Discrete Applied Mathematics 296 (2021) 68–84

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Vertices removal for feasibility of clustered spanning trees

Nili Guttmann-Beck [a], Roni Rozen [a], Michal Stern [a,b,*]

[a] Academic College of Tel-Aviv Yaffo, Yaffo, Israel
[b] Caesarea Rothchild Institute, university of Haifa, Haifa, Israel

ARTICLE INFO

Article history:
Received 27 March 2019
Received in revised form 5 May 2020
Accepted 29 August 2020
Available online 19 September 2020

ABSTRACT

Let $H = \langle V, \mathcal{S} \rangle$ be a hypergraph, where V is a set of vertices and $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a set of not necessarily disjoint clusters $S_i \subseteq V$ such that $\cup_{i=1}^{m} S_i = V$. The Clustered Spanning Tree problem is to find a tree spanning all the vertices in V which satisfies that each cluster induces a subtree, when it exists. For cases when a given hypergraph does not have a feasible solution tree, we consider removing vertices from some clusters in order to gain feasibility. We provide a special layered graph to represent the intersections of the clusters and use this graph to find a suitable vertices removal which gains feasibility for the hypergraph. We consider how this approach manifests for different structures of hypergraphs using polynomial time algorithms.

1. Introduction

Let $H = \langle V, \mathcal{S} \rangle$ be a hypergraph, where V is a set of vertices and $\mathcal{S}$ is a set of clusters $S_1, \ldots, S_m$, $S_i \subseteq V$ for $i \in \{1, \ldots, m\}$, $\cup_{i=1}^{m} S_i = V$, and the clusters are not necessarily disjoint. The Clustered Spanning Tree problem, denoted by *CST*, is to find a tree spanning all the vertices of V, such that each cluster induces a subtree, if one exists. Note that in the literature clusters are also known as hyperedges and the feasible solution tree may be called tree support.

Verifying whether a hypergraph has a feasible solution tree can be performed in a few manners. According to [8,10,22] and summarized in [18], a hypergraph $H = \langle V, \mathcal{S} \rangle$ has a feasible solution tree if and only if it satisfies the Helly property and its intersection graph is chordal. In [9] and in [25] a different approach to check feasibility is introduced. by verifying chordality and acyclicity, which can be performed in a linear time complexity. Another approach is to use Algorithm *ES*, presented and proved in [13], which finds a maximum spanning tree in a weighted graph which represents the hypergraph. In the weighted graph the vertex set is V, an edge exists if there is a cluster which contains both endpoints of the edge. The weight of an edge (v, u) is the number of clusters which contain both v and u. The hypergraph has a feasible solution tree if and only if the weight of the maximum spanning tree is $\sum_{i=1}^{m} |S_i| - m$, and equality holds when the maximum spanning tree is also a feasible solution tree.

Using the conditions presented in [18] it can be shown that when the intersection graph of a hypergraph is a tree, the hypergraph has a feasible solution tree. Furthermore, in [13] it is proved that if every vertex is included in at most 2 clusters then a given hypergraph has a feasible solution tree if and only if the intersection graph is a tree. Similarly, in this paper we show that if a given hypergraph has a slender reduction graph, it has a feasible solution tree if and only if its intersection graph is a tree.

In this paper we consider hypergraphs with no feasible solution. For these hypergraphs we suggest a way to change the given instance in order to gain feasibility. The changes we consider are the removal of some vertices from clusters

* Corresponding author at: Academic College of Tel-Aviv Yaffo, Yaffo, Israel.
E-mail address: stern@mta.ac.il (M. Stern).

https://doi.org/10.1016/j.dam.2020.08.030
0166-218X/

which contain them. The main results of this paper are algorithms FindRem and FindSol. Algorithm FindRem finds a possible list of vertices removal, which creates a new hypergraph with a feasible solution tree. Algorithm FindSol finds the corresponding solution tree, thus proving the correctness of Algorithm FindRem. Both algorithms use a reduction graph which is a layered graph that represents all the intersection sets of the clusters. For the case where the sizes of all intersections are bounded by a constant C, both algorithms have a polynomial time complexity. On the other hand, when all intersections are big and contain more than $\frac{|V|}{2}$ vertices, we use previous results to prove that the given hypergraph has a feasible solution tree and no vertices removal is required.

Algorithm FindRem uses a spanning tree of the reduction graph, and its output of a suggested list of vertices removal depends on the structure of the tree. In this paper we present a few special structures of the spanning tree used by Algorithm FindRem. Procedure FindSpanNoDisV offers a spanning tree which ensures that after the vertices removal every vertex will stay in at least one cluster and thus will not disappear. Procedure FindSpanNoDisC offers a spanning tree which ensures that after the vertices removal every cluster will still contain at least one vertex and thus will not disappear. Procedure FindSpanInSlender is suitable for slender reduction graphs, where no intersection set contains another intersection set. Using the spanning tree offered by Procedure FindSpanInSlender yields a minimum cardinality list of vertices removal.

Throughout this paper, we assume that the intersection graph of any given hypergraph is connected. Otherwise, a feasible solution tree for the hypergraph can be constructed by properly adding edges between the feasible solutions of each connected component, if they exist. Therefore, the union of feasible removal lists of the different connected components creates a feasible removal list for the given hypergraph.

Related problems consider different structures of the solution tree and the clusters induced subtrees. In [23] a polynomial time algorithm is presented, which constructs a tree where each cluster spans a path. Another polynomial time algorithm for this problem is presented in [4]. This algorithm connects subpaths in a specific order using a special coloring of their end vertices. The most restricted problem, where both the spanning tree and subtrees are required to be paths, is in fact the Consecutive Ones Problem, which Booth and Leuker [3] solved in linear time complexity using PQ-trees.

Considering the optimization *CST* problem, the edges between vertices in V have weights, and the objective is to find a feasible solution tree for *CST* with minimum weight. This problem was solved in [15] where an optimum solution is found in $O(m^2|V|^4)$ time complexity, when a feasible solution exists. In addition, an abstraction of the problem using matroids is presented. For the restricted case where each cluster contains at most three vertices, there is a linear time algorithm and a polyhedral description of all feasible solutions. An improved time complexity algorithm for the optimization *CST* problem can be found in [14]. A special case of the optimization *CST* problem, where the optimum spanning tree solution is required to span a complete star on each cluster, is presented in [16]. Furthermore, a structure theorem which describes all feasible solutions and a polynomial time algorithm for finding an optimum solution is presented.

Another related optimization problem, called the clustered-*TSP*-path, arises when the optimum solution tree is required to be a *TSP*-path. The *TSP*-path is known to be NP-hard, which was proved in [6]. In [12] several algorithms for the not necessarily disjoint clustered *TSP*-path are presented. For a restricted case of the problem, an exact polynomial time algorithm is presented, and for other cases approximation algorithms are presented, whose efficiency depends on the structure of the hypergraph. A lot of research has also been investigated on the clustered *TSP*-path, where the clusters are disjoint. A heuristic for this problem is presented in [5], a branch and bound algorithm for solving this problem is presented in [17], and bounded-approximation algorithms are presented in [2] and [11]. In [1] the ordered disjoint clustered *TSP* is considered and an approximation algorithm is offered. In [19] a genetic algorithm for solving this problem is presented.

We would like to suggest three possible motivations for the *CST* problem. The first one comes from the area of communication networks and is presented in [24]. Given a complete graph where each vertex represents a customer, each edge represents a link between two customers, and there is a collection of not necessary disjoint clusters of vertices where each cluster represents a group of customers. The problem is to construct a communication tree network in such a way that each cluster of vertices from the given collection induces a subtree in the solution tree. In this way the network has the *group broadcast* property. If a customer wants to broadcast a message to a group that he belongs to, he sends it using only the links in the connected subtree representing the group. In addition, we get the *group fault tolerance* property, every group of customers is not sensitive to network faults that occur outside of the group. Also, the network has the *group privacy* property, for each group the communications between the members of the group do no traverse via customers outside the group. We note that when no feasible solution exists, we may consider removing some customers from some of the groups, in order to gain feasibility.

Another possible application of the *CST* problem arises from the area of databases with synchronous replications and is presented in [21]. Assume that we would like to construct a network with a number of databases, each of them is a database with synchronous replication. Every copy of the database is located in a vertex of the network, and the set of vertices which contain this database represents a cluster. A vertex may contain some replications of different databases, thus the clusters are not necessarily disjoint. In order to make the process of updating easy, our aim is to construct a spanning tree such that each database with all its replications will form a connected subtree in the network, enabling easy synchronizations between the different copies of each database. Constructing such a network is exactly the *CST* problem. When no feasible solution exists, we may consider removing some copies of some databases from several vertices, in order to gain feasibility.

The third application arises from the area of key management for secure group communications and is presented in [20]. Suppose that there are several non-disjoint groups of users and we would like to construct a communication network that will enable each group to have a secure group communication. From time to time we need to distribute a new key privately among all the group members. A solution for this problem, which uses a minimum number of edges, is a feasible solution tree for the *CST* problem. Ensuring that the induced subgraph on each group is a connected subtree enables privacy among the group members. When no feasible solution exists, we may consider removing some members from some of the groups, in order to gain feasibility.

