
\textsuperscript{3} CEA, LIST MeASI – Gif-sur-Yvette, France
\{firstname.lastname@eads.net, inria.fr, cea.fr\}

Abstract. We present a novel approach to compute efficiently all the extreme elements of tropical polyhedra, which are the analogues of convex polyhedra in the max-plus algebra. It relies on a method successively eliminating the inequalities, and in which redundant generators are removed a priori. The cornerstone of this method is a new combinatorial characterization of the extreme elements of tropical polyhedra defined by inequalities: we prove that the extremality of an element amounts to the existence of a strongly connected component reachable from any other in a directed hypergraph. We show that the latter property can be checked in almost linear time in the size of the hypergraph. We give theoretical bounds and experimental results on various benchmarks, including instances arising from an application to static analysis, showing that our algorithm is faster than the previous ones by several orders of magnitude.

\* This work was partially supported by Arpege programme of the French National Agency of Research (ANR), project “ASOPT”, number ANR-08-SEGI-005.
1 Introduction

Tropical polyhedra are the analogues of convex polyhedra in tropical algebra. The latter deals with structures like the max-plus semiring $\mathbb{R}_{\text{max}}$ (also called max-plus algebra), which is the set $\mathbb{R} \cup \{-\infty\}$, equipped with the addition $x \oplus y := \max(x, y)$ and the multiplication $x \otimes y := x + y$.

The study of the analogues of convex sets in tropical or max-plus algebra is an active research topic, and has been treated in various guises. It arose in the work of Zimmerman [1], following a way opened by Vorobyev [2], motivated by optimization theory. Max-plus cones were studied by Cuninghame-Green [3]. Their theory was independently developed by Litvinov, Maslov and Shpiz [4] (see also [5]) with motivations from variations calculus and asymptotic analysis, and by Cohen, Gaubert, and Quadrat [6, 7] who initiated a “geometric approach” of discrete event systems [8], further developed in [9, 10]. In [11, 12], Singer recasted this theory in the setting of abstract convexity [13]. Briec and Horvath [14] had similar motivations. The field has attracted recently much attention after the work of Develin and Sturmfels [15], who pointed out important connections with tropical geometry and applications to phylogenetical analysis. They developed a new combinatorial point of view, thinking of tropical polyhedra as polyhedral complexes in the usual sense (see also the series of works by Joswig, Yu, and the same authors [16–19]).

A tropical polyhedron can be represented in two different ways, either internally, in terms of extreme points and rays, or externally, in terms of linear inequalities (see Sect. 2 for details). As in the classical case, passing from the external description of a polyhedron to its internal description is a fundamental computational issue. This is the object of the present paper.

Butkovič and Hegedus [20] gave an algorithm to compute the generators of a tropical polyhedral cone described by linear inequalities, with motivations from economic problems. Gaubert gave a similar one and derived the equivalence between the internal and external representations [21, Ch. III] (see also [22]). Both algorithms rely on a successive elimination of inequalities, but have the inconvenient of squaring at each step the number of candidate generators. Then, an elimination technique must be incorporated to eliminate the redundant candidates. A first implementation of these ideas was included in the Maxplus toolbox of Scilab [23]. Joswig developed a different approach, implemented in Polymake [24], in which a tropical polytope is represented as a polyhedral complex [19]. However, the latter has many more generators.

The present work grew out of an application to software verification by static analysis [25]. Tropical polyhedra are used to automatically discover complex relations involving the operators min and max which hold over the variables of a program. These relations allow to prove the absence of bugs in the program. The execution time of the static analyzer directly depends on the efficiency of the algorithm computing the extreme elements of tropical polyhedra. While the results were encouraging, all the previously developed algorithms were too slow to make the static analyzer usable in practice. Similar limitations were observed in other applications [9, 10].

Contributions. We develop a new algorithm which computes the extreme elements of tropical polyhedra. It is based on a successive elimination of inequalities, and a result (Th. 1) allowing one, given a polyhedron $P$ and a tropical halfspace $H$, to construct a list of candidates for the generators of $P \cap H$. The key ingredient is a combinatorial characterization of the extreme generators of a polyhedron defined externally (Th. 2 and 3): we reduce the verification of the extremality of a candidate to the existence of a strongly connected component reachable from any other in a directed hypergraph. We show that the latter problem can be solved very efficiently (Th. 4), a result which is of independent interest. We include a complexity analysis and experimental results (Sect. 6), showing that the new algorithm outperforms the earlier ones, allowing us to solve instances which were previously by far inaccessible.

2 Tropical Polyhedra and Polyhedral Cones

The neutral elements for the addition $\oplus$ and multiplication $\otimes$, i.e., the zero and the unit, will be denoted by $0 := -\infty$ and $1 := 0$, respectively. The tropical analogues of the operations on
vectors and matrices are defined naturally. The elements of $\mathbb{R}_{\max}^d$, the $d$th fold Cartesian product of $\mathbb{R}_{\max}$, will be thought of as vectors, and denoted by bold symbols, like $x = (x_1, \ldots, x_d)$.

A tropical halfspace is a set of the vectors $x = (x_i) \in \mathbb{R}_{\max}^d$ verifying an inequality constraint of the form $\max_{1 \leq i \leq d} a_i + x_i \leq \max_{1 \leq i \leq d} b_i + x_i$, with $a_i, b_i \in \mathbb{R}_{\max}^d$. A tropical polyhedral cone is defined as the intersection of $n$ halfspaces. It can be equivalently written as the set of the solutions of a system of inequality constraints $Ax \leq Bx$. Here, $A = (a_{ij})$ and $B = (b_{ij})$ are $n \times d$ matrices with entries in $\mathbb{R}_{\max}$, concatenation denotes the matrix product (with the laws of $\mathbb{R}_{\max}$), and $\leq$ denotes the standard partial ordering of vectors. For sake of readability, tropical polyhedral cones will be often referred to as polyhedral cones or cones.

Tropical polyhedral cones are known to be generated by their extreme rays [26–28]. Recall that a ray is the set of scalar multiples of a non-zero vector $u$. It is extreme in a cone $C$ if $u \in C$ and if $u = v \oplus w$ with $v, w \in C$ implies that $u = v$ or $u = w$. A finite set $G = \{g^i\}_{i \in I}$ of vectors is said to generate a polyhedral cone $C$ if each $g^i$ belongs to $C$, and if every vector $x$ of $C$ can be written as a tropical linear combination $\bigoplus_j \lambda_j g^j$ of the vectors of $G$ (with $\lambda_i \in \mathbb{R}_{\max}$). The tropical analogue of the Minkowski theorem [27,28] shows in particular that every generating set of a cone that is minimal for inclusion is obtained by selecting precisely one (non-zero) element in each extreme ray. Note that in tropical linear combinations, the requirement that $\lambda_i$ be non-negative is omitted. Indeed, $0 = -\infty \leq \lambda$ holds for all scalar $\lambda \in \mathbb{R}_{\max}$.

A tropical polyhedron of $\mathbb{R}_{\max}^d$ is the affine analogue of a tropical polyhedral cone. It is defined by a system of inequalities of the form $Ax \oplus c \leq Bx \oplus d$. It can be also expressed as the set of the tropical affine combinations of its generators. The latter are of the form $\bigoplus_{i \in I} \lambda_i u^i \oplus \bigoplus_{j \in J} \mu_j r^j$, where the $(u^i)_{i \in I}$ are the extreme points, the $(r^j)_{j \in J}$ a set formed by one element of each extreme ray, and $\bigoplus_{i} \lambda_i = 1$. It is known [7,27] that every tropical polyhedron of $\mathbb{R}_{\max}^d$ can be represented by a tropical polyhedral cone of $\mathbb{R}_{\max}^{d+1}$ thanks to an analogue of the homogenization method used in the classical case (see [29, Sect. 1.5]). Then, the extreme rays of the cone are in one-to-one correspondence with the extreme generators of the polyhedron. That is why, in the present paper, we will only state the main results for cones, leaving to the reader the derivation of the affine analogues, along the lines of [27].

In the sequel, we will illustrate our results on the polyhedral cone $C$ given in Fig. 1, defined by the system in the right side. The left side is a representation of $C$ in barycentric coordinates: each element $(x_1, x_2, x_3)$ is represented as a barycenter with weights $(e^{x_1}, e^{x_2}, e^{x_3})$ of the three vertices of the outermost triangle. Then two elements of a same ray are represented by the same point. The cone $C$ is depicted in solid gray (the black border is included), and is generated by the extreme elements $g^0 = (0, 0, 0)$, $g^1 = (-2, 1, 0)$, $g^2 = (2, 2, 0)$, and $g^3 = (0, 0, 0)$.

### 3 Principle of the Algorithm

Our algorithm is based on a successive elimination of inequalities. Given a polyhedral cone $C$ defined by a system of $n$ constraints, the algorithm computes by induction on $k$ ($0 \leq k \leq n$) a generating set $G_k$ of the intermediate cone defined by the first $k$ constraints. Then $G_n$ forms a generating set of the cone $C$. Passing from the set $G_k$ to the set $G_{k+1}$ relies on a result which, given a polyhedral cone $\mathcal{K}$ and a tropical halfspace $\mathcal{H} = \{ x \mid ax \leq bx \}$, allows to build a generating set $G'$ of $\mathcal{K} \cap \mathcal{H}$ from a generating set $G$ of $\mathcal{K}$.
Theorem 1. Let $K$ be a polyhedral cone generated by a set $G \subseteq \mathbb{R}^d$, and $H = \{ x \mid ax \leq bx \}$ a tropical halfspace $(a, b \in \mathbb{R}^{1 \times d})$. Then the polyhedral cone $K \cap H$ is generated by the set:

$$\{ g \in G \mid ag \leq bg \} \cup \{ (ah)g \oplus (bg)h \mid g, h \in G, ag \leq bg, and ah > bh \}.$$ 

As an example, consider the cone defined in Fig. 1 and the constraint $x_2 \leq x_3 + 2.5$ (depicted in semi-transparent gray in Fig. 2). The three generators $g^1, g^2$, and $g^3$ satisfy the constraint, while $g^0$ does not. Their combinations are the elements $h^{1,0}, h^{2,0}$, and $h^{3,0}$ respectively.

Nevertheless, Th. 1, and subsequently the whole inductive approach, may return non-extreme elements. (In the example represented in Fig. 2, only $h^{1,0}$ is extreme, whereas $h^{2,0}$ and $h^{3,0}$ are not.) As mentioned above, the extreme rays of a cone form a minimal generating set, so that it is useless to compute non-extreme elements. On top of that, the latter considerably degrade the performance of the approach: at each inductive step, the number of generators, among which many can be non-extreme, grows quadratically in the worst case (because of the pairwise combinations in Th. 1 of the $g$ and $h$). Hence the complexity of the inductive method in total is $O(d^2)$ both in time and space, which is untractable.

For that reason, the approach is improved by including the elimination of non-extreme elements at each step of the induction. This idea is implemented in the algorithm COMPUTE-EXTREME (Fig. 3), which returns one element of each extreme ray of the cone $C$. The argument $n$ corresponds to the number of constraints of the system $Ax \leq Bx$. When it is equal to 0, the cone coincides with $\mathbb{R}^d$, so that it is generated by the tropical canonical basis $(e^i)_{1 \leq i \leq d}$ (where $e^i_j = 1$ and $e^i_j = 0$ if $j \neq i$). Otherwise, the system is split into $Cx \leq Dx$ formed by the $(n-1)$ first inequalities, and $ax \leq bx$ by the last one. Then the elements provided by Th. 1 are computed from the set $G$ of extreme elements of the intermediary cone $D = \{ x \mid Cx \leq Dx \}$. Note that the extremality test (Line 10) is applied only on the combinations $(ag^i)g^i \oplus (bg^i)g^j$, and not on the elements $g \in G^\mathcal{S}$ which satisfies $ag \leq bg$. Indeed, the latter belong to $C$, and by induction hypothesis, they are extreme in the cone $D$, which is included in $C$. Therefore, they can be shown to be also extreme in $C$. Furthermore, before being appended to the set $H$, every combination $h = (ag^i)g^i \oplus (bg^j)g^j$ is first normalized to the element $\kappa h$, so that the set $H$ contains only one element of each represented ray.

The efficiency of the extremality test used at Line 10 plays a crucial role in the whole performance of COMPUTE-EXTREME. The existing extremality criteria, which are implemented in the Maxplus toolbox of SCILAB [23] and in our previous work [25], are based on the following property: $h$ is extreme in the cone generated by a given set $H$ if and only if $h$ can not be expressed as the tropical linear combination of the elements of $H$ which are not proportional to it. This property can be checked in $O(d \times |H|)$ time using residuation (see [28] for algorithmic details). If this criterion is used in our algorithm, the amortized complexity of the extremality test (over the sequence of tests performed in the loop from Lines 8 to 13) is $O(d \times |G|^2)$, where $G$ is the set of the extreme rays of the intermediary cone $D$. Unfortunately, as confirmed theoretically and by our experiments, the set $G$ can be very large, even if the cone $C$ has few extreme rays (see Fig. 3. Our main algorithm computing the extreme rays of tropical cones.)
Given a polyhedral cone \( C \subset \mathbb{R}^d \) and on Th. 14 of [28], shows that extremality can be expressed as a minimality property:

The following lemma, which is a variation on the proof of Th. 3.1 Preliminaries on Extremality. The following lemma, which is a variation on the proof of Th. 3.1

4 Characterizing Extremality from Inequality Constraints

Efficiently characterize the extremality of a vector in a cone defined by inequality constraints. The former is the bottleneck of the algorithm, and the latter may not terminate in a practical time even for small values of \( d \) or \( n \) (see Sect. 6). This motivated us to develop a novel approach to efficiently characterize the extremality of a vector in a cone defined by inequality constraints.

Sect. 6 for benchmarks and a discussion on the size of \( G \). As a consequence, this extremality test is the bottleneck of the algorithm, and the latter may not terminate in a practical time even for small values of \( d \) or \( n \) (see Sect. 6). This motivated us to develop a novel approach to efficiently characterize the extremality of a vector in a cone defined by inequality constraints.

4 Characterizing Extremality from Inequality Constraints

Preliminaries on Extremality. The following lemma, which is a variation on the proof of Th. 3.1 of [27] and on Th. 14 of [28], shows that extremality can be expressed as a minimality property:

**Proposition 1.** Given a polyhedral cone \( C \subset \mathbb{R}^d \), \( g \) is extreme if and only if there exists \( 1 \leq t \leq d \) such that \( g \) is a minimal element of the set \( \{ x \in C \mid x_t = g_t \} \), i.e. \( g \in C \) and for each \( x \in C \), \( ( x \leq g \) and \( x_t = g_t ) \) implies \( x = g \). In that case, \( g \) is said to be extreme of type \( t \).

In Fig. 4, the light gray area represents the set of the elements \( (x_1, x_2, x_3) \) of \( \mathbb{R}^3_{\text{max}} \) such that \( (x_1, x_2, x_3) \leq g^2 \) implies \( x_1 < g_1^2 \). It clearly contains the whole cone except \( g^2 \), which shows that \( g^2 \) is extreme of type 1.

A tropical segment is the set of the tropical linear combinations of two points. Using the fact that a tropical segment joining two points of a polyhedral cone \( C \) yields a continuous path included in \( C \), one can check that \( g \) is extreme of type \( t \in C \) if and only if there is a neighborhood \( N \) of \( g \) such that \( g \) is minimal in \( \{ x \in C \cap N \mid x_t = g_t \} \). Thus, extremality is a local property.

Finally, the extremality of an element \( g \) in a cone \( C \) can be equivalently established by considering the vector formed by its non-0 coordinates. Formally, let \( \text{supp}(x) := \{ i \mid x_i \neq 0 \} \) for any \( x \in \mathbb{R}^d_{\text{max}} \). One can show that \( g \) is extreme in \( C \) if and only if it is extreme in \( \{ x \in C \mid \text{supp}(x) \subset \text{supp}(g) \} \). This allows to assume that \( \text{supp}(g) = \{ 1, \ldots, d \} \) without loss of generality.

Expressing Extremality Using the Tangent Cone. For now, the polyhedral cone \( C \) is supposed to be defined by a system \( Ax \leq Bx \) of \( n \) inequalities.

Consider an element \( g \) of the cone \( C \), which we assume, from the previous discussion, to satisfy \( \text{supp}(g) = \{ 1, \ldots, d \} \). In this context, the tangent cone of \( C \) at \( g \) is defined as the tropical polyhedral cone \( T(g, C) \) of \( \mathbb{R}^d_{\text{max}} \) given by the system of inequalities

\[
\max_{i \in \text{arg max}(A_k g)} x_i \leq \max_{j \in \text{arg max}(B_k g)} x_j \quad \text{for all } k \text{ such that } A_k g = B_k g, \tag{1}
\]

where for each row vector \( c \in \mathbb{R}^{1 \times d}_{\text{max}} \), \( \text{arg max}(cg) \) is defined as the argument of the maximum \( cg = \max_{1 \leq i \leq d} (c_i + g_i) \), and where \( A_k \) and \( B_k \) denote the \( k \)th rows of \( A \) and \( B \), respectively.

The tangent cone \( T(g, C) \) provides a local description of the cone \( C \) around \( g \): **Proposition 2.** There exists a neighborhood \( N \) of \( g \) such that for all \( x \in N \), \( x \) belongs to \( C \) if and only if it is an element of \( g + T(g, C) \).

As an illustration, Fig. 5 depicts the set \( g^2 + T(g^2, C) \) (in semi-transparent light gray) when \( C \) is the cone given in Fig. 1. Both clearly coincide in the neighborhood of \( g^2 \). Since extremality is a local property, it can be equivalently characterized in terms of the tangent cone. Let \( 1 \) be the element of \( \mathbb{R}^d_{\text{max}} \) whose all coordinates are equal to \( 1 \).
Proposition 3. The element $g$ is extreme in $C$ if and only if the vector 1 is extreme in $T(g, C)$.

The problem is now reduced to the characterization of the extremality of the vector 1 in a $\{0, 1\}$-cone, i.e. defined by a system of the form $C x \leq D x$ where $C, D \in \{0, 1\}^{n \times d}$. The following proposition states that only $\{0, 1\}$-vectors (i.e. elements of $\{0, 1\}^d$) have to be considered:

Proposition 4. Let $\mathcal{D} \subset \mathbb{R}_{\text{max}}^d$ be a cone defined by a system $C x \leq D x$ with $C, D \in \{0, 1\}^{n \times d}$. Then 1 is extreme of type $t$ if and only if $\mathcal{D}$ is the unique element $x$ of $\mathcal{D} \cap \{0, 1\}^d$ satisfying $x_t = 1$.

The following criterion of extremality is a direct consequence of Prop. 3 and 4:

Theorem 2. Let $C \subset \mathbb{R}_{\text{max}}^d$ be a polyhedral cone. The element $g \in C$ is extreme of type $t$ if and only if 1 is the unique $\{0, 1\}$-element of the tangent cone $T(g, C)$ whose $t$-th coordinate is 1.

Figure 6 shows that in our running example, the $\{0, 1\}$-elements of $T(g^2, C)$ distinct from 1 (in squares) all satisfy $x_1 = 0$. Naturally, testing, by exploration, whether the set of $2^{d-1} \{0, 1\}$-elements $x$ verifying $x_t = 1$ belonging to $T(g, C)$ consists only of 1 does not have an acceptable complexity. Instead, we propose to use the equivalent formulation of the criterion of Th. 2:

$$\forall l \in \{1, \ldots, d\}, \quad \forall x \in T(g, C) \cap \{0, 1\}^d, \quad x_l = 0 \implies x_t = 0. \tag{2}$$

Characterizing Extremality with Directed Hypergraphs. A directed hypergraph is a couple $(N, E)$ such that each element of $E$ is of the form $(T, H)$ with $T, H \subset N$. The elements of $N$ and $E$ are respectively called nodes and hyperedges. Given a hyperedge $e = (T, H) \in E$, the sets $T$ and $H$ represent the tail and the head of $e$ respectively, and are also denoted by $T(e)$ and $H(e)$. Figure 8 depicts an example of hypergraph whose nodes are $u, v, w, x, y, t$, and of hyperedges $e_1 = (\{u\}, \{v\}), e_2 = (\{v\}, \{w\}), e_3 = (\{w\}, \{u\}), e_4 = (\{v, w\}, \{x, y\})$, and $e_5 = (\{w, y\}, \{\})$.

Reachability is extended from digraphs to directed hypergraphs by the following recursive definition: given $u, v \in N$, then $v$ is reachable from $u$ in $H$, which is denoted $u \leadsto_H v$, if $u = v$, or there exists $e \in E$ such that $v \in H(e)$ and all the elements of $T(e)$ are reachable from $u$. This induces a notion of hyperpaths, defined as a sequence of hyperedges of $E$, which can be empty if $u = v$, or of the form $e_1, \ldots, e_p$ with $T(e_i) \subset \{u\} \cup H(e_1) \cup \cdots \cup H(e_{i-1})$ for all $1 \leq i \leq p$, and $v \in H(e_p)$. In our example, $t$ is reachable from $u$ through the hyperpath $e_1, e_2, e_4, e_5$.

The size of a hypergraph $\mathcal{H} = (N, E)$, denoted by size($\mathcal{H}$), is defined as $|N| + \sum_{e \in E}|(T(e))| + |(H(e))|$. In the rest of the paper, directed hypergraphs will be simply referred to as hypergraphs.

We associate to the tangent cone $T(g, C)$ the hypergraph $\mathcal{H}(g, C) = (N, E)$ defined by:

$$N = \{1, \ldots, d\} \quad E = \{ \{\arg\max(B_k g), \arg\max(A_k g)\} | A_k g = B_k g, 1 \leq k \leq n\}.$$ 

The extremality criterion of Eq. (2) suggests to evaluate, given an element of $T(g, C) \cap \{0, 1\}^d$, the effect of setting its $l$-th coordinate to the other coordinates. Suppose that it has been discovered that $x_l = 0$ implies $x_j = \cdots = x_n = 0$. For any hyperedge $e$ of $\mathcal{H}(g, C)$ such that $T(e) \subset \{l, j_1, \ldots, j_n\}$, $x$ satisfies: max$_{i \in H(e)} x_i \leq \max_{j \in T(e)} x_j = 0$, so that $x_i = 0$ for all $i \in H(e)$. Thus, the propagation of the value 0 from the $l$-th coordinate to other coordinates mimicks the inductive definition of the reachability relation from the node $l$ in $\mathcal{H}(g, C)$:

Proposition 5. For all $l \in \{1, \ldots, d\}$, the statement given between brackets in Eq. (2) holds if and only if $l$ is reachable from $l$ in the hypergraph $\mathcal{H}(g, C)$.

Hence, the extremity criterion can be restated thanks to some considerations on the strongly connected components of $\mathcal{H}(g, C)$. The strongly connected components (SCCS for short) of a hypergraph $\mathcal{H}$ are the equivalence classes of the equivalence relation $\equiv_{\mathcal{H}}$, defined by $u \equiv_{\mathcal{H}} v$ if $u \leadsto_{\mathcal{H}} v$ and $v \leadsto_{\mathcal{H}} u$. They form a partition of the set of nodes of $\mathcal{H}$. They can be partially ordered by the relation $\preceq_{\mathcal{H}}$, defined by $C_1 \preceq_{\mathcal{H}} C_2$ if $C_1$ and $C_2$ admit a representative $u$ and $v$ respectively such that $v \leadsto_{\mathcal{H}} u$ (beware of the order of $v$ and $u$ in $v \leadsto_{\mathcal{H}} u$). Then Prop. 5 and Th. 2 imply the following statement:

Theorem 3. Let $C \subset \mathbb{R}_{\text{max}}^d$ be a polyhedral cone, and $g \in C$. Then $g$ is extreme if and only if the set of the SCCS of the hypergraph $\mathcal{H}(g, C)$, partially ordered by $\preceq_{\mathcal{H}(g, C)}$, admits a least element.
5 Determining the Minimal SCCs in Hypergraphs

Following Th. 3, we describe an algorithm which determines the minimal SCCs for the order $\preceq$ in hypergraphs. In particular, it returns the number of such SCCs, which is equal to 1 if and only if there exists a least SCC. Our approach is best understood by presenting first an instrumentation of Tarjan’s algorithm [30] which applies only on directed graphs (digraphs), and has a linear time complexity. The order $\preceq$ can be defined on the SCCs of digraphs as well. Note that in a digraph (or in a hypergraph), a SCC is minimal for this order if and only if it has no leaving (hyper)edge (except those entering itself). Therefore, a SCC is a least element if and only if it is reachable from any other SCC. Besides, if $C_1 \preceq_G C_2$ in a digraph $G$, then $C_1$ is greater than or equal to $C_2$ in the topological order.

The algorithm for digraphs is given in Fig. 7 with the version of MINSCCCOUNT calling the visiting function GVisit at Line 10 (parts highlighted in gray can be ignored). The array index tracks the order in which the nodes are visited: $\text{index}[u] = i$ if the node $u$ is the $i$th one to be visited. The value $\text{low}[u]$ is used to determine the minimal index of the nodes which are reachable from $u$ in the digraph. A SCC $C$ is discovered as soon as a node $u$ satisfies $\text{low}[u] = \text{index}[u]$ (Line 27). Then $C$ consists of all the nodes stored in the stack $S$ above $u$. The node $u$ is the node of the SCC which has been visited first, and is called its root. The main difference between our algorithm and Tarjan’s original one is that the nodes $v$ are provided with a boolean $\text{ismin}[v]$ allowing to track the minimality of the SCC. A SCC is minimal if and only if its root $u$ satisfies $\text{ismin}[u] = \text{true}$. In particular, the boolean $\text{ismin}[v]$ is set to false as soon as it is connected to a node $w$ located in a distinct SCC (Line 21) or satisfying $\text{ismin}[w] = \text{false}$ (Line 24). The counter $nb$ determines the number of minimal SCCs which have been discovered (see Line 28). For the sake of brevity, we have removed the operations allowing to return the SCCs. Instead, when a node is discovered to be in a SCC, it is placed in the set $\text{Finished}$ (Line 30).

Nevertheless, this algorithm can not be applied on hypergraphs, since the reachability relation in the latter is much more complex. However, determining the minimal SCCs in digraphs can be helpful. First observe that a digraph $G(\mathcal{H}) = (N, E')$ can be associated to any hypergraph $\mathcal{H} = (N, E)$, by defining $E' = \{(t, h) \mid \{t\}, H \in E \text{ and } h \in H\}$. The digraph $G(\mathcal{H})$ is generated by the simple hyperedges of $\mathcal{H}$, i.e. the elements $e \in E$ such that $|T(e)| = 1$. The minimal SCCs of $\mathcal{H}$ and $G(\mathcal{H})$ coincide in the following remarkable special case:

**Proposition 6.** Let $\mathcal{H}$ be a hypergraph such that each minimal SCC of $G(\mathcal{H})$ is reduced to a singleton. Then $\mathcal{H}$ and $G(\mathcal{H})$ have the same minimal SCCs.