The paper is organized as follows. In Section 2 we define the reduction graph which is a layered graph that introduces the intersection sets created by intersecting the clusters of a given hypergraph. In Section 3 we present an algorithm which uses the reduction graph to find a list of vertices removal which creates a hypergraph with a feasible solution tree and also present an algorithm to find an appropriate solution tree. Section 4 focuses on ensuring that the vertices removal will not cause a vertex to be removed from all clusters or that no cluster will be removed from the hypergraph. Section 5 considers hypergraphs where each intersection set contains more than $\frac{|V|}{2}$ vertices, and prove that such hypergraphs always have a feasible solution tree. Section 6 considers slender reduction graphs, where no intersection set contains another intersection set, for these hypergraphs we present an algorithm which finds a minimum cardinality vertices removal.

2. The reduction graph

In this section we introduce the reduction graph and some related definitions, which will be used later. The reduction graph introduces the intersection sets created by intersecting the clusters of a given hypergraph. The reduction graph is arranged by layers. The top layer contains a node for each cluster of $\mathcal{S}$ and the bottom layer contains a node for each vertex of V. The inside layers contain a node for each intersection set, such that each layer contains intersection sets of the same cardinality, where lower levels contain intersection sets of smaller cardinality.

Definition 2.1. Given a hypergraph $H = \langle V, \mathcal{S}\rangle$ where $\mathcal{S} = \{S_1, \ldots, S_m\}$, define ***IS*** (Intersection Sets) to contain all the intersection sets $IS = \{\bigcap_{S_i \in \mathcal{S}'} S_i | \mathcal{S}' \subseteq \mathcal{S}, |\mathcal{S}'| > 1\}$. Furthermore, define $\boldsymbol{ISS} = IS \cup \mathcal{S}$.

Definition 2.2. For every hypergraph $H = \langle V, \mathcal{S}\rangle$, define **the reduction graph $G^r = (V^r, E^r)$** as a layered graph, in the following manner:

V^r contains:

- Every $S_i \in \mathcal{S}$ is represented by a node $s_i \in V^r$. We call such a node a **cluster-node**. All cluster-nodes constitute the top layer of the reduction graph.
- Every intersection set $X \in IS$ is represented by a node $x \in V^r$. We call such a node an **intersection-node**. All intersection-nodes constitute the inside layers of the reduction graph. The layers are arranged according to their sizes, such that x_1 and x_2 are in the same layer if $|X_1| = |X_2|$, where X_1 and X_2 are the intersection sets represented by x_1 and x_2, respectively. In addition an intersection set with a bigger cardinality appears in a higher level.
- Every $v \in V$ is represented by a node $v \in V^r$. We call such a node a **vertex-node**. All vertex-nodes constitute the bottom layer of the reduction graph.

Define a **set-node** to be either a cluster-node or an intersection-node.

E^r contains:

- For every two set-nodes x_1, x_2 representing $X_1 \in IS$ and $X_2 \in ISS$, E^r contains the undirected edge (x_1, x_2) if $X_1 \subseteq X_2$ and there is no intersection set $X' \in IS$ which satisfies $X_1 \subseteq X' \subseteq X_2$.
- For every vertex-node $v \in V$ and a set-node x representing $X \in ISS$, E^r contains the undirected edge (v, x) if $v \in X$ and there is no intersection set $X' \in IS$ which satisfies $v \in X' \subseteq X$.

An example of a reduction graph is presented in Fig. 1.

Definition 2.3. Given a hypergraph $H = \langle V, \mathcal{S}\rangle$ and its reduction graph $G^r = (V^r, E^r)$, $X \in IS$ is a **minimal intersection set** if there is no $Y \in IS$ such that $Y \subsetneq X$. The intersection-node x which represents X is defined to be a **minimal intersection-node**. Note that, in the reduction graph the children nodes of a minimal intersection node are vertex-nodes.

In Fig. 1, the intersection sets $\{2, 3\}$, $\{4\}$ and $\{5\}$ are minimal intersection sets, and their corresponding nodes in the reduction graph are minimal intersection-nodes.

Property 2.4. *In a reduction graph G^r, the degree of every vertex-node $v \in V^r$ is one. The only edge touching v is the edge connecting it to the node representing the smallest set which contains v. This set is either a minimal intersection set or a cluster.*

Definition 2.5. Given a hypergraph $H = \langle V, \mathcal{S}\rangle$ and its reduction graph $G^r = (V^r, E^r)$, a set-node x is **contained** in a set-node y if x is at a lower level than y and their corresponding sets X and Y satisfy $X \subseteq Y$, where equality may occur only if Y is a cluster in $\mathcal{S}$. When x is contained in y there is a descending path from y to x in G^r.

For example, in Fig. 1, set-node $\{4\}$ is contained in set-nodes $\{4, 5\}$, $\{2, 3, 4\}$, s_2, s_3 and s_4.

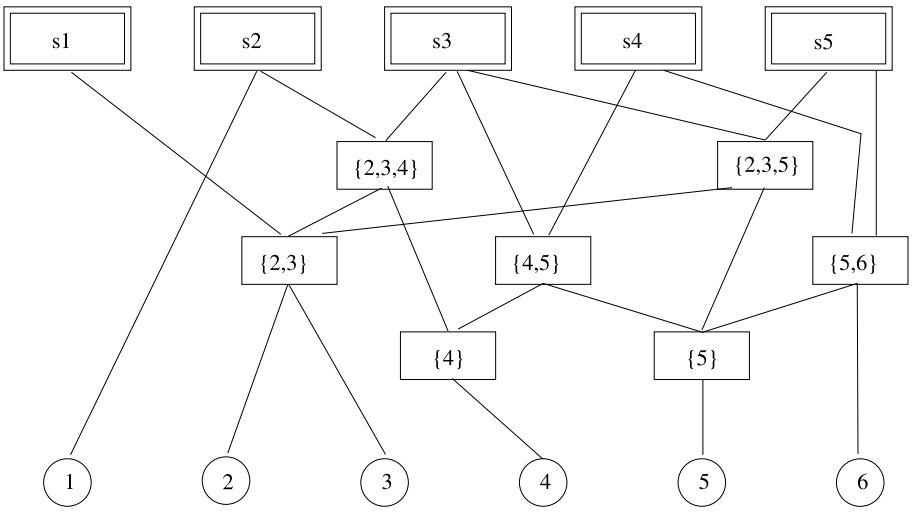

Fig. 1. The reduction graph for $H = \langle V, \mathcal{S} \rangle$ with $V = \{1, 2, 3, 4, 5, 6\}$, $\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5\}$ where $S_1 = \{2, 3\}, S_2 = \{1, 2, 3, 4\}, S_3 = \{2, 3, 4, 5\}, S_4 = \{4, 5, 6\}$ and $S_5 = \{2, 3, 5, 6\}$. Vertex-nodes are described in circles and set-nodes are described in rectangles with cluster-nodes in double-lined rectangles.

3. Gaining feasibility

For hypergraphs with no feasible solution, this section presents an algorithm that suggests a possible vertices removal, such that after applying this removal the hypergraph has a feasible solution tree. We use properties of the reduction graph to prove that any spanning tree of the reduction graph creates a suitable vertices removal suggestion. The section also includes an algorithm to construct a feasible solution tree for the hypergraph after the vertices removal using the spanning tree of the reduction graph.

In this section we use the idea of **induced subgraph**, denoted by $\boldsymbol{G[W]}$, for a set of vertices W of a graph G. The induced subgraph contains all the vertices of W and all the edges of G whose both endpoints are in W.

Definition 3.1. Let $H = \langle V, \mathcal{S} \rangle$ be a hypergraph. L is a **removal list of $\boldsymbol{H}$** if L is a list of pairs: $L = \{(v_1, S_{i_1}), \ldots, (v_k, S_{i_k})\}$ with $v_j \in S_{i_j}$, such that if we remove, for all the pairs in L, vertex v_j from cluster S_{i_j}, we create a new instance of the hypergraph denoted by $\boldsymbol{H \backslash L}$. Note that a cluster may appear in the list a few times, each time with a different vertex. If $H \backslash L$ has a feasible solution tree we say that L is a **feasible removal list of $\boldsymbol{H}$**. If L is also of minimum cardinality (minimum value of k) we say that L is a **minimum feasible removal list of $\boldsymbol{H}$**.

Note that every hypergraph has a feasible removal list. For example, the list which removes all the vertices from all the clusters. Furthermore, if L_1 is a feasible removal list and L_2 is a removal list which satisfies that $L_1 \subseteq L_2$, then L_2 is a feasible removal list.

3.1. Algorithms FindRem and FindSol

In this section we present two algorithms. Algorithm FindRem (Fig. 2) takes as input a hypergraph H and constructs the reduction graph G^r of this hypergraph. The algorithm uses a spanning tree T^r of the corresponding reduction graph to return L, a feasible removal list, defined in Definition 3.1. The idea is that for each cluster S_i, keep all the vertices whose corresponding vertex-nodes have a path connecting them to s_i, such that all the nodes in the path correspond to subsets of S_i. Thus, the hypergraph obtained by the removal list is determined by the sets $A(T^r, s_i)$, defined in Definition 3.3. This hypergraph has a feasible solution tree which can be constructed by applying Algorithm FindSol (Fig. 4) which uses the same spanning tree T^r.