If $f$ is a function from $N$ to an arbitrary set, we denote by $f(\mathcal{H})$ the hypergraph of nodes $f(N)$ and of hyperedges $\{(f(T(e)), f(H(e))) \mid e \in E\}$. The following proposition ensures that, in a hypergraph, merging two nodes of a same SCC does not alter the reachability relation:

**Proposition 7.** Let $\mathcal{H} = (N, E)$ be a hypergraph, and let $x, y \in N$ such that $x \equiv\mathcal{H} y$. Consider the function $f$ mapping any node distinct from $x$ and $y$ to itself, and both $x$ and $y$ to a same node $z$ (with $z \notin N \setminus \{x, y\}$). Then $u \rightsquigarrow\mathcal{H} v$ if and only if $f(u) \rightsquigarrow f(\mathcal{H}) f(v)$.

Thus, the minimal SCCs of $\mathcal{H}$ and $f(\mathcal{H})$ are in one-to-one correspondence. Using Prop. 6 and 7, we now sketch a method which computes the minimal SCCs in a hypergraph $\mathcal{H}$: starting from the hypergraph $\mathcal{H}_{\text{cur}}$ image of $\mathcal{H}$ by the map $u \mapsto \{u\}$,

(i) compute the minimal SCCs of the digraph $G(\mathcal{H}_{\text{cur}})$.
(ii) if one of them, say $C$, is not reduced to a singleton, replace $\mathcal{H}_{\text{cur}}$ by $f(\mathcal{H}_{\text{cur}})$, where $f$ merges all the elements $U$ of $C$ into the node $\bigcup_{U \subseteq C} U$. Then go back to Step (i).
(iii) otherwise, return the number of minimal SCCs of the digraph $G(\mathcal{H}_{\text{cur}})$.

Each time the node merging step (Step (ii)) is executed, new edges may appear in the digraph $G(\mathcal{H}_{\text{cur}})$. This situation is illustrated by Fig. 9. In both sides, the edges of $G(\mathcal{H}_{\text{cur}})$ are depicted in solid, and the non-simple hyperedges of $\mathcal{H}_{\text{cur}}$ in dotted line. The nodes of $\mathcal{H}_{\text{cur}}$ contain subsets of $N$, but enclosing braces are omitted. Applying Step (i) from node $u$ (left side) discovers a
minimal SCC formed by \( u, v, \) and \( w \) in the digraph \( G(H_{\text{cur}}) \). At Step (ii) (right side), the nodes are merged, and the hyperedge \( e_3 \) is transformed into two graph edges leaving the new node.

The termination of this method is ensured by the fact that the number of nodes in \( H_{\text{cur}} \) is strictly decreased each time Step (ii) is applied. When the method is terminated, minimal SCCs of \( H_{\text{cur}} \) are all reduced to single nodes, which contain subsets of \( N \). Propositions 6 and 7 prove that these subsets are precisely the minimal SCCs of \( H \). Besides, the method returns the exact number of minimal SCCs in \( H \). However, in order for this approach to be efficient, the algorithm should avoid computing a same minimal SCC several times. For this reason, we propose to directly integrate the node merging step in our first algorithm on digraphs.

First observe that the nodes of the hypergraph \( H_{\text{cur}} \) always form a partition of the initial set \( N \) of nodes. Instead of referring to them as subsets of \( N \), we use a union-find structure, which consists in three functions \texttt{Find}, \texttt{Merge}, and \texttt{MAKESET} (see e.g. [31, Chap. 21]):

- \texttt{Find(u)} returns, for each original node \( u \in N \), the unique node of \( H_{\text{cur}} \) containing \( u \).
- two nodes \( U \) and \( V \) of \( H_{\text{cur}} \) can be merged by \texttt{Merge(U,V)}), which returns the new node.
- the “singleton” nodes \( \{ u \} \) of the initial \( H_{\text{cur}} \) are created by a function \texttt{MAKESET}.

With this structure, each node of \( H_{\text{cur}} \) is represented by an element \( u \in N \), and then corresponds to the subset \( \{ v \in N \mid \texttt{Find(v)} = u \} \). Nevertheless, we avoid confusion by denoting the nodes of the hypergraph \( H \) by lower case letters, and the nodes of \( H_{\text{cur}} \) by capital ones. By convention, if \( u \in N \), \texttt{Find(u)} will correspond to the associated capital letter \( U \). Note that when an element \( u \in N \) has never been merged with another one, it satisfies \texttt{Find(u)} = \( u \).

```plaintext
1: function MinSccCount(N, E) 40:        auxiliary data update step
2:   n := 0, nb := 0, S := [], Finished := ∅ 41:          \[ \]
3:   for all \( e \in E \) do \( r_e := \text{undef}, c_e := 0 \) 42:          \[ \]
4:   for all \( u \in N \) do 43:          \[ \]
5:     index[u] := undef, low[u] := undef 44:          \[ \]
6:     \( F_u := [], \text{MAKESET}(u) \) 45:          \[ \]
7:   done 46:          \[ \]
8:   for all \( u \in N \) do 47:          \[ \]
9:     if index[u] = undef then 48:          \[ \]
10: \quad \text{GVisit(u)} \quad \text{if } (N, E) \text{ is a digraph} 49:          \[ \]
11: \quad \text{HVisit(u)} \quad \text{if it is a hypergraph} 50:          \[ \]
12:     end 51:          \[ \]
13:   done 52:          \[ \]
14:   return nb 53:          \[ \]
15: function GVisit(u) 54:          \[ \]
16:   index[u] := n, low[u] := n, n := n + 1 55:          \[ \]
17:   ismin[u] := true, push u on the stack \( S \) 56:          \[ \]
18:   for all \( (u, e) \in E \) do 57:          \[ \]
19:     if index[w] = undef then GVisit(w) 58:          \[ \]
20:     if \( w \in \text{Finished} \) then 59:          \[ \]
21:       ismin[u] := false 60:          \[ \]
22:     else 61:          \[ \]
23:       low[u] := min(low[u], low[w]) 62:          \[ \]
24:       ismin[u] := ismin[u] && ismin[w] 63:          \[ \]
25:     end 64:          \[ \]
26:   done 65:          \[ \]
27:   if low[w] = index[u] then 66:          \[ \]
28:     if ismin[u] = true then nb := nb + 1 67:          \[ \]
29:     repeat 68:          \[ \]
30:       pop v from stack \( S \), add \( v \) to \( \text{Finished} \) 69:          \[ \]
31:     until index[v] = index[u] 70:          \[ \]
32:   end 71:          \[ \]
33:   end 72:          \[ \]
34: function HVisit(u) 73:          \[ \]
35:   local \( U := \text{Find}(u) \), local \( F := [] \) 74:          \[ \]
36:   index[U] := n, low[U] := n, n := n + 1 75:          \[ \]
37:   ismin[U] := true, push \( U \) on the stack \( S \) 76:          \[ \]
38:   for all \( e \in E_u \) do 77:          \[ \]
39:     if \( |T(e)| = 1 \) then push \( e \) on \( F \) 78:          \[ \]
40: else 79:          \[ \]
41: if \( r_e = \text{undef} \) then \( r_e := u \) 80:          \[ \]
42: local \( R_e := \text{Find}(r_e) \) 81:          \[ \]
43: if \( R_e \) appears in \( S \) then 82:          \[ \]
44: \quad \text{c}_e := \text{c}_e + 1 83:          \[ \]
45: \quad if \( c_e = |T(e)| \) then 84:          \[ \]
46: \quad \quad push \( e \) on the stack \( F_{R_e} \) 85:          \[ \]
47: end 86: \end
```

Fig. 7. Computing the minimal SCCs in hypergraphs
The resulting algorithm on hypergraphs is obtained by using the function \texttt{HVisit} at Line 10. We present the main ideas used in the correctness proof of the algorithm, highlighting the differences with the first algorithm on digraphs. Albeit \( \mathcal{H}_{\text{cur}} \) is not explicitly manipulated, it can always be inferred as the image of \( \mathcal{H} \) by the function \texttt{FIND}. The visiting function \texttt{HVisit}(u) computes the minimal \texttt{Scc}s reachable from the node \texttt{FIND}(u) in the digraph \( G(\mathcal{H}_{\text{cur}}) \), using the same method as in \texttt{GVisit} (see the part corresponding to Step (i), from Lines 51 to 63). However, as soon as a minimal \texttt{Scc} is discovered, the node merging step (Step (ii)) is executed.

\textbf{Node Merging Step.} It is performed from Lines 66 to 75, when the node \( U = \texttt{FIND}(u) \) is the root of a minimal \texttt{Scc} in \( G(\mathcal{H}_{\text{cur}}) \). All nodes \( V \) which have been discovered in that \texttt{Scc} are merged to \( U \) (Line 71). Let \( \mathcal{H}_{\text{new}} \) be the resulting hypergraph. At Line 75, the stack \( F \) is supposed to contain the new edges of \( G(\mathcal{H}_{\text{new}}) \) leaving the newly “big” node \( U \). If it is empty, \( \{ U \} \) is a minimal \texttt{Scc} of \( G(\mathcal{H}_{\text{new}}) \), hence also of \( \mathcal{H}_{\text{new}} \) (Prop. 6). Thus \( nb \) is incremented. Otherwise, we go back to the beginning of Step (i) to discover minimal \texttt{Scc}s from the new \( U \) in \( G(\mathcal{H}_{\text{new}}) \).

\textbf{Discovering the New Graph Edges.} During the execution of \texttt{HVisit}(u), the local stack \( F \) is used to collect the hyperedges which represent edges leaving the node \texttt{FIND}(u) in the digraph \( G(\mathcal{H}_{\text{cur}}) \). When \texttt{HVisit}(u) is called, \texttt{FIND}(u) is initially equal to \( u \). The loop from Lines 38 to 50 iterates over the set \( E_u \), which consists of the hyperedges \( e \in E \) such that \( u \in T(e) \). (The \( E_u \) can be built in linear time by traversing the set \( E \) before running \texttt{MinSccCount}.) At the end of the loop, \( F \) is filled with all the simple hyperedges leaving \( u = \texttt{FIND}(u) \) in \( \mathcal{H}_{\text{cur}} \), as expected.

Now the main difficulty is to collect in \( F \) the edges which are added to the digraph \( G(\mathcal{H}_{\text{cur}}) \) after a node merging step, without examining all the non-simple hyperedges. To overcome this problem, each non-simple hyperedge \( e \in E \) is provided with two auxiliary data: a counter \( c_e \geq 0 \), and a node \( r_e \) called its \textit{root}. Invariants defining \( r_e \) and \( c_e \) are given in Fig. 10. Observe that at the call to \texttt{HVisit}(u), the counter \( c_e \) of each non-simple hyperedge \( e \in E_u \) is incremented only when \( R_e = \texttt{FIND}(r_e) \) belongs to the stack \( S \) (Line 44). This indeed holds if and only if \texttt{FIND}(u) is reachable from \texttt{FIND}(r_e) in the digraph \( G(\mathcal{H}_{\text{cur}}) \).

According to Fig. 10, when \( c_e = |T(e)| \), all the nodes \( X = \texttt{FIND}(x) \) (for \( x \in T(e) \)) are reachable from \( R_e \) in \( G(\mathcal{H}_{\text{cur}}) \). Now suppose that, later, it is discovered that \( R_e \) belongs to a minimal \texttt{Scc} of \( G(\mathcal{H}_{\text{cur}}) \). Then all the \( X \) stand in the same \texttt{Scc}. (Indeed, if \( C \) is a minimal \texttt{Scc} and \( t \in C \), \( z \) is reachable from \( t \) if and only if \( z \in C \).) Hence, when the node merging step is applied on this \texttt{Scc}, the \( X \) are merged into a single node \( U \). In that case, the hyperedge \( e \) generates new simple edges leaving \( U \) in the new version of the digraph \( G(\mathcal{H}_{\text{cur}}) \). Now let us verify that \( e \) is correctly placed into \( F \) by our algorithm: as soon as \( c_e \) reaches the threshold \( |T(e)| \), \( e \) is placed into a temporary stack \( F_{R_e} \) associated to the node \( R_e \) (Line 46). It is then emptied into \( F \) at Lines 67 or 70 during the node merging step.

For example, in the left side of Fig. 9, the execution of the loop from Lines 38 to 50 during the call to \texttt{HVisit}(v) sets the root of the hyperedge \( e_4 \) to the node \( v \), and \( c_{e_4} \) to 1. Then, during \texttt{HVisit}(w), \( c_{e_4} \) is incremented to \( 2 = |T(e_4)| \). The hyperedge \( e_4 \) is therefore pushed on the stack \( F_v \) (because \( R_{e_4} = \texttt{FIND}(r_{e_4}) = \texttt{FIND}(v) = v \)). Once it is discovered that \( u \), \( v \), and \( w \) form a

| nature | description |
|--------|-------------|
| \( r_e \) | element of \( N \) or \texttt{undef} first node \( x \) of \( T(e) \) to be visited by a call to \texttt{HVisit} |
| \( c_e \) | integer number of elements \( x \in T(e) \) such that \( \text{index}[x] \) is defined, and \texttt{FIND}(x) is reachable from \texttt{FIND}(r_e) in the current graph \( G(\mathcal{H}_{\text{cur}}) \) |
| \( F \) | stack local to \texttt{HVisit}(u) contain the hyperedges which represent edges leaving the node \texttt{FIND}(u) in \( G(\mathcal{H}_{\text{cur}}) \) |
minimal SCC of $G(\mathcal{H}_{\text{cur}})$, $e_4$ is collected into $F$. It then allows to visit the node $x$ from the new node (rightmost hypergraph). A fully detailed execution trace is provided in Appendix E.

**Complexity.** Using disjoint-set forests with union by rank and path compression as union-find structure (see [31, Chapter 21]), a sequence of $p$ operations MAKESET, FIND, or MERGE can be performed in time $O(p \times \alpha(|N|))$, where $\alpha$ is the very slowly growing inverse of the map $x \rightarrow A(x, x)$, and where $A$ is the Ackermann function. Then the following statement holds:

**Theorem 4.** Let $\mathcal{H} = (N, E)$ be a hypergraph. Then $\text{MINSCCCount}(\mathcal{H})$ returns the number of minimal SCCs in $\mathcal{H}$ in time $O(|\text{size}(\mathcal{H})| \times \alpha(|N|))$. Besides, the minimal SCCs are formed by the sets $\{ v \in N | \text{FIND}(v) = U \text{ and } \text{isinmin}(U) = \text{true} \}$.

For any practical value of $x$, $\alpha(x) \leq 4$. That is why the complexity of $\text{MINSCCCount}$ is said to be almost linear in $\text{size}(\mathcal{H})$. Finally, our method does not allow to determine all SCCs, which is apparently a harder problem (see Appendix F for further details).

### 6 Resulting Algorithm and Experiments

Thanks to Th. 3, the equality test $\text{MINSCCCount}(\mathcal{H}(h, \mathcal{C})) = 1$ can be used to determine the extremality of $h$ at Line 10 in $\text{COMPUTEEXTREME}$. Since the hypergraph $\mathcal{H}(h, \mathcal{C})$ can be computed in time $O(nd)$, which is also the order of magnitude of its size, the complexity of each extremality test is $O(n \alpha(d))$. Using the notations of Sect. 3, this represents an improvement factor of $O((G)^2/(n \alpha(d)))$ over the previous extremality test presented in Sect. 3. As discussed below and confirmed by the experiments, the size of $G$, which is the number of extreme rays in the intermediary cone $D$, is much larger than $n \alpha(d)$ in general, so that the performance of $\text{COMPUTEEXTREME}$ is significantly improved.

In classical geometry, the upper bound theorem of McMullen [32] shows that the maximal number of extreme points of a convex polytope in $\mathbb{R}^d$ defined by $n$ inequality constraints is equal to $U(n, d) := (n-\lfloor (d+1)/2 \rfloor) + (n-\lfloor (d+2)/2 \rfloor)$. The cyclic polytopes (see [29]) are known to reach this bound. In the tropical setting, a recent work of Allamigeon, Gauwert, and Katz [33] proves that the number of extreme rays of a tropical polyhedral cone $C$ in $\mathbb{R}^d_{\text{max}}$ defined by $n$ inequalities is bounded by similar quantity, $U(n+d, d-1)$. The latter is of an order of magnitude of $(n+d)^{d/2}$. Besides, natural candidates to be the maximizing cones, named tropical signed cyclic polyhedral cones, have been studied in [33]. While they asymptotically reach the bound $U(n+d, d-1)$ when $d \to +\infty$ and $n$ is fixed, the bound does not seem to be attained in general. However, a lower bound on the number of extreme rays of signed cyclic polyhedral cones have been established for various cases: their order of magnitude are $O((n-2d)2^{d-2})$ when $n \geq 2d$, and $O(d(n-1)2^{d-1})$ when $d \geq 2n+1$. This confirms our previous statement on the size of $G$.

Like Motzkin’s double description method [34, 35], the performance of $\text{COMPUTEEXTREME}$ depends on the size of the sets returned in the intermediary steps. However, the complexity of the elementary step, i.e. the computation of the elements provided by Th. 1 and the elimination of non-extreme ones (Lines 7 to 13), can be precisely characterized: $O(n \alpha(d)/|G|^2)$. In comparison, the same step in the double description method relies on a combinatorial characterization of the adjacency of rays, of complexity $O(n |G|^3)$ [35]. As in general, the size of $G$ is much larger than $n \alpha(d)$, the elementary step in our algorithm has a better complexity than its classical analogue.

A common point between our algorithm and Motzkin’s approach is that the result provided by our algorithm does not depend on the order of the inequalities in the initial system. This order may impact the size of the intermediary sets and subsequently the execution time. In our experiments, inequalities are dynamically ordered during the execution: at each step of the induction, the inequality $a x \leq b x$ is chosen so as to minimize the number of combinations $(ag^i)g^i \oplus (bg^i)g^i$. Note that this strategy does not guarantee that the size of the intermediate sets of extreme elements is smaller. However, it reports better results than without ordering.

**Benchmarks.** The algorithm $\text{COMPUTEEXTREME}$ has been implemented in a prototype written in OCaml. Table 1 reports some experiments for different classes of tropical cones: samples
formed by several cones chosen randomly (referred to as rndx where x is the size of the sample), and signed cyclic cones which are known to have a very large number of extreme elements. For each, the first columns respectively report the dimension d, the number of constraints n, the size of the final set of extreme rays, the mean size of the intermediary sets, and the execution time T (for samples of “random” cones, we give average results). We compare our algorithm with a variant using the extremality criterion which is discussed in Sect. 3 and used in the other existing implementations [23, 25]. Its execution time T' is given in the seventh column. The ratio T/T' shows that our algorithm brings a huge breakthrough in terms of execution time. When the number of extreme rays is of order of 10^4, the second algorithm needs several days to terminate (the asterisk means that its execution was not terminated at the moment of the submission). Therefore, the comparison could not be made in practice for some cases.

Table 1 also reports some benchmarks from applications to static analysis. The experiments odd-even correspond to the static analysis of the odd-even sorting algorithm of i elements. It is a sort of worst case for our analysis. The number of variables and lines in each program is given in the first columns. The analyzer automatically shows that the sorting program returns an array in which the last (resp. first) element is the maximum (minimum) of the array given as input. It clearly benefits from the improvements of ComputeExtreme, as shown by the ratio with the execution time T' of the previous implementation of the static analyzer [25].

## 7 Conclusion and Related Work

This work provides an efficient method to compute all extreme elements of tropical polyhedra and polyhedral cones. It is based on a successive elimination of inequalities, and a criterion to characterize extremality of elements by determining the existence of a SCC reachable from any other in a directed hypergraph, in almost linear time of its size. Existing approaches [20–22, 25] are also based on a successive inequality elimination technique, however their elementary step differs from the result of Th. 1, so that in practice, possibly more extremality tests are performed in their existing implementations [23, 25]. Besides, these tests rely on the method described in Sect. 3. As discussed in Sect. 6, they are consequently less efficient than our algorithm.

The algorithm MINSCCOUNT is also a noteworthy contribution, since as far as we know, there is no method in the existing literature to determine minimal SCCs for the order ≤, in linear (or almost) time in directed hypergraphs. Alternatively, we could have used Gallo’s et al. [36] linear algorithm to compute the set of reachable nodes from each node and obtain the whole reachability graph, but the total complexity would have been \(O(|N| \text{size}(H))\), i.e. a slowdown of almost a factor \(|N|\).

We hope that our method can be considered as a tropical analogue of the very popular double description method of Motzkin, which is able to handle various kinds of convex polyhedra. Avis and Fukuda have developed a method which is polynomial for a subclass of non-degenerate polyhedra [37, 38]. As a future work, it would be very interesting to adapt it for a class of tropical polyhedra. This is a longer term objective, as tropical geometry is a young theory, and several difficulties still have to be overcome: for instance, the notion of faces is not yet well defined in the tropical setting (see [17] for a discussion). However, we believe that our characterization of extreme points in terms of directed hypergraphs is a first step towards understanding the combinatorics of tropical polyhedra, and will be helpful to develop further algorithms.
References

1. Zimmermann, K.: A general separation theorem in extremal algebras. Ekonom.-Mat. Obzor 13(2) (1977) 179–201
2. Vorobyev, N.: Extremal algebra of positive matrices. Elektron. Informationsverarbeitung und Kybernetik 3 (1967) 39–71 in Russian.
3. Cuninghame-Green, R.A.: Minimax algebra. Volume 166 of Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, Berlin (1979)
4. Litvinov, G., Maslov, V., Shpiz, G.: Idempotent functional analysis: an algebraic approach. Math. Notes 69(5) (2001) 696–729
5. Maslov, V., Samborski˘ ı, S., eds.: Idempotent Mathematics and Mathematical Physics. Contemporary Mathematics. American Mathematical Society (2005) 105–129
6. Cohen, G., Gaubert, S., Quadrat, J.P.: Duality and separation theorem in idempotent semimodules. Linear Algebra and Appl. 379 (2004) 395–422
7. Cohen, G., Gaubert, S., Quadrat, J.P.: Max-plus algebra and system theory: where we are and where to go now. Annual Reviews in Control 23 (1999) 207–219
8. Katz, R.D.: Max-plus (A, B)-invariant spaces and control of timed discrete event systems. IEEE Trans. Autom. Control 52(2) (2007) 229–241
9. Develin, M., Yu, J.: Tropical polytopes and cellular resolutions. Experimental Mathematics 16(3) (2007) 277–292
10. Butkoviˇ c, P., Schneider, H., Sergeev, S.: Generators, extremals and bases of max cones. Linear Algebra Appl. 421(2-3) (2007) 394–406
11. Ziegler, G.M.: Lectures on Polytopes. Second edn. Springer-Verlag New York Inc. (1998)
12. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2) (1972) 146–160
13. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education (2001)
14. McMullen, P.: The maximum numbers of faces of a convex polytope. Mathematika 17 (1970) 179–184
15. Allamigeon, X., Gaubert, S., Katz, R.D.: The number of extreme points of tropical polyhedra. Eprint arXiv:math/0906.3492 (submitted) (2009)
36. Gallo, G., Longo, G., Pallottino, S., Nguyen, S.: Directed hypergraphs and applications. Discrete Appl. Math. 42(2-3) (1993) 177–201
37. Avis, D., Fukuda, K.: A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. Discrete Comput. Geom. 8(3) (1992) 295–313
38. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Appl. Math. 65(1-3) (1996) 21–46
A Proof of Th. 1

Let \( G' \) be the family of elements provided by Th. 1. The inclusion \( \text{cone}(G') \subset \mathcal{C} \cap \mathcal{H} \) is obvious.

Now consider \( x \in \mathcal{C} \cap \mathcal{H} \). Suppose that \( G = \{ \, g^1, \ldots, g^n \} \). Let \( \lambda_1, \ldots, \lambda_p \in \mathbb{R}_{\max}^d \) such that \( x = \sum_{i=1}^p \lambda_i g^i \). Observe that \( ax \leq bx \) implies:

\[
\bigoplus_{ag^i \geq bg^j} \lambda_i (ag^i) + \bigoplus_{ag^i > bg^j} \lambda_j (ag^j) \leq \bigoplus_{ag^i \geq bg^j} \lambda_i (bg^i) + \bigoplus_{ag^i > bg^j} \lambda_j (bg^j).
\]  

(3)

Suppose that \( \bigoplus_{ag^i > bg^j} \lambda_j (bg^j) > \bigoplus_{ag^i \leq bg^j} \lambda_i (bg^j) \). Then there must be a \( k \) such that \( \lambda_k (bg^k) \) is equal to \( \bigoplus_{ag^i > bg^j} \lambda_j (bg^j) \), and necessarily \( \lambda_k > 0 \). But (3) leads to \( \lambda_k (bg^k) \geq \lambda_k (ag^k) \) while \( ag^k > bg^k \), which is a contradiction. It follows that \( \bigoplus_{ag^i > bg^j} \lambda_j (bg^j) \leq \bigoplus_{ag^i \leq bg^j} \lambda_i (bg^j) \), so that, by (3),

\[
\bigoplus_{ag^i > bg^j} \lambda_j (ag^j) \leq \bigoplus_{ag^i \leq bg^j} \lambda_i (bg^i).
\]  

(4)

Let \( \kappa \) be the right member of (4). If \( \kappa > 0 \), then

\[
x = \bigoplus_{ag^i \geq bg^j} \lambda_i g^i + \bigoplus_{ag^i < bg^j} \lambda_j g^j
\
= \bigoplus_{ag^i \geq bg^j} \lambda_i g^i + \kappa^{-1} \bigg[ \bigoplus_{ag^i \leq bg^j} \lambda_j (ag^j) \bigg] \lambda_i g^i
\
+ \kappa^{-1} \bigg[ \bigoplus_{ag^i > bg^j} \lambda_i (bg^i) \bigg] \lambda_j g^j
\
= \bigoplus_{ag^i \geq bg^j} \lambda_i g^i + \kappa^{-1} \bigg[ \bigoplus_{ag^i \leq bg^j} \lambda_i j (ag^j) \bigg] (ag^j) + (bg^i) g^j
\]

which shows \( x \in \text{cone}(G') \). Otherwise, \( \kappa = 0 \). By (4), \( \lambda_j (ag^j) = 0 \) for each \( j \) such that \( ag^j > bg^j \), hence \( \lambda_j = 0 \). It follows that \( x = \bigoplus_{ag^i \leq bg^j} \lambda_i g^i \), thus \( x \in \text{cone}(G') \).

\[\square\]

B Proofs of the Statements in Sect. 4

This section also includes additional material which was only sketched in Sect. 4 for sake of brevity.