Definition 3.2. Given a hypergraph $H = \langle V, \mathcal{S} \rangle$, its reduction graph $G^r = (V^r, E^r)$, G' a subgraph of G^r, $w, x \in V^r$ where x is a set-node, we say that **w is accessible to x in G'** if there is a path from x to w in G' satisfying that all the set-nodes in the path, excluding x and w, are contained in x.

Definition 3.3. Given a hypergraph $H = \langle V, \mathcal{S} \rangle$, its reduction graph $G^r = (V^r, E^r)$, T^r a spanning tree of G^r and $x \in V^r$ a set-node, $\boldsymbol{A(T^r, x)}$ is the set of all vertex-nodes in G^r which are accessible to x in T^r. For a vertex-node x we define $A(T^r, x) = \{x\}$.

For example, in Fig. 3, $A(T^r_{left}, \{2, 3, 5\}) = \{5\}$ and $A(T^r_{right}, \{2, 3, 5\}) = \{2, 3\}$.

FindRem(Finds a list of vertices to be Removed)
input
A hypergraph $H = \langle V, \mathcal{S} \rangle$.
returns
1. A reduction graph G^r.
2. A spanning tree T^r *of the reduction graph* G^r.
3. A removal list $L = \{(v_1, S_{i_1}), (v_2, S_{i_2}), \ldots, (v_k, S_{i_k})\}$ *where* $v_j \in S_{i_j}$ *for* $1 \leq j \leq k$.
begin
Build G^r *the reduction graph of* H.
T^r = *FindSpan*(G^r). *(See Remark 3.4)*
Initialize an empty list L.
for every ($v \in V$, $S_i \in \mathcal{S}$ *such that* $v \in S_i$ *and* $v \notin A(T^r, s_i)$)
Add (v, S_i) *to* L.
end for
return (G^r, T^r, L).
end *FindRem*

Fig. 2. Algorithm FindRem.

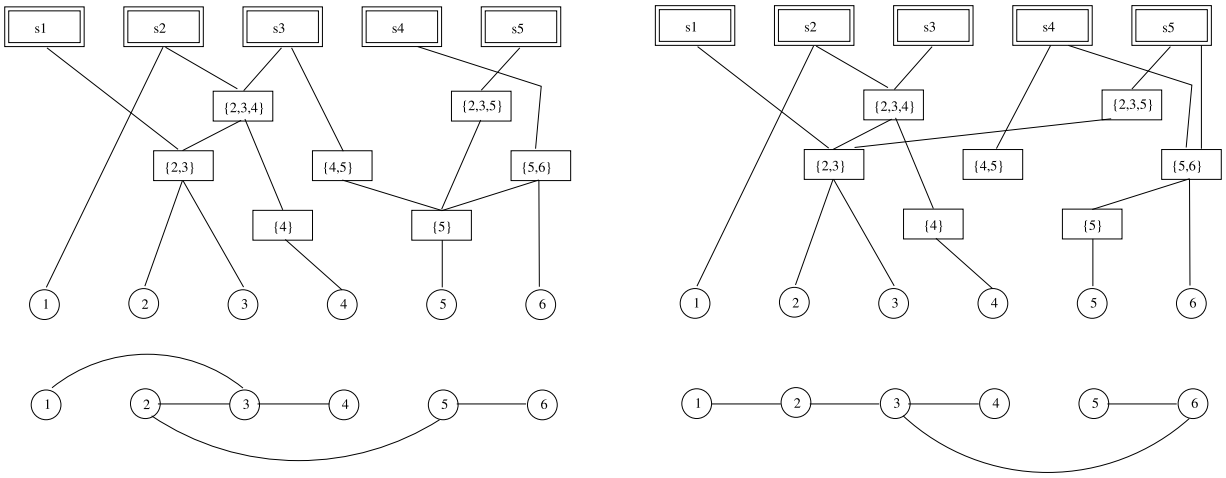

Fig. 3. Spanning trees T^r_{left} and T^r_{right} of the reduction graph in Fig. 1 and possible corresponding solution trees.

Remark 3.4. In Algorithm FindRem (Fig. 2) we call procedure FindSpan to find a spanning tree T^r of the reduction graph G^r. We prove later in this section that any spanning tree will produce a feasible removal list. However, in later sections we prove that special spanning tree procedures contribute special attributes to the removal list offered by Algorithm FindRem. Examples of different FindSpan procedures can be found in Figs. 7–10.

Example 3.5. Consider the Example presented in Fig. 1. Fig. 3 demonstrates Algorithms FindRem and FindSol. The top two drawings of the figure are two possible spanning trees of the reduction graph.

For the left spanning tree T^r_{left} Algorithm FindRem outputs removal list $\{(4, S_4), (2, S_5), (3, S_5)\}$. For the right spanning tree T^r_{right} the algorithm outputs removal list $\{(5, S_3), (4, S_4)\}$.

The bottom two drawings of the figure are possible results of Algorithm FindSol when applied on the two spanning trees, creating two different feasible solution trees.

When T^r_{left} is the input of algorithm FindSol, it may scan the nodes of the tree in the following order: {4}, {5}, {2, 3} [adding edge $2-3$], {4, 5}, {5, 6} [adding edge $5-6$], {2, 3, 4} [adding edge $3-4$], {2, 3, 5}, s_1, s_2 [adding edge $1-3$], s_3 [adding edge $2-5$], s_4, s_5.

When T^r_{right} is the input of algorithm FindSol, it may scan the nodes of the tree in the following order: {4}, {5}, {2, 3} [adding edge $2-3$], {4, 5}, {5, 6} [adding edge $5-6$], {2, 3, 4} [adding edge $3-4$], {2, 3, 5}, s_1, s_2 [adding edge $1-2$], s_3, s_4, s_5 [adding edge $3-6$].

The bottom drawings are the corresponding feasible solution trees.

FindSol(Finds a Solution tree)
input
The return values of Algorithm FindRem (Figure 2) for a given hypergraph $H = \langle V, \mathcal{S} \rangle$:
1. The reduction graph $G^r = (V^r, E^r)$,
2. A spanning tree T^r of G^r,
3. The removal list L which corresponds to T^r.
returns
A solution tree of $H \backslash L$, denoted by F.
begin
Initialize F to contain V with no edges.
Let all the set-nodes of T^r be unmarked.
Scan T^r from bottom to top (according to the order of the layers of G^r)
for every *(unmarked set-node $x \in V^r$)*
1. Add to F edges to create a subtree spanning all the vertices of $A(T^r, x)$ which are isolated vertices in F.
2. Connect, without creating any cycle, all the connected components of $F[(A(T^r, x)]$, using edges whose both endpoints are in $A(T^r, x)$.
3. Mark x.
end for
if *(F is not connected)*
then *Add arbitrarily edges to connect F, without creating any cycle.*
end if
return *F.*
end *FindSol*

Fig. 4. Algorithm FindSol.

3.2. Correctness of algorithms

In this section we prove the correctness of the two algorithms described in Section 3.1. The main theorem of this section claims that $H \backslash L$, where L is the removal list returned by Algorithm FindRem (Fig. 2), has a feasible solution tree which is found by Algorithm FindSol (Fig. 4). Thus proving that the output of FindRem is a feasible removal list of a given hypergraph.

Property 3.6. *In Algorithm FindSol (Fig. 4) :*

1. *Throughout the algorithm, F is cycle-less.*
2. *When the algorithm reaches a set-node x, all the nodes which are contained in x (and as such are in lower levels than x) are marked.*
3. *When the algorithm processes a set-node x, all the edges which are added to F are with both endpoints in $A(T^r, x)$.*
4. *When a set-node x is marked, there are no isolated vertices in $A(T^r, x)$.*
5. *If a vertex $v \in V$ is not isolated in F, then there is a set-node y such that $v \in A(T^r, y)$, and that during the process of y, an edge touching v is added to F.*

Lemma 3.7. *When Algorithm FindSol (Fig. 4) reaches an unmarked set-node x which is a minimal intersection-node, all the nodes in $A(T^r, x)$ are isolated in F.*

Proof. Suppose otherwise, there is a node $v \in A(T^r, x)$ which is not isolated in F. According to Property 3.6, there is a set-node y with $v \in A(T^r, y)$, such that an edge touching v was added to F when y was processed. Since x is a minimal intersection-node, it occurs in the lowest level of set-nodes whose corresponding intersection sets contain v, hence x is at a lower level than y. Contradicting the fact that the algorithm processes the set-nodes according to the order of the layers of the reduction graph, from bottom to top. ■

The following lemma is fundamental for the proof of correctness of Algorithm FindSol. We prove that when the algorithm starts to process a set-node x, all paths between vertices that are contained in $A(T^r, x)$ use only vertices from $A(T^r, x)$. Note that such a path may be contained in several (possibly more than two) $A(T^r, y_i)$, for y_i which are son nodes of x.

Lemma 3.8. *When Algorithm FindSol (Fig. 4) starts to process a set-node x, if for every set-node y, which was marked before this step, $F[A(T^r, y)]$ is a subtree, then for every two vertices $v, u \in A(T^r, x)$ which are connected in F by a path, the path uses only vertices in $A(T^r, x)$.*

Proof. To avoid ambiguity between paths in T^r and paths in F we denote by $v \sim u$ a path in T^r and $v - u$ a path in F.