Proof (Prop. 1). If there exists \( 1 \leq t \leq d \) such that \( g \) is minimal in \( \{ x \in \mathcal{C} \mid x_t = g_t \} \), then let \( x^1, x^2 \in \mathcal{C} \) such that \( g = x^1 \oplus x^2 \). In that case, for each \( i \in \{ 1, 2 \} \), \( x^i \leq g \), and there is an \( i \) such that \( x^i = g_i \), so that \( x^i = g \).

Conversely, assume that for every index \( t \), \( g \) is not minimal in the set \( \{ x \in \mathcal{C} \mid x_t = g_t \} \), so that we can find a vector \( x^t \) such that \( x^t \leq g, x^t = g_t \), and \( x^t \neq g \). As a result, \( g = x^1 \oplus x^2 \). Since no \( x^t \) is equal to \( g \), this shows that \( g \) cannot be extreme.

\[\square\]

The following proposition shows that extremality can be expressed as a local property.

**Proposition 8.** Given a polyhedral cone \( \mathcal{C} \subset \mathbb{R}_{\max}^d \), \( g \) is extreme of type \( t \) if and only if there exists a neighborhood \( V \) of \( g \) such that \( g \) is a minimal element of the set \( \{ x \in \mathcal{C} \cap V \mid x_t = g_t \} \).

Proof. If such a neighborhood \( V \) exists, let us consider \( x \in \mathcal{C} \) such that \( x \leq g \) and \( x_t = g_t \). Suppose that \( x \) is distinct from \( g \). Then any element of the form \( y = x \oplus \alpha g \) with \( \alpha < 0 \) also satisfies \( y \leq g, y_t = g_t \), and \( y \neq g \). Now, for \( \alpha \) enough close to \( g \), \( y \) belongs to \( V \), which contradicts the extremality of \( g \). \[\square\]
Lemma 1. If $g \neq 0$ is extreme of type $t$ in a cone $C \subset \mathbb{R}^d_{\max}$, then $g_t \neq 0$.

Proof. Suppose that $g_t = 0$, and let $g' = (-1)g$. Then $g' \in C$, $g' \leq g$, and $g' \neq g$ since $g \neq 0$. This is a contradiction with the extremality of type $t$ of $g$. \hfill \Box

Given a subset $I$ of $\{1, \ldots, d\}$, let $\pi_I$ be the function mapping any $x \in \mathbb{R}^d_{\max}$ onto the element consisting of its coordinates $x_i$ for each $i \in I$. Then the following proposition holds:

Proposition 9. Let $C \subset \mathbb{R}^d_{\max}$ be a polyhedral cone, and $g \neq 0$ an element of $C$ such that $\text{supp}(g)$. Then the three following statements are equivalent:

(i) $g$ is extreme in $C$,
(ii) $g$ is extreme in $\{ x \in C \mid \text{supp}(x) \subset S \}$,
(iii) $\pi_S(g)$ is extreme in $\pi_S(\{ x \in C \mid \text{supp}(x) \subset S \})$.

Proof. For sake of simplicity, we assume that $S = \{1, \ldots, p\}$, where $p$ is the cardinality of $S$. Let $D := \{ x \in C \mid \text{supp}(x) \subset S \}$, $E := \pi_S(D)$ and $h := \pi_S(g)$.

(i) $\Rightarrow$ (ii) Supposing $g$ extreme in $C$, $g$ is obviously extreme in $D$ since $g \in D$ and $D \subset C$.

(ii) $\Rightarrow$ (iii) Now, suppose $g$ extreme of type $t$ in the cone $D$. By Lemma 1, $t$ belongs to $S$.

Consider $y \in E$ such that $y \leq h$ and $y_t = h_t$. Let $x \in D$ such that $y = \pi_S(x)$. Then $x \leq g$ and $x_t = y_t = h_t = g_t$. As a consequence, $x = g$, which implies $y = h$.

(iii) $\Rightarrow$ (i) Finally, if $h$ is extreme of type $t$ in $E$, let $x \in C$ such that $x \leq g$ and $x_t = g_t$. Then $\text{supp}(x) \subset S$, hence $y = \pi_S(x)$ satisfies $y \in E$, $y \leq h$ and $y_t = h_t$. This shows $y = h$, so that $x = g$. \hfill \Box

This allows to suppose in Sect. 4 that $\text{supp}(g) = \{1, \ldots, d\}$ without loss of generality.

Proof (Prop. 2). Consider a neighborhood $N$ in which all elements $x$ satisfy the following conditions:

- $A_kx < B_kx$ for all $k$ such that $A_kx < B_kx$,
- $\text{arg max}(A_kx) \subset \text{arg max}(Akx)$ and $\text{arg max}(B_kx) \subset \text{arg max}(Akx)$ for any other $k$.

Let $x \in N$. Note that $x$ belongs to $C$ if and only if, for each $k$ verifying $A_kx = B_kx$,

$$
\max_{i \in \text{arg max}(A_kx)} (a_{ki} + x_i) \leq \max_{j \in \text{arg max}(B_kx)} (b_{kj} + x_j),
$$

by definition of $N$.

Suppose that $x$ belongs to $C$. Let $k$ such that $A_kx = B_kx > 0$. Since for all $i \in \text{arg max}(A_kx)$ and $j \in \text{arg max}(B_kx)$, $a_{ki} + g_i = b_{kj} + g_j > 0$, the term $a_{ki} + g_i$ (resp. $b_{kj} + g_j$) can be substracted from $a_{ki} + x_i$ (resp. $b_{kj} + x_j$) in (5), which shows:

$$
\max_{i \in \text{arg max}(A_kx)} (x_i - g_i) \leq \max_{j \in \text{arg max}(B_kx)} (x_j - g_j).
$$

Now, if $A_kx = B_kx = 0$, then $\text{arg max}(A_kx) = \text{arg max}(B_kx) = \{1, \ldots, d\}$, hence (6) is a tautology.

Conversely, suppose that $x - g$ is an element of $T(g, C)$. Consider $k$ such that $A_kx = B_kx$. Adding the term $a_{ki} + g_i$ (resp. $b_{kj} + g_j$) to each $x_i - g_i$ (resp. $x_j - g_j$) in (6) shows that $x$ satisfies (5).

Proof (Prop. 3). Let $N'$ be the set consisting of the elements $x - g$ for $x \in N$, where $N$ is given by Prop. 2. First remark that $g \in C$ if and only if $1 \in T(g, C)$.

Suppose that $g$ is extreme of type $t$ in the cone $C$, and consider $y \in T(g, C) \cap N'$ such that $y \leq 1$ and $y_t = 1$. Let $x = g + y$, then $x \in C$ by Prop. 2. Besides, $x \leq g$ and $x_t = g_t$,
that \( x = g \) and \( y = 1 \). As \( N' \) is a neighborhood of \( \mathbf{1} \), this proves that \( \mathbf{1} \) is extreme of type \( t \) in \( T(g,C) \) by Prop. 8.

Conversely, suppose that \( \mathbf{1} \) is extreme of type \( t \) in \( T(g,C) \). Let \( x \in C \cap N \) verifying \( x \leq g \) and \( x_t = g_t \). If \( y = x - g \), then by Prop. 2 that \( y \) is an element of \( T(g,C) \). Moreover, \( y \leq 1 \) and \( y_t = 1 \), so that \( y = \mathbf{1} \) and \( x = g \).

\[ \square \]

**Proof (Prop. 4).** If \( \mathbf{1} \) is extreme of type \( t \), then it is minimal in the set \( \{ x \in D \mid x_t = 1 \} \), so that it is also minimal in \( \{ x \in D \cap \{ 0, 1 \}^d \mid x_t = 1 \} \). It follows that the latter set is reduced to \( \{ \mathbf{1} \} \).

Conversely, consider \( x \in D \) such that \( x \leq 1 \) and \( x_t = 1 \). Let \( (x^k)_{k \geq 1} \) be the sequence defined by \( x^k := k \times x_1 \). Since the coefficients of the matrices \( C \) and \( D \) are in the set \( \{ 0, 1 \} \), it is straightforward that each \( x^k \) belongs to \( D \). Moreover, the sequence \( (x^k)_{k \geq 1} \) obviously converges to an element \( x^\infty \in \{ 0, 1 \}^d \), such that \( x^\infty \leq x^k \) for all \( k \geq 1 \). It follows that \( x^\infty \leq 1 \), and \( x^\infty_t = 1 \). Besides, the cone \( D \) is closed, thus \( x^\infty \in D \). This implies \( x^\infty = \mathbf{1} \) by assumption, which leads to \( x = \mathbf{1} \).

\[ \square \]

**Proof (Th. 2).** It is a direct consequence of Prop. 3, and of the application of Prop. 4 on the cone \( T(g,C) \), which is indeed a \( \{ 0, 1 \} \)-cone.

\[ \square \]

**Proposition 10.** Suppose that \( g \in C \). Then \( g \) is extreme of type \( t \) if and only if the following statement holds for all \( l \in \{ 1, \ldots, d \} \):

\[ \forall x \in T(g,C) \cap \{ 0, 1 \}^d, x_l = 0 \implies x_t = 0. \]  

\[ (7) \]

**Proof.** Suppose that \( g \) is extreme of type \( t \). By Th. 2, \( \mathbf{1} \) is the unique element of \( T(g,C) \cap \{ 0, 1 \}^d \) whose \( t \)-th coordinate is 1. Let \( l \in \{ 1, \ldots, d \} \), and \( x \in T(g,C) \cap \{ 0, 1 \}^d \) such that \( x_l = 0 \). Then \( x \neq \mathbf{1} \), so that \( x_t \neq 0 \).

Conversely, since \( g \in C \), then \( \mathbf{1} \in T(g,C) \) by Prop. 2. Besides, consider \( x \in T(g,C) \cap \{ 0, 1 \}^d \) such that \( x_t = 1 \). Then for all \( l \in \{ 1, \ldots, d \} \), \( x_l = 1 \) by Eq. (7), so that \( x = \mathbf{1} \). This proves that \( \mathbf{1} \) is the unique element of \( T(g,C) \cap \{ 0, 1 \}^d \) whose \( t \)-th coordinate is 1, which implies that \( g \) is extreme of type \( t \) by Th. 2.

\[ \square \]

**Proof (Prop. 5).** Suppose that \( t \) is reachable from \( l \) in \( H(g,C) \). Suppose that \( x \in T(g,C) \cap \{ 0, 1 \}^d \) such that \( x_l = 0 \). Let us show by induction on the definition of the reachability in \( H(g,C) \) that \( x_t = 0 \):

- if \( t = l \), then obviously \( x_t = 0 \).
- otherwise, there exists a hyperedge e such that \( t \in H(e) \) and for all \( j \in T(e) \), \( j \) is reachable from \( l \). By induction hypothesis, \( x_j = 0 \). Let \( A_k y \leq B_k y \) be the inequality associated to \( e \). Then \( \arg \max(A_k g) = H(e) \) and \( \arg \max(B_k g) = T(e) \). Since \( x \) satisfies \( \max_{i \in \arg \max(A_k g)} x_i \leq \max_{j \in \arg \max(B_k g)} x_j \), we have \( \max_{i \in H(e)} x_i \leq \max_{j \in T(e)} x_j = 0 \). Hence \( x_t = 0 \), so that Eq. (2) holds.

Conversely, suppose \( t \) is not reachable from \( l \). Let us define \( x = x_i \) by \( x_i = 0 \) if \( i \) is reachable from \( l \), and \( 1 \) otherwise. Let \( 1 \leq k \leq n \) such that \( A_k g = B_k g \). Let \( e \) be the associated hyperedge in \( H(g,C) \). Remember that \( \arg \max(A_k g) = H(e) \) and \( \arg \max(B_k g) = T(e) \). If for all \( j \in T(e) \), \( j \) is reachable from \( l \), then all elements \( i \in H(e) \) are also reachable from \( l \), so that \( \max_{i \in \arg \max(A_k g)} x_i = \max_{j \in \arg \max(B_k g)} x_j = 0 \). Otherwise, there exists \( j \in T(e) \) which is not reachable from \( l \), so that \( \max_{j \in \arg \max(B_k g)} x_j = 1 \). Since \( \max_{i \in \arg \max(A_k g)} x_i \) is always less than or equal to \( 1 \), the inequality \( \max_{i \in \arg \max(A_k g)} x_i \leq \max_{j \in \arg \max(B_k g)} x_j \) holds. It follows that \( T(g,C) \cap \{ 0, 1 \}^d \) admits an element \( x \) such that \( x_l = 0 \) and \( x \neq \mathbf{1} \).

\[ \square \]

**Proof (Th. 3).** From Prop. 5 and 10, \( g \) is extreme of type \( t \) if and only if \( t \) is reachable from any node \( l \) of the hypergraph \( H(g,C) \). This holds if and only if \( t \) belongs to a SCC \( C \) such that \( C \leq H(g,C) \) for any SCC \( D \). As a consequence, \( g \) is extreme if and only if a least SCC exists.

\[ \square \]
C Proofs of Prop. 6 and 7

The reachability relation in a digraph $G$ is denoted by $\sim_G$. We first introduce a lemma:

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

**Proof.** Formally, consider $u \in C$. Then there exists a SCC $C'$ of $G(\mathcal{H})$ such that $u \in C$ (since the SCCs of $G(\mathcal{H})$ form a partition of the set $N$), and obviously $C \cap C' \neq \emptyset$.

Conversely, suppose that $C'$ is a SCC of $G(\mathcal{H})$ such that $C \cap C' \neq \emptyset$. Let $u \in C \cap C'$. Then for any $v \in C'$, $u \sim_{G(\mathcal{H})} v \sim_{G(\mathcal{H})} u$, so that $u \sim_{\mathcal{H}} v \sim_{\mathcal{H}} u$, hence $v \in C$. □

**Proof (Prop. 6).** First suppose that $\{ u \}$ is a minimal SCC of $G(\mathcal{H})$. Suppose that there exists $v \neq u$ such that $u \sim_{\mathcal{H}} v$. Consider a hyperpath $e_1, \ldots, e_p$ from $u$ to $v$ in $\mathcal{H}$. Then there must be a hyperedge $e_i$ such that $T(e_i) = \{ u \}$ and $H(e_i) \neq \{ u \}$ (otherwise, the hyperpath is a cycle and $v = u$). Let $w \in H(e_i) \setminus \{ u \}$. Then $(u, w)$ is an edge of $G(\mathcal{H})$. Since $\{ u \}$ is a minimal SCC of $G(\mathcal{H})$, this enforces $w = u$, which is a contradiction. Hence $\{ u \}$ is a minimal SCC of $\mathcal{H}$.

Conversely, consider a minimal SCC $C$ of $\mathcal{H}$. Let $u \in C$, and let $D$ be the SCC of $G(\mathcal{H})$ containing $u$. Consider $D'$ a minimal SCC of $G(\mathcal{H})$ such that $D' \preceq_{G(\mathcal{H})} D$, and let $C'$ be a SCC of $\mathcal{H}$ such that $D' \cap C' \neq \emptyset$. By Lemma 2, we have $D' \subset C'$. It follows that $C' \preceq_{\mathcal{H}} C$, hence $C = C'$ by minimality of $C$. Thus, $D' \subset C$, 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 nodes), so that $C$ is a minimal SCC of $G(\mathcal{H})$. □

**Proof (Prop. 7).** Let $\mathcal{H}' = f(\mathcal{H})$. Suppose that $s \sim_{\mathcal{H}} t$. Observe that if $X, Y$ are subsets of $N$, $f(X) \subset f(Y)$ as soon as $X \subset Y$, and $f(X \cup Y) \subset f(X) \cup f(Y)$. Therefore, if $e_1, \ldots, e_p$ is a hyperpath from $s$ to $t$, then:

$$T(e_i) \subset \{ s \} \cup H(e_1) \cup \cdots \cup H(e_{i-1})$$

$$t \in H(e_p)$$

so that:

$$f(T(e_i)) \subset \{ f(s) \} \cup f(H(e_1)) \cup \cdots \cup f(H(e_{i-1}))$$

$$f(t) \in f(H(e_p))$$

It follows that $f(s) \sim_{\mathcal{H}'} f(t)$.

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 $e_1 = (T_1, H_1), \ldots, e_p = (T_p, H_p)$ in $E$ such that for each $i \in \{ 1, \ldots, p+1 \}$, $f(T_i) \subset f(H_0) \cup \cdots \cup f(H_{i-1})$.

Also note that for any $s \in \wp(N)$, $f(s) = s$ in $s \cap \{ x, y \} = \emptyset$ and $f(s) = s \cup \{ xy \} \setminus \{ x, y \}$ otherwise. In particular, as soon as $xy \notin f(s)$, $f(s)$ coincides with $s$. Besides, $f(s) \setminus \{ xy \} \subset s \subset f(s) \setminus \{ xy \} \cup \{ x, y \}$.

Two cases can be distinguished:

(a) suppose that $xy$ 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 $xy$, hence $f(T_i) = T_i$. Besides, $T_i \subset H_0 \cup \cdots \cup H_{i-1}$ for each $i$, so that is is straightforward that $f(s) \sim_{\mathcal{H}'} f(t)$.

(b) now, if $xy$ in one of the $f(H_j)$, let $k$ be the smallest integer such that $xy \in f(H_k)$. 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 $\mathcal{H}$.

When $i \leq k$, $f(T_i)$ does not contain $xy$, hence $f(T_i) = T_i$ and $T_i \subset f(H_0) \cup \cdots \cup f(H_{i-1}) = H_0 \cup \cdots \cup H_{i-1}$.
The correctness proof of the algorithm MinSccCount turns out to be harder than the one of the classical Tarjan’s algorithm, 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 MinSccCount2 (Fig. 11) and MinSccCount3 (Fig. 12), and then to prove that they are equivalent to MinSccCount.

The main difference between the first intermediary form and MinSccCount is that it does not use auxiliary data associated to the hyperedges to determine which ones are added to the digraph $G(\mathcal{H}_{cur})$ after a node merging step. Instead, the stack $F$ is directly filled with the right

**D Proof of Th. 4**

Theorem 4 contains two statements, a first one relative to the correctness of MinSccCount (i.e. it precisely computes the minimal SCCs), and a second one to its time complexity. We first focus on the first part.

**D.1 Correctness of the Algorithm**

The correctness proof of the algorithm MinSccCount turns out to be harder than the one of the classical Tarjan’s algorithm, 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 MinSccCount2 (Fig. 11) and MinSccCount3 (Fig. 12), and then to prove that they are equivalent to MinSccCount.

The main difference between the first intermediary form and MinSccCount is that it does not use auxiliary data associated to the hyperedges to determine which ones are added to the digraph $G(\mathcal{H}_{cur})$ after a node merging step. Instead, the stack $F$ is directly filled with the right
hyperedges (Lines 18 and 43). Besides, a boolean no_merge is used to determine whether a node merging step has been executed. The notion of node merging step is refined: it now refers to the execution of the instructions between Lines 35 and 44 in which the boolean no_merge is set to false.

For 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 14 and 19, or from Lines 35 and 44, and at from Lines 52 to 54, 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 node of the current hypergraph $H_{\text{cur}}$. Then index[$U$] is defined if and only if index[$u$] is defined for all $u \in N$ such that FIND($u$) = $U$.

**Proof.** It can be shown by induction on the number of node merging steps which has been performed on $U$.

In the basis case, there is a unique element $u \in N$ such that FIND($u$) = $U$. Besides, $U = u$, so that the statement is trivial.

After a node merging step yielding the node $U$, we necessarily have index[$U$] $\neq$ undef. Moreover, all the nodes $V$ which has been merged into $U$ satisfied index[$V$] $\neq$ undef because they were stored in the stack $S$. Applying the induction hypothesis terminates the proof.

**Invariant 2.** Let $u \in N$. When index[$u$] is defined, then FIND($u$) belongs either to the stack $S$, or to the set Finished (both cases can not happen simultaneously).

**Proof.** Initially, FIND($u$) = $u$, and once index[$u$] is defined, FIND($u$) is pushed on $S$ (Line 16). Naturally, $u \notin$ Finished, because otherwise, index[$u$] would have been defined before (see the condition Line 54). After that, $U =$ FIND($u$) can be popped from $S$ at three possible locations:

- at Lines 36 or 40, in which case $U$ is transformed into a node $U'$ which is immediately pushed on the stack $S$ at Line 42. Since after that, FIND($u$) = $U'$, the property FIND($u$) $\in$ $S$ still holds.
- at Line 53, in which case it is directly appended to the set Finished.

**Invariant 3.** The set Finished is always growing.

**Proof.** Once an element is added to Finished, it is never removed from it nor merged into another node (the function MERGE is always called on elements immediately popped from the stack $S$).

**Proposition 11.** The function MINSCCCount2($H$) returns the number of minimal Scc of $H$. Besides, the minimal Sccs are formed by the sets \{ $v \in N$ | FIND($v$) = $U$ and ismin[$U$] = true \}.

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

**Basis Case.** First, suppose that the hypergraph $H$ is such that no nodes are merged during the execution of MINSCCCOUNT2($H$), i.e. the node merging loop (from Lines 37 to 41) is never executed. Then the boolean no_merge is always set to true, so that $n$ is never redefined to $i + 1$ (Line 46), and there is no back edge to Line 20 in the control-flow graph. It follows that removing all the lines between Lines 35 to 47 does not change the behavior of the algorithm. Besides, since the function MERGE is never called, FIND($u$) always coincides with $u$. Finally, at Line 18, $F$ is precisely assigned to the set of simple hyperedges leaving $u$ in $H$, so that the loop from Lines 20 to 32 iterates on the successors of $u$ in $G(H)$. As a consequence, the algorithm MINSCCCOUNT2($H$) behaves exactly like MINSCCCOUNT($G(H)$). Moreover, under our assumption, the minimal Sccs of $G(H)$ are all reduced to singletons (otherwise, the loop from Lines 37 to 41 would be executed, and some nodes would be merged). Therefore, by Prop. 6, the statement in Prop. 11 holds.
Inductive Case. Now suppose that the node merging loop is executed at least once, and that its first execution happens during the execution of, say, $\text{HVisit2}(x)$. Consider the state of the algorithm at Line 35 just before the execution of the first occurrence of the node merging step. Until that point, $\text{Find}(v)$ is still equal to $v$ for all node $v \in N$, so that the execution of $\text{MinSccCount}(\mathcal{H})$ coincides with the execution of $\text{MinSccCount}(G(\mathcal{H}))$. Consequently, if $C$ is the set formed by the nodes $y$ located above $x$ in the stack $S$ (including $x$), $C$ forms a minimal Scc of $G(\mathcal{H})$. 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 $(N,E \setminus \{ e \mid \exists y \in C \text{ s.t. } T(e) = \{ y \} \})$, and let $X$ be the resulting node. For now, we may add a hypergraph as last argument of the functions $\text{HVisit2}$, $\text{Find}$, ... to distinguish their execution in the context of the call to $\text{MinSccCount2}(\mathcal{H})$ or $\text{MinSccCount2}(\mathcal{H}')$. We make the following observations:

- the node $x$ is the first element of $C$ to be visited during the execution of $\text{MinSccCount2}(\mathcal{H})$.
  It follows that the execution of $\text{MinSccCount2}(\mathcal{H})$ until the call to $\text{HVisit2}(x, \mathcal{H})$ coincides with the execution of $\text{MinSccCount2}(\mathcal{H}')$ until the call to $\text{HVisit2}(X, \mathcal{H}')$.
- besides, during the execution of $\text{HVisit2}(x, \mathcal{H})$, the execution of the loop from Lines 20 to 32 only has a local impact, i.e. on the $\text{ismin}[y]$, $\text{index}[y]$, or $\text{low}[y]$ for $y \in C$, and not on $\text{nb}$ or any information relative to other nodes. Indeed, we claim that the set of the nodes $y$ on which $\text{HVisit2}$ is called during the execution of the loop is exactly $C \setminus \{ x \}$. First, for all $y \in C \setminus \{ x \}$, $\text{HVisit2}(y)$ has necessarily been executed after Line 20 (otherwise, by Inv. 2, $y$ would be either below $x$ in the stack $S$, or in $\text{Finished}$). Conversely, suppose that after Line 20, there is a call to $\text{HVisit2}(t)$ with $t \notin C$. By Inv. 2, $t$ belongs to $\text{Finished}$, so that for one of the nodes $w$ examined in the loop, either $w \in \text{Finished}$ or $\text{ismin}[w] = \text{false}$ after the call to $\text{HVisit2}(w)$. Hence $\text{ismin}[x]$ should be $\text{false}$, which contradicts our assumptions.
- finally, from the execution of Line 47 during the call to $\text{HVisit2}(x, \mathcal{H})$, our algorithm behaves exactly as $\text{MinSccCount2}(\mathcal{H}')$ from the execution of Line 20 in $\text{HVisit2}(X, \mathcal{H}')$. 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 $\text{ismin}[y] = \text{true}$. The node $X$ being equal to one of the $y \in C$, we also have $\text{low}[X] = i$ and $\text{ismin}[X] = \text{true}$. Moreover, $X$ is the top element of $S$.

Furthermore, it can be verified that at Line 43, the set $F$ contains exactly all the hyperedges of $E$ which generate the simple hyperedges leaving $X$ in $\mathcal{H}'$: they are exactly characterized by

$\text{Find}(z, \mathcal{H}) = X \text{ for all } z \in T(e), \text{ and } T(e) \neq \{ y \} \text{ for all } y \in C$

$\iff \text{Find}(z, \mathcal{H}) = X \text{ for all } z \in T(e), \text{ and } \text{collected}_e = \text{false}$

since at that Line 43, a hyperedge $e$ satisfies $\text{collected}_{e} = \text{true}$ if and only if $T(e)$ is reduced to a singleton $\{ t \}$ such that $\text{index}[t]$ is defined.

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

As a consequence, $\text{MinSccCount2}(\mathcal{H})$ returns the same result as $\text{MinSccCount2}(\mathcal{H}')$. Besides, both functions perform the same union-find operations, except the first the node merging step executed by $\text{MinSccCount2}(\mathcal{H})$ on $C$.