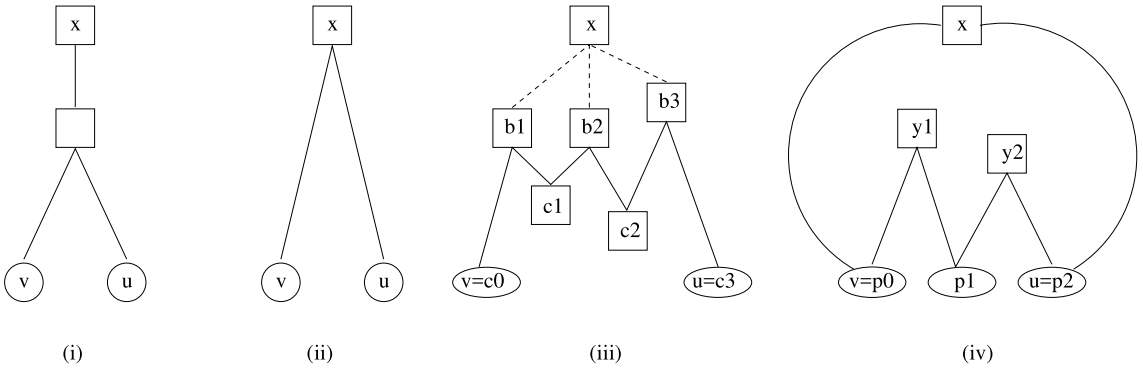

Fig. 5. Figure supporting the proof of Lemma 3.8.

Suppose by contradiction that there are two vertices $v, u \in A(T^r, x)$, with a path $v - u$ (in F) which uses vertices not in $A(T^r, x)$. According to Property 3.6, F is cycle-less and therefore this is the only path between v and u in F. Since v and u are not isolated in F, when the algorithm starts to process x, according to Lemma 3.7, x cannot be a minimal intersection-node. From Definitions 3.2 and 3.3, paths $v \sim x$ and $u \sim x$ (in T^r) use only set-nodes which are contained in x. The union of these two paths contains a path $v \sim u$ in T^r which either does not contain set-node x (as in Fig. 5(i)) or contains set-node x (as in Fig. 5(ii)). All the set-nodes of the path, except x (if it is in the path), are contained in x and are accessible to x in T^r. The rest of the proof considers whether x is in the path or not:

1. The path $v \sim u$ does not contain set-node x (Fig. 5(i)).
   In this case all the nodes in the path are contained in x and are accessible to x in T^r. Define minimal (maximal) nodes of the path as nodes whose neighbors in the path are of a higher (lower) level, respectively. The vertex-nodes v and u are in the bottom level of G^r and are therefore minimal nodes in this path. Mark $b_1, \ldots, b_k$ all the maximal nodes of this path and $c_0, \ldots, c_k$ all the minimal nodes of the path, where $c_0 = v$ and $c_k = u$. An illustration of this path is shown in Fig. 5(iii). According to the structure of the path, c_{i-1} and c_i are contained in b_i and therefore are accessible to b_i in T^r. If a set-node $x \in V^r$ is accessible to another set-node $y \in V^r$, then $A(T^r, x) \subseteq A(T^r, y)$. Therefore, $A(T^r, c_{i-1}) \subseteq A(T^r, b_i)$ and $A(T^r, c_i) \subseteq A(T^r, b_i)$. Since b_i is contained in x, by Property 3.6, b_i is already marked. Choose $k+1$ different vertices $w_0, \ldots, w_{k+1}$ such that $w_o = v$, $w_{k+1} = u$ and for every $i \in \{1, \ldots, k\}$ vertex $w_i \in A(T^r, c_i)$. By the assumption of the lemma, $F[A(T^r, b_i)]$ is a subtree, thus for every $i \in \{1, \ldots, k\}$, F contains a path $w_{i-1} - w_i$ whose vertices are in $A(T^r, b_i)$. The union of all these paths gives a path $v - u$, whose vertices are in $\bigcup_{i=1}^{k} A(T^r, b_i)$. Since every b_i is accessible to x in T^r, $A(T^r, b_i) \subseteq A(T^r, x)$ which means that also $\bigcup_{i=1}^{k} A(T^r, b_i) \subseteq A(T^r, x)$ and all the vertices in the path $v - u$ are in $A(T^r, x)$, contradicting the assumption in the beginning of the proof.
2. The path $v \sim u$ contains set-node x (Fig. 5(ii)).
   Denote the vertices in the path $v - u$ (in F) as $p_0, \ldots, p_l$ where $p_0 = v$ and $p_l = u$. Let y_i, for $i = 1, \ldots, l$, be the set-node in T^r which was processed by the algorithm when the edge (p_{i-1}, p_i) was added to F. By Property 3.6, both $p_{i-1} \in A(T^r, y_i)$ and $p_i \in A(T^r, y_i)$. Since x is processed after y_i is marked, by Property 3.6, x is **not** contained in y_i. By Definitions 3.2 and 3.3, paths $p_{i-1} \sim y_i$ and $p_i \sim y_i$ do not contain set-node x. Hence T^r contains a path $p_{i-1} \sim p_i$, for $i = 1, \ldots, l$, which does not contain set-node x. An illustration of these paths is shown in Fig. 5(iv). The union of these paths $p_{i-1} \sim p_i$, for $i = 1, \ldots, l-1$, contains a path in T^r between v and u which does not contain x, contradicting the assumption of this case.

Hence, when Algorithm FindSol starts to process x, if F contains a path $v - u$ for $v, u \in A(T^r, x)$, the path uses only vertices in $A(T^r, x)$. ■

Theorem 3.9. *When Algorithm FindSol (Fig. 4) marks a set-node x, $F[A(T^r, x)]$ is a subtree.*

Proof. We prove the theorem using induction on the level of x in T^r.

If x is a minimal intersection-node, then according to Lemma 3.7, when the algorithm starts to process x, all the vertices in $A(T^r, x)$ are isolated in F. All these vertices are added to F as a subtree in step 1 during the process of x. This subtree is the only connected component containing vertices from $A(T^r, x)$, so no edges are added in step 2 during the process of x. Hence, when x is marked, $F[A(T^r, x)]$ is a subtree.

Suppose by induction that for every set-node y which is marked before x is processed, $F[A(T^r, y)]$ is a subtree.

Consider the process of set-node x: by Lemma 3.8, when the algorithm starts to process x, F does not contain paths between different connected components of $F[A(T^r, x)]$. Since F is cycle-less every connected component of $F[A(T^r, x)]$ is

either a subtree or an isolated vertex. When the algorithm marks x, $F[A(T^r, x)]$ is one connected cycle-less component, and all the paths between vertices in this component contain only vertices from $A(T^r, x)$, giving that $F[A(T^r, x)]$ is a subtree. ■

Corollary 3.10. *When Algorithm FindSol (Fig. 4) ends, $F[A(T^r, x)]$ is a subtree for every set-node $x \in V^r$.*

Theorem 3.11. *Algorithm FindRem (Fig. 2) finds a removal list L which is a feasible removal list.*

Proof. According to the way L is created in Algorithm FindRem, for every cluster S_i, L contains all the vertices whose corresponding vertex-nodes are not accessible to s_i in T^r. Hence, after the vertices removal, the clusters are changed to be $\{A(T^r, s_1), \ldots, A(T^r, s_m)\}$. According to Corollary 3.10, at the end of Algorithm FindSol, $F[A(T^r, s_i)]$ is a subtree of F for every cluster-node s_i. At the last step of the algorithm, edges are added to F to create connectivity. The addition of the edges is performed without creating any cycle, hence at the end of the algorithm, F is connected and cycle-less, such that $F[A(T^r, s_i)]$ is a subtree for every $S_i \in \mathcal{S}$. Thus, F is a solution tree for $H\backslash L$ and L is a feasible removal list. ■

Corollary 3.12. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, Algorithms FindRem and FindSol (Figs. 2 and 4) find a feasible removal list L and a solution tree for $H\backslash L$.*

Theorem 3.13. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, if there exists a constant C such that $|S_i \cap S_j| \leq C$ for every $S_i, S_j \in \mathcal{S}$, then Algorithms FindRem and FindSol (Figs. 2 and 4) find a feasible removal list and a solution tree in polynomial time.*

Proof. When $|S_i \cap S_j| \leq C$ for every $S_i, S_j \in \mathcal{S}$, the size of each intersection set in IS is also bounded by C. Thus, the size of IS is polynomial, and therefore the number of nodes in G^r is polynomial. Since Algorithms FindRem and FindSol are polynomial in the number of nodes in G^r, their time complexity is polynomial. ■

4. Non disappearing vertices and clusters

Algorithm FindRem (Fig. 2) suggests a feasible removal list that depends on the spanning tree used by the algorithm. The removal list may remove a vertex from every cluster which contains it, causing the disappearance of this vertex. On the other hand, the removal list may remove all the vertices contained in a cluster, causing the disappearance of this cluster. In Example 4.1, if Algorithm FindRem uses the left spanning tree in Fig. 6, it will suggest removing vertex 1 from all the clusters which contain it, thus causing the disappearance of this vertex. On the other hand, if Algorithm FindRem uses the right spanning tree in Fig. 6, it will suggest removing all the vertices of cluster S_1, thus causing the disappearance of this cluster.

In this section we present two procedures that find spanning trees of the reduction graph.

If Algorithm FindRem uses the spanning tree offered by Procedure FindSpanNoDisV (Fig. 7) every vertex is still contained in at least one cluster after the vertices removal. The spanning tree offered by this procedure ensures that every vertex-node will have at least one path connecting it to a cluster-node. This is done by choosing, for every vertex-node and every intersection-node of the reduction graph, at least one edge connecting it to a higher level.

If Algorithm FindRem uses the spanning tree offered by Procedure FindSpanNoDisC (Fig. 8) every cluster still contains at least one vertex after the vertices removal. The spanning tree offered by this procedure ensures that every cluster-node has at least one path connecting it to a vertex-node. This is done by choosing, for every set-node of the reduction graph, at least one edge connecting it to a lower level.

Procedure FindSpanNoDisV (Fig. 7) is a generalization of a similar procedure for the restricted case when the cardinality of the intersection sets is bounded by 2, which appears in [7].

Example 4.1. Consider hypergraph $H = \langle V, \mathcal{S}\rangle$ with $V = \{1, 2, 3, 4, 5, 6\}$, $\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5\}$ where $S_1 = \{2, 3, 4\}$, $S_2 = \{3, 4, 5\}$, $S_3 = \{1, 2, 4, 5\}$, $S_4 = \{1, 2, 5, 6\}$ and $S_5 = \{2, 3, 5, 6\}$. Fig. 6 presents two possible spanning trees of the reduction graph of H.

If Algorithm FindRem uses the left spanning tree T_1^r the output is the feasible removal list $L_1 = \{(1, S_3), (2, S_3), (1, S_4), (2, S_4), (2, S_5), (3, S_5)\}$. The clusters in $H\backslash L_1$ are $S_1 = \{2, 3, 4\}$, $S_2 = \{3, 4, 5\}$, $S_3 = \{4, 5\}$, $S_4 = \{5, 6\}$ and $S_5 = \{5, 6\}$. Since vertex 1 is removed from clusters S_3 and S_4, using T_1^r causes the disappearance of vertex 1.

If Algorithm FindRem uses the right spanning tree T_2^r the output is the feasible removal list $L_2 = \{(2, S_1), (3, S_1), (4, S_1), (4, S_2), (5, S_2), (1, S_4), (2, S_4), (2, S_5)\}$. The clusters in $H\backslash L_2$ are $S_1 = \{\}$, $S_2 = \{3\}$, $S_3 = \{1, 2, 4, 5\}$, $S_4 = \{5, 6\}$ and $S_5 = \{3, 5, 6\}$. Since S_1 is empty in $H\backslash L_2$, using T_2^r causes the total removal of cluster S_1.

Lemma 4.2. *Procedure FindSpanNoDisV (Fig. 7) returns a spanning tree of the reduction graph.*

Proof. In Procedure FindSpanNoDisV, at the end of Step 2, $V(T^r)$ includes all the vertex-nodes and all the intersection-nodes. At the end of Step 3 it also includes all the cluster-nodes. Hence, at the end of the procedure $V(T^r) = V^r$.

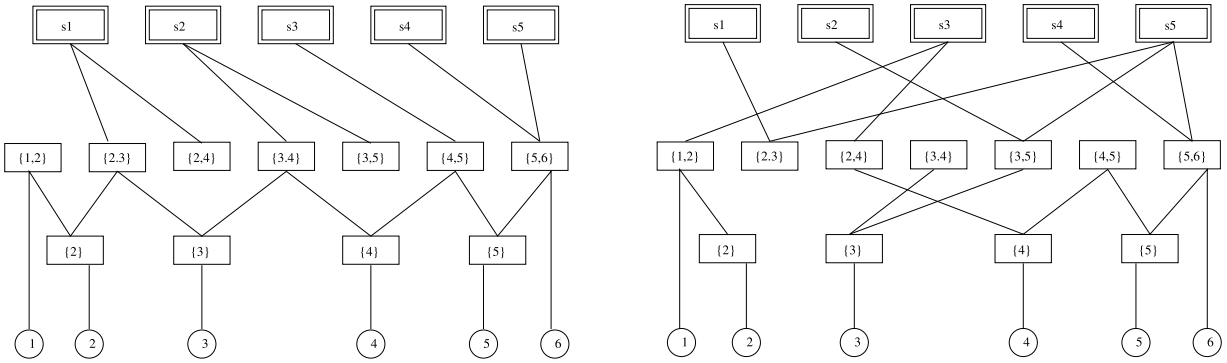

Fig. 6. Two optional spanning trees: T_1^r (on the left) and T_2^r (on the right), for the reduction graph of the hypergraph described in Example 4.1.

FindSpanNoDisV(Finds a Spanning tree of the reduction graph with No Disappearing Vertices)
input
The reduction graph $G^r = (V^r, E^r)$.
returns
A spanning tree T^r *of* G^r.
begin
1. Initialize $V(T^r)$ *and* $E(T^r)$ *to be an empty list of vertices and edges, respectively.*
2. **for every** *(node* $u \in V^r$ *which is a vertex-node or an intersection-node)*
Choose a node $w \in V^r$ *such that* $(u, w) \in E^r$ *and* w *is at a higher level than* u.
$E(T^r) = E(T^r) \cup \{(u, w)\}$.
$V(T^r) = V(T^r) \cup \{u, w\}$.
end for
3. **for every** *(cluster-node* $u \in V^r$*)*
if *(*$u \notin V(T^r)$*)*
Choose a node $w \in V^r$ *such that* $(u, w) \in E^r$.
then $E(T^r) = E(T^r) \cup \{(u, w)\}$.
$V(T^r) = V(T^r) \cup \{u\}$.
end if
end for
4. Add to T^r *edges to connect between its connected components without creating any cycle.*
return T^r
end *FindSpanNoDisV*

Fig. 7. Procedure FindSpan with no disappearing vertices.

By the way the procedure works, at the end of Step 2, every node $u \in V^r$ has exactly one edge (u, w), for w at a higher level than u. Suppose by contradiction that, at the end of Step 2, T^r contains a cycle. Let u_c be the node with the lowest level in C. Since u_c is in the cycle, there are two edges of the cycle in T^r which touch u_c and connect it to nodes of higher level, contradicting the way edges are added to $E(T^r)$ in Step 2. Therefore, at the end of Step 2, T^r is cycle-less. In Step 3 we add to $V(T^r)$ cluster-nodes which were not added in Step 2. Since, at the end of step 3, the degree of all these nodes in T^r is one, they cannot be part of any cycle. Hence, at the end of Step 3, T^r is cycle-less. In Step 4 we only add edges to connect between different connected components of T^r, without creating any cycle. Thus, at the end of this step, T^r is cycle-less.

Furthermore, at the end of Step 4 T^r is connected.

Therefore, at the end of the Procedure, T^r contains all the nodes of the reduction graph, is cycle-less and connected, and thus is a spanning tree of the reduction graph. ■

Lemma 4.3. *If Algorithm FindRem (Fig. 2) uses the spanning tree offered by Procedure FindSpanNoDisV (Fig. 7), no vertices will disappear from the hypergraph.*

Proof. In the spanning tree T^r returned by Procedure FindSpanNoDisV, every node $u \in V^r$, which is either a vertex-node or an intersection-node, has at least one edge connecting it to a higher level, this is the edge which is added in Step 2 when processing u. Therefore, at the end of the procedure, for each vertex-node v we can start a path from v to a cluster-node s_i, corresponding to cluster S_i, using only edges from a lower level to a higher level. This path uses only nodes which are

FindSpanNoDisC(Finds a Spanning tree of the reduction graph with No Disappearing Clusters)
input
The reduction graph $G^r = (V^r, E^r)$.
returns
A spanning tree T^r *of* G^r.
begin
1. Initialize $V(T^r)$ *and* $E(T^r)$ *to be an empty list of vertices and edges, respectively.*
2. **for every** *(set-node* $u \in V^r$*)*
Choose a node $w \in V^r$ *such that* $(u, w) \in E^r$ *and* w *is at a lower level than* u.
$E(T^r) = E(T^r) \cup \{(u, w)\}$.
$V(T^r) = V(T^r) \cup \{u, w\}$.
end for
3. **for every** *(vertex-node* $u \in V^r$*)*
if $(u \notin V(T^r))$
Choose a node $w \in V^r$ *such that* $(u, w) \in E^r$.
then $E(T^r) = E(T^r) \cup \{(u, w)\}$.
$V(T^r) = V(T^r) \cup \{u\}$.
end if
end for
4. Add to T^r *edges to connect between its connected components without creating any cycle.*
return T^r
end *FindSpanNoDisC*

Fig. 8. Procedure FindSpan with no disappearing clusters.

contained in s_i, so v is accessible to s_i in T^r. Hence, every vertex-node v is accessible to at least one cluster and does not disappear from the hypergraph. ■

Lemma 4.4. *Procedure FindSpanNoDisC (Fig. 8) returns a spanning tree of the reduction graph.*

Proof. Similar to the proof of Lemma 4.2. ■

Lemma 4.5. *If Algorithm FindRem (Fig. 2) uses the spanning tree offered by Procedure FindSpanNoDisC (Fig. 8), no clusters will disappear from the hypergraph.*

Proof. In the spanning tree T^r returned by Procedure FindSpanNoDisC, every set-node $u \in V^r$ has at least one edge connecting it to a lower level, this is the edge which is added in Step 2 when processing u. Therefore, at the end of the procedure, for every cluster-node s_i, corresponding to cluster S_i, we can start a path from s_i until we reach a vertex-node v, using only edges from a higher level to a lower level. This path uses only nodes which are contained in s_i, so v is accessible to s_i in T^r. Hence, for every cluster S_i there is at least one vertex-node v which is accessible to cluster-node s_i, and cluster S_i does not disappear from the hypergraph. ■

5. Big-intersection hypergraphs

In this section we consider big-intersection hypergraphs in which every intersection set contains more than $\frac{|V|}{2}$ vertices, defined in Definition 5.1. We prove that these hypergraphs always have a feasible solution tree. The proof is based on the fact that if we use a specific spanning tree of the reduction graph which is the output of Procedure FindSpanInBig (Fig. 9), the removal list offered by FindRem (Fig. 2) is empty.