Let $f$ be the function which maps all nodes $y \in C$ to $X$, and any other node 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 hyperedges $e \in E$ such that $T(e) = \{ y \}$ for some $y \in C$. For such hyperedges, we have $H(e) \subset C$, because otherwise, $\text{ismin}[x]$ would have been set to $\text{false}$ (i.e. the Scc $C$ would not be minimal). 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 minimal Sccs.
Finally, since the elements of \( C \) are in a same SCC of \( \mathcal{H} \), Prop. 7 shows that the function \( f \) induces a one-to-one correspondence between the SCCs of \( \mathcal{H} \) and the SCCs of \( f(\mathcal{H}) \).

\[
D \leftrightarrow f(D)
\]

\[
(D' \setminus \{ X \}) \cup C \leftrightarrow D'
\]

if \( X \in D' \)

\[
D' \longrightarrow D'
\]

otherwise.

The action of the function \( f \) exactly corresponds to the node merging step performed on \( C \). Since by induction hypothesis, \( \text{MINSCCCount2}(\mathcal{H'}) \) determines the minimal SCCs in \( f(\mathcal{H}) \), it follows that Prop. 11 holds.

The second intermediary version of our algorithm, \( \text{MINSCCCount3} \), is based on the first one, but it performs the same computations on the auxiliary data \( r_e \) and \( c_e \) as in \( \text{MINSCCCount} \). However, the latter are never used, because at Line 58, \( F \) is re-assigned to the value provided in \( \text{MINSCCCount2} \). It follows that for now, the parts in gray can be ignored. The following lemma states that \( \text{MINSCCCount2} \) and \( \text{MINSCCCount3} \) are equivalent:

**Proposition 12.** Let \( \mathcal{H} \) be a hypergraph. Then \( \text{MINSCCCount3}(\mathcal{H}) \) returns the number of minimal SCC of \( \mathcal{H} \). Besides, the minimal SCCs are formed by the sets \( \{ v \in N \mid \text{FIND}(v) = U \text{ and ismin}[U] = true \} \).

**Proof.** When \( \text{HVISIT3}(u) \) is executed, the local stack \( F \) is not directly assigned to the set \( \{ e \in E \mid T(e) = \{ u \} \} \) (see Line 18 in Fig. 11), but built by several iterations on the set \( E_u \) (Line 21). Since \( u \in T(e) \) and \( |T(e)| = 1 \) holds if and only if \( T(e) \) is reduced to \( \{ u \} \), \( \text{HVISIT3}(u) \) initially fills \( F \) with the same hyperedges as \( \text{HVISIT2}(u) \).
Besides, the condition \( \text{no\_merge} = \text{false} \) in \text{HVisit2} (Line 45) is replaced by \( F \neq \emptyset \) (Line 60). We claim that the condition \( F \neq \emptyset \) can be safely used in \text{HVisit2} as well. Indeed, in \text{HVisit2}, \( F \neq \emptyset \) implies \( \text{no\_merge} = \text{false} \). Conversely, suppose that in \text{HVisit2}, \( \text{no\_merge} = \text{false} \) and \( F = \emptyset \), so that the algorithm goes back to Line 47 after having \( \text{no\_merge} \) to \text{true}. The loop from Lines 20 to 32 is not executed since \( F = \emptyset \), and it directly leads to a new execution of Lines 33 to 45 with \( \text{no\_merge} = \text{true} \). Therefore, going back to Line 47 was useless.

Finally, during the node merging step in \text{HVisit3}, \( n \) keeps its value, which is greater than or equal to \( i + 1 \), but is not necessarily equal to \( i + 1 \) like in \text{HVisit2} (just after Line 46). This is safe because the whole algorithm only need that \( n \) take increasing values, and not necessarily consecutive ones.

We conclude by applying Prop. 11. \( \square \)

We make similar assumptions on the atomicity of the sequences of instructions. Note that Inv. 1, 2, and 3 still holds in \text{HVisit3}.

**Invariant 4.** Let \( e \in E \) such that \( |T(e)| > 1 \). If for all \( x \in T(e) \), \( \text{index}[x] \) is defined, then the root \( r_e \) is defined.

**Proof.** For all \( x \in T(e) \), \text{HVisit3}(x) \) has been called. The root \( r_e \) has necessarily been defined at the first of these calls (remember that the block from Lines 17 to 33 is supposed to be executed atomically). \( \square \)

**Invariant 5.** Consider a state \( \text{cur} \) of the algorithm in which \( U \in \text{Finished} \). Then any node reachable from \( U \) in \( G(\mathcal{H}_{\text{cur}}) \) is also in \( \text{Finished} \).

**Proof.** The invariant clearly holds when \( U \) is placed in \( \text{Finished} \). Using the atomicity assumptions, the call to \text{HVisit3}(u) \) is necessarily terminated. Let \( \text{old} \) be the state of the algorithm at that point, and \( \mathcal{H}_{\text{old}} \) and \( \text{Finished}_{\text{old}} \) the corresponding hypergraph and set of terminated nodes at that state respectively. Since \text{HVisit3}(u) \) has performed a depth-first search from the node \( U \) in \( G(\mathcal{H}_{\text{old}}) \), all the nodes reachable from \( U \) in \( \mathcal{H}_{\text{old}} \) stand in \( \text{Finished}_{\text{old}} \).

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

**Invariant 6.** In the digraph \( G(\mathcal{H}_{\text{cur}}) \), at the call to \text{HVisit3}(u), \( u \) is reachable from a node \( W \) such that \( \text{index}[W] \) is defined if and only if \( W \) belongs to the stack \( S \).

**Proof.** The “if” part can be shown by induction. When the function \text{HVisit3}(u) \) is called from Line 12, the stack \( S \) is empty, so that this is obvious. Otherwise, it is called from Line 38 during the execution of \text{HVisit3}(x) \). Then \( X = \text{Find}(x) \) is reachable from any node in the stack, since \( x \) was itself reachable from any node in the stack at the call to \( \text{Find}(X) \) (inductive hypothesis) and that this reachability property is preserved by potential node merging steps (Prop. 7). 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 Inv. 2, \( W \) is necessarily an element of \( \text{Finished} \). Hence \( u \) also belongs to \( \text{Finished} \) by Inv. 5, which is a contradiction since this can not hold at the call to \text{HVisit}(u). \( \square \)

**Invariant 7.** Let \( e \in E \) such that \( |T(e)| > 1 \). Consider a state \( \text{cur} \) of the algorithm \text{MinScc-Count3} in which \( r_e \) is defined.

Then \( c_e \) is equal to the number of elements \( x \in T(e) \) such that \( \text{index}[x] \) is defined and \( \text{Find}(x) \) is reachable from \( \text{Find}(r_e) \) in \( G(\mathcal{H}_{\text{cur}}) \).
Proof. Since at Line 26, \(c_e\) is incremented only if \(R_e = \text{FIND}(r_e)\) belongs to \(S\), we already know using Inv. 6 that \(c_e\) is equal to the number of elements \(x \in T(e)\) such that, at the call to HVISIT3\(x\), \(x\) was reachable from \(\text{FIND}(r_e)\).

Now, let \(x \in N\), and consider a state \(\text{cur}\) of the algorithm in which \(r_e\) and \(\text{index}[x]\) are both defined, and \(\text{FIND}(r_e)\) appears in the stack \(S\). Since \(\text{index}[x]\) is defined, HVISIT3 has been called on \(x\), and let \(\text{old}\) be the state of the algorithm at that point. Let us denote by \(H_{\text{old}}\) and \(H_{\text{cur}}\) 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 \(\text{FIND}\) to distinguish its execution in the states \(\text{old}\) and \(\text{cur}\). We claim that \(\text{FIND}(r_e, H_{\text{cur}}) \sim_{G(H_{\text{cur}})} \text{FIND}(x, H_{\text{cur}})\) if and only if \(\text{FIND}(r_e, H_{\text{old}}) \sim_{G(H_{\text{old}})} x\). The “if” part is due to the fact that reachability in \(G(H_{\text{old}})\) is not altered by the node merging steps (Prop. 7). Conversely, if \(x\) is not reachable from \(\text{FIND}(r_e, H_{\text{old}})\) in \(H_{\text{old}}\), then \(\text{FIND}(r_e, H_{\text{old}})\) is not in the call stack \(S_{\text{old}}\) (Invariant 6), so that it is an element of \(\text{Finished}_{\text{old}}\). But \(\text{Finished}_{\text{old}} \subset \text{Finished}_{\text{cur}}\), which contradicts our assumption since by Inv. 2, an element can not be stored in \(\text{Finished}_{\text{cur}}\) and \(S_{\text{cur}}\) at the same time. It follows that if \(r_e\) is defined and \(\text{FIND}(r_e)\) appears in the stack \(S\), \(c_e\) is equal to the number of elements \(x \in T(e)\) such that \(\text{index}[x]\) is defined and \(\text{FIND}(r_e) \sim_{G(H_{\text{cur}})} \text{FIND}(x)\).

Let \(\text{cur}\) be the state of the algorithm when \(\text{FIND}(r_e)\) is moved from \(S\) to \(\text{Finished}\). The invariant still holds. Besides, in the future states \(\text{new}\), \(c_e\) is not incremented because \(\text{FIND}(r_e, H_{\text{cur}}) \in \text{Finished}_{\text{cur}} \subset \text{Finished}_{\text{new}}\) (Inv. 3), so that \(\text{FIND}(r_e, H_{\text{new}}) = \text{FIND}(r_e, H_{\text{cur}})\), and the latter can not appear in the stack \(S_{\text{new}}\) (Inv. 2). Furthermore, any node reachable from \(r_e = \text{FIND}(r_e, H_{\text{new}})\) in \(G(H_{\text{new}})\) belongs to \(\text{Finished}_{\text{cur}}\), as shown in the second part of the proof of Inv. 5 (emphasized sentence). It follows that the number of reachable nodes from \(\text{FIND}(r_e)\) has not changed between states \(\text{cur}\) and \(\text{new}\). Therefore, the invariant on \(c_e\) will be preserved, which completes the proof.

\[\Box\]

**Proposition 13.** In HVISIT3, the assignment at Line 58 does not change the value of \(F\).

**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 49, and let \(X_1, \ldots, X_q\) be the elements of the stack located above the root \(U = X_1\) of the minimal SCC of \(G(H_{\text{cur}})\). Any edge \(e\) which will transferred to \(F\) from Line 49 to Line 56 satisfies \(c_e = \lfloor T(e) \rfloor > 1\) and \(\text{FIND}(r_e) = X_i\) for some \(1 \leq i \leq q\) (since at 49, \(F\) is initially empty). Invariant 7 implies that for all elements \(x \in T(e)\), \(\text{FIND}(x)\) is reachable from \(X_i\) in \(G(H_{\text{cur}})\), so that by minimality 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 56, \(\text{FIND}(x) = U\) for all \(x \in T(e)\). Then, we claim that \(\text{collected}_e = \text{false}\) at Line 56. Indeed, \(e' \in E\) satisfies \(\text{collected}_e = \text{true}\) if and only

- either it has been copied to \(F\) at Line 21, in which case \(|T(e')| = 1\),
- or if it has been copied to \(F\) at the \(r\)-th execution of Line 58, with \(r < p\). By induction hypothesis, this means that \(e'\) has been pushed on a stack \(F_X\) and then popped from it strictly before the \(r\)-th execution of Line 58.

Observe that a given hyperedge can be popped from a stack \(F_x\) at most once during the whole execution of MinSccCount3. Here, \(e\) has been popped from \(F_X\), after the \(p\)-th execution of Line 58, and \(|T(e)| > 1\). It follows that \(\text{collected}_e = \text{false}\).

Conversely, suppose for that, at Line 58, \(\text{collected}_e = \text{false}\), and all the \(x \in T(e)\) satisfies \(\text{FIND}(x) = U\). Clearly, \(|T(e)| > 1\) (otherwise, \(e\) would have been placed into \(F\) at Line 21 and \(\text{collected}_e\) would be equal to \(true\)). Few steps before, at Line 49, \(\text{FIND}(x)\) 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 Inv. 1, \(\text{index}[x]\) is also defined for all \(x \in T(e)\), hence, the root \(r_e\) is defined by Inv. 4. Besides, \(\text{FIND}(r_e)\) is equal to one of the \(X_j\), say \(X_k\) (since \(r_e \in T(e)\)). As all the \(\text{FIND}(x)\) are reachable from \(\text{FIND}(r_e)\) in \(G(H_{\text{cur}})\), then \(c_e = |T(e)|\) using Invariant 7. It follows that \(e\) has been pushed on the stack \(F_{R_e}\), where \(R_e = \text{FIND}(r_e, H_{\text{old}})\) in an previous state \(\text{old}\) of the algorithm. As \(\text{collected}_e = \text{false}\),
$e$ has not been popped from $F_{R_e}$, and consequently, the node $R_e$ of $H_{\text{old}}$ has not involved in a node merging step. Therefore, $R_e$ is still equal to $\text{Find}((r_e, H_{\text{cur}})) = X_k$. It follows that at Line 49, $e$ is stored in $F_{X_k}$, and thus it is copied to $F$ between Lines 49 and 56. This completes the proof.

We now can prove the correctness of $\text{MinSccCount}$. By Proposition 13, Line 58 can be safely removed in $\text{HVisit3}$. It follows that the booleans collected$_e$ are now useless, to that Lines 5, 33, and 59 can be also removed. After that, we precisely obtain the algorithm $\text{MinSccCount}$. Proposition 12 completes the proof.

\[\square\]

D.2 Complexity Proof

Then analysis of the time complexity $\text{MinSccCount}$ 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 $\text{HVisit}(u)$ is executed exactly once for each $u \in N$ during the execution of $\text{MinSccCount}$.