This procedure is based on the idea that in big intersection hypergraphs, every pair of intersection-nodes corresponds to intersection sets with nonempty intersection. In this case we can use containment cycles, defined in Definition 5.3, which are specially structured cycles of the intersection graph. Removing specific edges from these cycles creates the required spanning tree, which maintains accessibility between every vertex-node v and every cluster-node s_i, such that $v \in S_i$.

The purpose of this section is to prove that every big intersection hypergraph has a feasible solution tree. In practice, after verifying that a given hypergraph is a big intersection hypergraph, finding a feasible solution tree can be performed using known algorithms such as Algorithm *ES* in [13] or the algorithms presented in [18] or [25].

Definition 5.1. A hypergraph $H = \langle V, \mathcal{S} \rangle$ is a **big-intersection hypergraph** if every intersection set $X \in IS$ satisfies $|X| > \frac{|V|}{2}$.

FindSpanInBig(Finds a Spanning tree In the reduction graph of Big-intersection hypergraphs)
input
The reduction graph $G^r = (V^r, E^r)$ of a big-intersection hypergraph.
returns
A spanning tree T^r of G'.
begin
Set T^r to contain all the edges in E^r.
while *(T^r contains a cycle)*
Find cycle C in T^r whose maximal node y is of the highest level.
Remove one of the edges in C with an end-point y.
end while
return T^r
end *FindSpanInBig*

Fig. 9. Procedure FindSpan In Big-intersection hypergraphs.

5.1. Containment cycles

In this section we define containment cycles and small containment cycles. In a containment cycle there is one node that is included in all the other nodes of the cycle, thus this is a minimal node with respect to the cycle. In a small containment cycle there is also one node which contains all the other nodes of the cycle, thus this is the only maximal node with respect to the cycle. As proved in this section, in a small containment cycle, we can remove one of two specific edges of the cycle without changing the accessibility of any of the vertex-nodes to the cluster-nodes which contain them.

Definition 5.2. Given a hypergraph $H = \langle V, \mathcal{S} \rangle$, its reduction graph $G^r = (V^r, E^r)$ and $G' = (V', E')$ a subgraph of G^r, a **maximal node with respect to G'** is a node $u \in V'$, such that all the edges of E' which touch u are edges connecting u to lower level nodes. A **minimal node with respect to G'** is a node $u \in V'$, such that all the edges of E' which touch u are edges connecting u to higher level nodes.

For example, in Fig. 1, consider path $P = 2 - \{2,3\} - \{2,3,5\} - \{5\} - \{5,6\} - 6$. Set-nodes $\{2,3,5\}$ and $\{5,6\}$ are maximal nodes with respect to P and set-node $\{5\}$ and vertex-nodes 2 and 6 are minimal nodes with respect to P.

Note that, in a reduction graph a vertex-node has always degree 1, and hence it cannot be contained in any cycle of the reduction graph. Hence, every minimal node with respect to a cycle is an intersection-node.

Definition 5.3. Given a hypergraph $H = \langle V, \mathcal{S} \rangle$ and its reduction graph G^r, a set-node $x \in V^r$ and a cycle $C \in G^r$ which contains x, we define x to be a **containment node with respect to C** if for every $z \in C$, x is contained in z. In this case we say that C is a **containment cycle**. In addition, a containment cycle $C \in G^r$ is a **small containment cycle** if there is exactly one maximal node with respect to C.

Note, if x is a containment node with respect to a cycle C, then x is also a minimal node with respect to the cycle, but not necessarily vice versa.

Lemma 5.4. *Given a hypergraph $H = \langle V, \mathcal{S} \rangle$ and its reduction graph G^r, every containment cycle C has exactly one containment node with respect to C, which is a minimal node with respect to C. Furthermore, all the set-nodes in a small containment cycle C are contained in the maximal node with respect to C.*

Proof. Suppose by contradiction that C is a containment cycle with two different containment nodes x and y. By Definition 5.3, x and y are minimal nodes with respect to C. Since x is a containment node of C and $y \in C$, Definition 5.3 implies that $X \subseteq Y$. Similarly, since y is a containment node of C, $Y \subseteq X$, contradicting $X \neq Y$.

A small containment cycle is composed of two paths between its unique containment node and its unique maximal node. Clearly, all the set-nodes in these two paths are contained in the maximal node. ■

The next lemma states the most important feature of a small containment cycle. If we remove one of the edges which touch y, where y is the maximal node of the cycle, all the vertex-nodes which were accessible to y, before the edge removal, are still accessible to y. Furthermore, every vertex-node which was accessible to any set-node in G^r is still accessible to this node after the edge removal.

Lemma 5.5. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$ and its reduction graph $G^r = (V^r, E^r)$, C a small containment cycle, y the maximal node with respect to C, x a son of y in C, and G' a subgraph of G^r which contains C.*

1. *If a node $v \in V^r$ is accessible to y in G', then v is accessible to y in $G'\backslash\{(x, y)\}$.*
2. *If a node $v \in V^r$ is accessible to a set-node u in G', then v is accessible to u in $G'\backslash\{(x, y)\}$.*

Proof.

1. Since v is accessible to y in G', G' contains a path $v \sim y$ which uses set-nodes which are contained in y. If this path does not use edge (x, y), then path $v \sim y$ is also in $G'\backslash\{(x, y)\}$. If path $v \sim y$ contains edge (x, y), then replace edge (x, y) by the subpath $x \sim m \sim y$, where m is the containment node with respect to C. Since G' contains C, $G'\backslash\{(x, y)\}$ contains subpath $x \sim m \sim y$. According to Lemma 5.4, all the set-nodes in subpath $x \sim m \sim y$ are contained in y, yielding a path between v and y in $G'\backslash\{(x, y)\}$ which uses only set-nodes contained in y.
   In both cases, $G'\backslash\{(x, y)\}$ contains a path between v and y which uses only set-nodes that are contained in y, creating accessibility of v to y in $G'\backslash\{(x, y)\}$.
2. Since v is accessible to u in G', there is a path P in G' which connects v and u and uses only set-nodes contained in u. If P does not contain edge (x, y), then clearly v is accessible to u in $G'\backslash\{(x, y)\}$. If P uses edge (x, y), then either $u = y$ or y is contained in u. If $u = y$, then according to the previous case of this lemma, v is accessible to u in $G'\backslash\{(x, y)\}$. If y is contained in u, then v is accessible to y in $G'\backslash\{(x, y)\}$, and therefore $G'\backslash\{(x, y)\}$ contains a path P' between v and y which uses only set-nodes contained in y. Since y is contained in u, every node which is contained in y is also contained in u. Therefore, the set-nodes in P' are also contained in u. Let P'' be the subpath of P which connects y and u. Therefore, the set-nodes in P'' are contained in u. The concatenation of P' and P'' is in $G'\backslash\{(x, y)\}$ and uses only set-nodes contained in u. Thus v is accessible to u in $G'\backslash\{(x, y)\}$. ■

5.2. Feasibility of big-intersection hypergraphs

In this section we prove that a big-intersection hypergraph always has a feasible solution tree using Procedure FindSpanInBig (Fig. 9), which is only used to theoretically prove the feasibility. Checking whether a hypergraph is a big intersection hypergraph can be performed in $O(m|V|^2)$, as is proved in Theorem 5.6. Furthermore, constructing a feasible solution tree of a big-intersection hypergraph can be performed using known algorithms.

Theorem 5.6. *Checking whether a hypergraph $H = \langle V, \mathcal{S}\rangle$ is a big-intersection hypergraph can be performed in $O(m|V|^2)$ time complexity.*

Proof. Assume the hypergraph is presented by a table, where each row represents a vertex from V and each column represents a cluster of $\mathcal{S}$. A matrix cell contains 1 if and only if the vertex of the row is contained in the cluster of the column. Perform the following steps:

1. Traverse the input table and create for each vertex $v \in V$ the list of clusters $\{S_{i_1^v}, \ldots, S_{i_k^v}\}$ containing this vertex.
2. For every $v \in V$ find the smallest cardinality intersection set $S_{i_1^v} \cap \cdots \cap S_{i_k^v}$ which contains v.
3. Count the number of vertices in each one of the calculated intersection sets.

Note that the intersection sets calculated in the above steps include all the minimal intersection sets, but may also include some intersection sets which are not minimal.

H is a big-intersection hypergraph if and only if each one of the intersection sets contains more than $\frac{|V|}{2}$ vertices.

The complexity of the first step is $O(m|V|)$, the second step is $O(m|V|^2)$, and the last step $O(|V|^2)$, thus achieving the required complexity. ■

In the following lemma we give some conditions on a subgraph G' of the reduction graph. These conditions correspond to the way Procedure FindSpanInBig (Fig. 9) operates and are kept during the process of the algorithm. The lemma states that when these conditions are kept and if G' contains a cycle with a maximal node y, then it also contains a small containment cycle whose unique maximal node is y. According to Lemma 5.5, we can remove one of the edges which connects y to one of its neighbors in C, while maintaining the accessibility of all the vertex-nodes to all the cluster-nodes which correspond to clusters that contain them.

Lemma 5.7. *Given a big-intersection hypergraph $H = \langle V, \mathcal{S}\rangle$ and its reduction graph G^r, consider the following conditions:*

1. *G' is a subgraph of G^r which contains a cycle C,*
2. *y is a maximal node with respect to C,*
3. *x_1 and x_2 are the sons of y in C,*
4. *G' contains every edge in G^r whose both endpoints are of level not higher than y.*

If all four conditions are satisfied, then G' contains a small containment cycle C', such that y is the unique maximal node with respect to C' and x_1, x_2 are the sons of y in C'.

Proof. Let X_1, X_2 be the intersection sets which correspond to x_1 and x_2, respectively. Since G^r is a reduction graph of a big-intersection hypergraph, $|X_1| > \frac{|V|}{2}$ and $|X_2| > \frac{|V|}{2}$. Hence $X_1 \cap X_2 \neq \phi$, and there is a set-node $w \in V^r$ which represents the intersection $X_1 \cap X_2 \in IS$. The union of paths $w \sim x_1 \sim y$ and $w \sim x_2 \sim y$ creates a small containment cycle C', such that y is the unique maximal node with respect to C'. Clearly, C' also contains x_1 and x_2 as the sons of y. Since all the edges in C' touch only nodes which are contained in y and are therefore of lower level than y, according to the assumption of the lemma, G' contains all the edges in the containment cycle C'. ■

Theorem 5.8. *If Algorithm FindRem (Fig. 2) uses the spanning tree returned by Procedure FindSpanInBig (Fig. 9), then the removal list L returned by Algorithm FindRem is empty.*

Proof. Consider a vertex v and a cluster S_i, such that $v \in S_i$.

At the beginning of Procedure FindSpanInBig, T^r contains all the edges of E^r. Therefore, at this stage, v is accessible to cluster-node s_i in T^r, where s_i is the cluster-node which corresponds to S_i.

By the way the procedure works, when it considers a cycle C, such that y is a maximal node with respect to C, all the edges which were removed before have at least one end-point at a higher level than y. Hence, at this stage, T^r satisfies condition (4) of Lemma 5.7. Denote x_1, x_2 the sons of y in C. According to Lemma 5.7, T^r still contains a small containment cycle C', with y the unique maximal node with respect to C', and the sons of y in C' are x_1 and x_2. The algorithm removes either edge (x_1, y) or edge (x_2, y) from T^r and in any case, according to Lemma 5.5, v is still accessible to s_i after the edge removal.

Hence, at the end of Procedure FindSpanInBig, v is accessible to s_i in T^r. Since this is true for every vertex v and every cluster S_i, the removal list L is empty. ■

Corollary 5.9. *A big-intersection hypergraph always has a feasible solution tree.*

6. Slender reduction graphs

In this section we consider the objective function of minimizing the number of vertices removed from the clusters. Note that if a vertex is removed from more than one cluster, then the objective function counts it more than once. This objective function is equal to the cardinality of the removal list returned by Algorithm FindRem (Fig. 2). We define a reduction graph which is slender (Definition 6.4) and a specific way of finding the spanning tree (FindSpanInSlender in Fig. 10) used by Algorithm FindRem for this class of graphs, to achieve a minimum number of vertices removal.

In slender reduction graphs there is no intersection set that contains another intersection set. Thus, an intersection-node is connected only to cluster-nodes and vertex-nodes. As will be shown in the following subsections, the fact that there are no edges between intersection-nodes enables us to make one decision for every intersection set X, which is contained in a cluster S_i. Either the whole intersection set is removed from the cluster or all the vertices of the intersection set stay in the cluster. Hence, we can make one decision for every pair of an intersection-node and a cluster-node which contains it. This decision is based on whether the edge between these two nodes is chosen to be in the spanning tree used by Algorithm FindRem. Since we define the weight of this edge to be equal to the cardinality of the intersection set, using a maximum weight spanning tree yields a minimum feasible removal list.

6.1. Intersection set consistency

This section includes two general lemmas. In the first lemma we prove that for every intersection set X, if a feasible removal list does not remove a vertex $v \in X$ from a cluster S_i, then it remains a feasible removal list when it does not remove all the vertices of X from S_i. In the second lemma we prove that any minimum feasible removal list is consistent with any intersection set, when consistency is defined in Definition 6.1. For every cluster which contains an intersection set, either all the vertices of the intersection will remain in the cluster or all the vertices of the intersection will be removed from the cluster.

In Section 6.2 we use these lemmas to prove that our procedure finds a minimum feasible removal list in slender reduction graphs.

Definition 6.1. Let $H = \langle V, \mathcal{S} \rangle$ be a hypergraph and let L be a removal list. L is **consistent with respect to** X for every $u, v \in X$ and every $S_i \in \mathcal{S}$, if $(u, S_i) \in L$ then $(v, S_i) \in L$. L is **AllXconsistent** if it is consistent with respect to every $X \in IS$.

Lemma 6.2. *Given a hypergraph $H = \langle V, \mathcal{S} \rangle$, where L is a feasible removal list of H and $X \in IS$ is an intersection set which satisfy $X \subseteq \bigcap_{j=1}^{l} S_{i_j}$, $S_{i_j} \in \mathcal{S}$. If there exists a vertex $v \in X$ such that $v \in \bigcap_{j=1}^{l} S_{i_j}$ in $H \backslash L$, then the following removal list $L' = L \backslash \{(u, S_{i_j}) | u \in X \backslash \{v\}, j \in \{1, \ldots, l\}\}$ is a feasible removal list and $X \subseteq \bigcap_{j=1}^{l} S_{i_j}$ in $H \backslash L'$.*

Proof. Create a new removal list $L^* = L \cup \{(u, S_i) | u \in X \backslash \{v\}, u \in S_i\}$. In $H \backslash L^*$ all the vertices in $X \backslash \{v\}$ are removed from all the clusters. Since L is a feasible removal list, every removal list which contains L is also a feasible removal list. Hence,

L^* is also a feasible removal list and $H\backslash L^*$ has a feasible solution tree, denote this tree as T^*. According to the conditions of the lemma, $v \in \cap_{j=1}^{l} S_{i_j}$ in $H\backslash L^*$. Since T^* is a feasible solution tree, it contains a path between every vertex in S_{i_j}, $\forall j \in \{1,\ldots,l\}$, to v which uses only vertices from S_{i_j}. Add to T^* a star whose center is v and its leaves are the vertices in $X\backslash\{v\}$. In the new tree there is a path between every vertex in S_{i_j} to every $u \in X$ which uses only vertices from S_{i_j}. Hence, the new solution tree is also a feasible solution tree for the hypergraph $H\backslash L'$, where L' is the removal list defined in the claim of the lemma. ■

Lemma 6.3. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, every minimum feasible removal list of H is AllXconsistent.*

Proof. Since every hypergraph has a feasible removal list, H has such a list. Therefore, it also has a minimum feasible removal list, denote such a list by L. Let $v \in X$ be the vertex which is removed the minimum number of times in L, compared with all other vertices of X. Let $S_{i_1},\ldots,S_{i_l}$ be the clusters which contain v in $H\backslash L$. According to Lemma 6.2, $L' = L\backslash\{(u, S_{i_j}) | u \in X\backslash\{v\}, j \in \{1,\ldots,l\}\}$ is also a feasible removal list. Since L is a minimum removal list, $L = L'$ and L is consistent with respect to X. Since this is true for every $X \in IS$, L is AllXconsistent. ■

6.2. Minimum removal

This section includes the definition of slender reduction graphs (Definition 6.4). Procedure FindSpanInSlender (Fig. 10) finds a maximum spanning tree in a weighted version of the slender reduction graph. When Algorithm FindRem (Fig. 2) uses this spanning tree, the output removal list has minimum cardinality.

Definition 6.4. A reduction graph G^r is **slender** if there are no $X_i, X_j \in IS$ such that $X_i \subsetneq X_j$.

Note that, in a slender reduction graph, every intersection set is a minimal intersection set. An intersection-node has only edges connecting it to cluster-nodes and to vertex-nodes. Also, there are many families of hypergraphs with slender intersection graphs, for example, if every vertex $v \in V$ belongs to at most two clusters, the corresponding reduction graph is slender.

The next lemma gives a condition for a graph to be a slender graph. This condition can be used to efficiently check, as is demonstrated in Theorem 6.6, whether a reduction graph is slender.