Operations on the Stacks $F$ and $F_u$. Each operation on the stack (pop or push) is in $O(1)$. A given hyperedge is pushed on a stack of the form $F_u$ at most once during the whole execution of $\text{MinSccCount}$. Once it is popped from it, it will never be pushed on a stack of the form $F_V$ again. Similarly, a hyperedge 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 to by $4 |N|$. It follows that the corresponding complexity is $O(|N|)$.

As a consequence, the total of the number of iterations of the loop from Lines 53 to 62 which occur during the whole execution of $\text{MinSccCount}$ is bounded to $\sum_{e \in E} |H(e)|$.

Union-find Operations. During the whole execution of $\text{MinSccCount}$, the function $\text{Find}$ is called:

- exactly $|N|$ times at Line 35,
- at most $\sum_{u \in N} |E_u| = \sum_{e \in E} |T(e)|$ times at Line 42 (since during the call to $\text{HVisit}(u)$, the loop from Lines 38 to 50 has exactly $|E_u|$ iterations),
- at most $\sum_{e \in E} |H(e)|$ at Line 54 (see above).

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

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

Finally, $\text{MakeSet}$ is called exactly $|N|$ times. It follows that the total time complexity of the operations $\text{MakeSet}$, $\text{Find}$ and $\text{Merge}$ is $O(\text{size}(\mathcal{H}) \times \alpha(|N|))$.

Usual Operations. The analysis of the usual complexity is split into several parts:

- the usual complexity $\text{MinSccCount}$ without the calls to the function $\text{HVisit}$ is clearly $O(|N| + |E|)$. 

– during the execution of HVisit(u), the usual complexity of the block between Lines 35 and 50 is \( O(1) + O(|E_u|) \). Indeed, we suppose that the test at Line 43 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 \( N \), whether it is stored in \( S \). Then the total usual complexity between Lines 35 and 50 is \( O(\text{size}(\mathcal{H})) \) for a whole execution of MINSCCCount.

– the usual complexity of the body of loop from Lines 53 to 62, without the recursive calls to HVisit, is clearly \( O(1) \). As mentioned above, the total number of iterations of this loop is less than \( \sum_{e \in E} |H(e)| \leq \text{size}(\mathcal{H}) \). Therefore, the total usual complexity of the loop from Lines 51 to 63 is in \( O(\text{size}(\mathcal{H})) \).

– the usual complexity of the loop between Lines 69 and 73 for a whole execution of MINSCCCount is \( O(|N|) \), since in total, it is iterated exactly the number of times the function Merge is called.

– the usual complexity of the loop between Lines 78 and 80 for a whole execution of MINSCCCount is \( O(|N|) \), 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 64 to 82, all of complexity \( O(1) \). The execution of this block either follows a call to HVisit or the execution of the goto statement (at Line 75). The latter is executed only if the stack \( F \) is not empty. Since each hyperedge can be pushed on a local stack \( F \) and then popped from it only once, it is executed \( |E| \) in the worst case during the whole execution of MINSCCCount. It follows that the usual complexity of the block from Lines 64 to 82 is \( O(|N| + |E|) \) in total (excluding the loops previously discussed).

### Total Time Complexity

Summing all the complexities above proves that the time complexity of MINSCCCount is \( O(\text{size}(\mathcal{H}) \times \alpha(|N|)) \). \( \square \)

### E An Example of an Execution Trace of the Algorithm MINSCCCount

In this section, we give the main steps of the execution of MINSCCCount on the hypergraph depicted in Fig. 8:

Nodes are depicted by solid circles if their index is defined, and by dashed circles otherwise. Once a node is placed into Finished, it is depicted in gray. Similarly, a hyperedge 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 sake of readability, gray hyperedges mapped to cycles after a node 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 Visit.

Initially, Find(z) = z for all \( z \in \{ u, v, w, x, y, t \} \). We suppose that HVisit(u) is called first.

After the execution of the block from Lines 35 to 50, the current state is:

\[
\begin{align*}
U &= u \\
\text{index}[u] &= 0 \\
\text{low}[u] &= 0 \\
\text{ismin}[u] &= \text{true} \\
S &= [u] \\
n &= 1 \\
F &= [e_1] \\
nb &= 0
\end{align*}
\]

\( ^4 \) Obviously, the push and pop operations on the stack \( S \) are still in \( O(1) \) under this assumption.
Following the hyperedge $e_1$, $HVisit(v)$ is called during the execution of the block from Lines 51 to 63 of $HVisit(u)$. After Line 50 in $HVisit(v)$, the root of the hyperedge $e_4$ is set to $v$, and the counter $c_{e_4}$ is incremented to 1 since $v \in S$. The state is:

$$
\begin{aligned}
index[u] &= 0 \\
low[u] &= 0 \\
ismin[u] &= true \\
\end{aligned}
$$

\[ S = [v; u] \]
\[ n = 2 \]
\[ F = [e_2] \]
\[ nb = 0 \]

Similarly, $HVisit(w)$ is called during the execution of the loop from Lines 51 to 63 of $HVisit(v)$. After Line 50 in $HVisit(w)$, the root of the hyperedge $e_5$ is set to $w$, and the counter $c_{e_5}$ is incremented to 1 since $w \in S$. Besides, the $c_{e_4}$ is incremented to $2 = |T(e_4)|$ since $\text{Find}(re_4) = \text{Find}(v) = v \in S$, so that $e_4$ is pushed on the stack $F_v$. The state is:

$$
\begin{aligned}
index[v] &= 1 \\
low[v] &= 1 \\
ismin[v] &= true \\
\end{aligned}
$$

\[ S = [w; v; u] \]
\[ n = 3 \]
\[ F = [e_3] \]
\[ F_v = [e_4] \]
\[ nb = 0 \]

The execution of the loop from Lines 51 to 63 of $HVisit(w)$ 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 $ismin[w]$ to $ismin[w] \& \& ismin[u] = true$. At the end of the loop, the state is therefore:

$$
\begin{aligned}
index[u] &= 0 \\
low[u] &= 0 \\
ismin[u] &= true \\
\end{aligned}
$$

\[ S = [w; v; u] \]
\[ n = 3 \]
\[ F = [] \]
\[ F_v = [e_4] \]
\[ nb = 0 \]

Since $low[w] \neq index[w]$, the block from Lines 64 to 82 is not executed, and $HVisit(w)$ terminates. Back to the loop from Lines 51 to 63 in $HVisit(v)$, $low[v]$ is set to $\min(low[v], low[w]) = 0$, and $ismin[v]$ to $ismin[v] \& \& ismin[w] = true$: 

$$
\begin{aligned}
index[v] &= 1 \\
low[v] &= 1 \\
ismin[v] &= true \\
\end{aligned}
$$

\[ S = [w; v; u] \]
\[ n = 3 \]
\[ F = [] \]
\[ F_v = [e_4] \]
\[ nb = 0 \]
Since \( \text{low}[v] \neq \text{index}[v] \), the block from Lines 64 to 82 is not executed, and \( \text{HVisit}(v) \) terminates. Back to the loop from Lines 51 to 63 in \( \text{HVisit}(u) \), \( \text{low}[u] \) is set to \( \min(\text{low}[u], \text{low}[v]) = 0 \), and \( \text{ismin}[u] \) to \( \text{ismin}[u] \land \text{ismin}[v] = \text{true} \). Therefore, at Line 64, the conditions \( \text{low}[u] = \text{index}[u] \) and \( \text{ismin}[u] = \text{true} \) hold, so that a node merging step is executed. At that point, the stack \( F \) is empty. After that, \( i \) is set to \( \text{index}[u] = 0 \) (Line 66), and \( F_u = [\ ] \) is emptied to \( F \) (Line 67), so that \( F \) is still empty. Then \( w \) is popped from \( S \), and since \( \text{index}[w] = 2 > i = 0 \), the loop from Lines 69 to 73 is iterated again. Then the stack \( F_w = [\ ] \) is emptied in \( F \). At Line 71, \( \text{MERGE}(u, w) \) is called. The result is denoted by \( U \) (in practice, either \( U = u \) or \( U = w \)). The state is:

Then \( v \) is popped from \( S \), and since \( \text{index}[v] = 1 > i = 0 \), the loop Lines 69 to 73 is iterated again. Then the stack \( F_u = [e_4] \) is emptied in \( F \). At Line 71, \( \text{MERGE}(U, v) \) is called. The result is set to \( U \) (in practice, \( U \) is one of the nodes \( u, v, w \)). The state is:

After that, \( u \) is popped from \( S \), and as \( \text{index}[u] = 0 = i \), the loop is terminated. At Line 74, \( \text{index}[U] \) is set to \( i \), and \( U \) is pushed on \( S \). Since \( F \neq \emptyset \), we go back to Line 51, in the state:

Then \( e_4 \) is popped from \( F \), and the loop from 53 to 62 iterates over \( H(e_4) = \{ x, y \} \). Suppose that \( x \) is treated first. Then \( \text{VISIT}(x) \) is called. During its execution, at Line 50, the state is:
Since $F$ is empty, the loop from Lines 51 to 63 is not executed. At Line 64, $\text{low}[x] = \text{index}[x]$ and $\text{ismin}[x] = \text{true}$, so that a trivial node merging step is performed, only on $x$, since it is the top element of $S$. At Line 74, it can be verified that $S = [x; U]$, $\text{index}[x] = 3$ and $F = \emptyset$. Therefore, the goto statement at Line 75 is not executed, and $\text{nb}$ is incremented at Line 76. It follows that the loop from Lines 78 to 80 is executed, and after that, the state is:

\[
\begin{align*}
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{ismin}[U] &= \text{true} \\
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{ismin}[x] &= \text{true} \\
S &= [x; U] \\
n &= 4 \\
F &= \emptyset \\
U &= \text{Find}(u) = \text{Find}(v) = \text{Find}(w) \\
\text{nb} &= 0 \\
\text{Finished} &= \{x\}
\end{align*}
\]

After the termination of $\text{HVisit}(x)$, since $x \in \text{Finished}$, $\text{ismin}[U]$ is set to $\text{false}$. After that, $\text{HVisit}(y)$ is called, and at Line 50, it can be checked that $c_{e_5}$ has been incremented to $2 = |T(e_5)|$ because $R_{e_5} = \text{Find}(r_{e_5}) = \text{Find}(w) = U$ and $U \in S$. Therefore, $e_5$ is pushed to $F_U$, and the state is:

\[
\begin{align*}
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{ismin}[U] &= \text{false} \\
\text{index}[y] &= 4 \\
\text{low}[y] &= 4 \\
\text{ismin}[y] &= \text{true} \\
S &= [y; U] \\
n &= 5 \\
F &= \emptyset \\
F_U &= [e_5] \\
U &= \text{Find}(u) = \text{Find}(v) = \text{Find}(w) \\
\text{nb} &= 1 \\
\text{Finished} &= \{x\}
\end{align*}
\]

As for the node $x$, $\text{HVisit}(y)$ terminates by incrementing $\text{nb}$, popping $y$ from $S$ and adding it to $\text{Finished}$. Back to the execution of $\text{HVisit}(U)$, at Line 64, the state is:

\[
\begin{align*}
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{ismin}[U] &= \text{false} \\
\text{index}[y] &= 4 \\
\text{low}[y] &= 4 \\
\text{ismin}[y] &= \text{true} \\
S &= [U] \\
n &= 5 \\
F &= \emptyset \\
F_U &= [e_5] \\
U &= \text{Find}(u) = \text{Find}(v) = \text{Find}(w) \\
\text{nb} &= 2 \\
\text{Finished} &= \{y, x\}
\end{align*}
\]

While $\text{low}[U] = \text{index}[U]$, $\text{ismin}[U]$ is equal to $\text{false}$, so that no node merging loop is performed on $U$. Therefore, $e_5$ is not popped from $F_U$ and $\text{nb}$ is not incremented. Nevertheless, the loop from Lines 78 to 80 is executed, and after that, $\text{HVisit}(u)$ is terminated in the state:
Finally, HVisit($t$) is called from MinSccCount at Line 10. It can be verified that a trivial node merging loop is performed on $t$ only, and that $nb$ is incremented. After that, $t$ is placed into $Finished$. Therefore, the final state of MinSccCount is:

\[
\begin{align*}
\text{index}[x] &= 3 \\
\text{low}[x] &= 3 \\
\text{ismin}[x] &= \text{true} \\
\text{index}[U] &= 0 \\
\text{low}[U] &= 0 \\
\text{ismin}[U] &= \text{false} \\
\text{ismin}[t] &= \text{true} \\
F &= [e] \\
S &= [ ] \\
n &= 6 \\
\text{nb} &= 3 \\
\text{Finished} &= \{ t, U, y, x \}
\end{align*}
\]

Consequently, there is 3 minimal SCCs in the hypergraph. As $\text{ismin}[x] = \text{ismin}[y] = \text{ismin}[t] = \text{true}$ and $\text{ismin}[\text{Find}(z)] = \text{false}$ for $z = u, v, w$, they are given by the sets:

\[
\begin{align*}
\{ z \mid \text{Find}(z) = x \} &= \{ x \} \\
\{ z \mid \text{Find}(z) = y \} &= \{ y \} \\
\{ z \mid \text{Find}(z) = t \} &= \{ t \}.
\end{align*}
\]

\section{Determining All SCCs in a Hypergraph}

We just give an example of hypergraph for which our algorithm does not compute all the SCCs, but only minimal ones:

Our algorithm determines that $\{ t \}$ is the least SCC. However, $u$ and $x$ stand in the same SCC, but they are not merged by the algorithm. This is similar for $v$ and $y$. 