Lemma 6.5. *A hypergraph $H = \langle V, \mathcal{S}\rangle$ has a slender reduction graph if and only if for every three different indices $i_1, i_2, i_3 \in \{1,\ldots,m\}$ either $S_{i_1} \cap S_{i_2} \cap S_{i_3} = \phi$ or $S_{i_1} \cap S_{i_2} \cap S_{i_3} = S_{i_1} \cap S_{i_2} = S_{i_1} \cap S_{i_3} = S_{i_2} \cap S_{i_3}$.*

Proof. Suppose that H has a slender reduction graph. Consider three different indices $i_1, i_2, i_3 \in \{1,\ldots,m\}$, such that $S_{i_1} \cap S_{i_2} \cap S_{i_3} \neq \phi$. If $S_{i_1} \cap S_{i_2} \cap S_{i_3} \neq S_{i_1} \cap S_{i_2}$ then $S_{i_1} \cap S_{i_2} \cap S_{i_3} \in IS$ and $S_{i_1} \cap S_{i_2} \in IS$ are intersection sets which satisfy $S_{i_1} \cap S_{i_2} \cap S_{i_3} \subsetneq S_{i_1} \cap S_{i_2}$, contradicting that the reduction graph is slender.

Suppose that for every three different indices $i_1, i_2, i_3 \in \{1,\ldots,m\}$ either $S_{i_1} \cap S_{i_2} \cap S_{i_3} = \phi$ or $S_{i_1} \cap S_{i_2} \cap S_{i_3} = S_{i_1} \cap S_{i_2} = S_{i_1} \cap S_{i_3} = S_{i_2} \cap S_{i_3}$. Suppose by contradiction that the reduction graph is not slender, and there are $X_i, X_j \in IS$ with $X_i \subsetneq X_j$. Since $X_j \in IS$ there are two indices $i_1, i_2 \in \{1,\ldots,m\}$ such that $X_j \subseteq S_{i_1} \cap S_{i_2}$. Since $X_i \subsetneq X_j$ there is an index $i_3 \in \{1,\ldots,m\}\backslash\{i_1, i_2\}$, such that $X_i \subseteq S_{i_1} \cap S_{i_2} \cap S_{i_3}$ and $X_j \not\subset S_{i_3}$. Hence, $S_{i_1} \cap S_{i_2} \cap S_{i_3} \neq \phi$ and there is a vertex $v \in X_j$ such that $v \notin S_{i_3}$. Therefore, $v \in S_{i_1} \cap S_{i_2}$ but $v \notin S_{i_1} \cap S_{i_2} \cap S_{i_3}$, proving that $S_{i_1} \cap S_{i_2} \cap S_{i_3} \neq S_{i_1} \cap S_{i_2}$, contradicting the assumption of this case. ■

Theorem 6.6. *Checking whether a hypergraph $H = \langle V, \mathcal{S}\rangle$ has a slender reduction graph can be performed in $O(m^3|V|)$ time complexity.*

Proof. Perform the following two steps:

1. For every two different indices $i_1, i_2 \in \{1,\ldots,m\}$ calculate $S_{i_1} \cap S_{i_2}$.
2. For every three different indices $i_1, i_2, i_3 \in \{1,\ldots,m\}$ calculate $S_{i_1} \cap S_{i_2} \cap S_{i_3}$.

According to Lemma 6.5, if for every intersection of three clusters either $S_{i_1} \cap S_{i_2} \cap S_{i_3} = \phi$ or $S_{i_1} \cap S_{i_2} \cap S_{i_3} = S_{i_1} \cap S_{i_2} = S_{i_1} \cap S_{i_3} = S_{i_2} \cap S_{i_3}$, then the reduction graph is slender. Hence, checking whether a hypergraph has a slender reduction graph takes $O(m^3|V|)$ time complexity. ■

Lemma 6.7 which follows gives intuition to the way we will later choose weights for the edges in the reduction graph. The lemma states that the vertices in an intersection set X are removed from a cluster if the edge connecting the corresponding intersection-node and cluster-node is not included in the chosen spanning tree. Therefore, not choosing this edge to the tree will create $|X|$ vertices removal, and the weight of this edge is therefore set to be $|X|$, as defined in Definition 6.8. All the edges which touch vertex-nodes must be included in any spanning tree and therefore their weights are set to zero. Next, in Corollary 6.9, we prove that in any slender graph the cardinality of a feasible removal list returned by Algorithm FindRem is equal to the weight of the edges of the whole reduction graph minus the weight of the edges

FindSpanInSlender(Finds a Spanning tree In the reduction graph of Slender graphs)
input
The reduction graph $G^r = (V^r, E^r)$, where G^r is slender.
returns
A spanning tree T^r of G^r.
begin
Add weights to E^r according to Definition 6.8.
return *T^r a maximum spanning tree of G^r.*
end *FindSpanInSlender*

Fig. 10. Procedure FindSpan in slender reduction graphs.

included in the spanning tree used by the algorithm. As proved in Lemma 6.10, since there exists a minimum feasible removal list which corresponds to a spanning tree of the reduction graph, using a maximum weight spanning tree, yields a minimum feasible removal list.

Lemma 6.7. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, whose reduction graph G^r is slender, and T^r a spanning tree of G^r. For every intersection-node x which corresponds to intersection set X, and every cluster-node s_i, $X \subseteq A(T^r, s_i)$ if and only if edge (x, s_i) is contained in T^r.*

Proof. According to Property 2.4, in a reduction graph G^r, the degree of every vertex-node $v \in X$ is one, and since T^r is a spanning tree, clearly the only edge which touches this vertex-node is contained in T^r. If edge (x, s_i) is included in T^r then x is contained in s_i. In this case, the path $v \sim x \sim s_i$ uses only set-nodes included in s_i and $v \in A(T^r, s_i)$. If edge (x, s_i) is not included in T^r, either it does not exist in G^r or it is not chosen to be included in T^r. Since in slender reduction graphs there are no edges whose both endpoints are intersection-nodes, every path connecting $v \in X$ to s_i must use an edge (x, s_j) with $s_j \neq s_i$. Since s_j cannot be contained in s_i, $v \notin A(T^r, s_i)$. ■

Definition 6.8. Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, whose reduction graph G^r is slender, define weights for the edges in E^r in the following manner:

- $w(v, y) = 0$ for every vertex-node v and every set-node y, such that $(v, y) \in E^r$.
- $w(x, s_i) = |X|$ for every intersection-node x and every cluster-node s_i, such that $(x, s_i) \in E^r$, where X is the intersection set which corresponds to x.

For any subgraph G' of the reduction graph, define $\boldsymbol{w(G')}$ to be the sum of the weights of the edges in G'.

Corollary 6.9. *If Algorithm FindRem (Fig. 2) uses the spanning tree returned by Procedure FindSpanInSlender (Fig. 10) then the removal list L returned by Algorithm FindRem satisfies $|L| = w(G^r) - w(T^r)$, using the weights defined in Definition 6.8.*

Proof. According to Property 2.4, all the edges in $E(G^r)\backslash E(T^r)$ are of type (x, s_i) for an intersection-node x, which corresponds to intersection set X, and a cluster-node s_i, which corresponds to cluster S_i. According to Lemma 6.7, when $(x, s_i) \in E(G^r)\backslash E(T^r)$ Algorithm FindRem finds a removal list which contains the pairs (v, S_i) for all vertices $v \in X$, thus creating $|X|$ vertices removal. Lemma 6.7 also indicates that no other vertices removal will be suggested by Algorithm FindRem. Therefore, $|L| = \sum_{\substack{x \in IS \\ S_i \in \mathcal{S} \\ (x,s_i) \in G^r \setminus T^r}} |X| = \sum_{(x,s_i) \in G^r \setminus T^r} w(x, s_i) = w(G^r) - w(T^r)$. ■

Lemma 6.10. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, whose reduction graph G^r is slender, there exists a minimum feasible removal list L for H and a spanning tree T^r of G^r such that applying Algorithm FindRem (Fig. 2) on T^r will return L.*

Proof. Since every hypergraph has a minimum feasible removal list, according to Lemma 6.3, H has a minimum feasible removal list L which is also AllXconsistent. Therefore, for every $X \in IS$ and $S_i \in \mathcal{S}$ such that $X \subseteq S_i$, L chooses to remove either all or none of the vertices of X from S_i.

Choose the following list of edges in G^r to create T_L, which we prove to be a spanning tree of G^r:

- (v, x) for every $v \in X$ and every $X \in ISS$.
- (x, s_i) where x corresponds to intersection set X, s_i corresponds to cluster S_i and $X \subseteq S_i$ in $H\backslash L$.

Clearly, T_L touches all the vertex-nodes and intersection-nodes of V^r.

First, we prove that T_L does not contain a cycle. Suppose by contradiction that T_L contains a cycle C and, without loss of generality, consider the cycle with the minimum number of edges. By Property 2.4, the degree of every vertex-node is 1, therefore no cycle can contain an edge of type (v, x). Hence the cycle is of the form $s_{i_1} - x_{j_1} - s_{i_2} - x_{j_2} - \cdots - s_{i_k} - x_{j_k} - s_{i_1}$.

L is feasible, therefore, $H\backslash L$ has a solution tree, denote this tree by T^*.

Edges (s_{i_1}, x_{j_1}) and (x_{j_k}, s_{i_1}) are included in T_L, in $H\backslash L$ all the vertices of X_{j_1} and X_{j_k} are contained in cluster S_{i_1}. Similarly, all the vertices of $X_{j_{p-1}}$ and X_{j_p} are contained in S_{i_p}, for all $p \in \{2, \ldots, k\}$. Consider a set of k different vertices $v_{j_1}, \ldots, v_{j_k}$ such that $v_{j_p} \in X_{j_p}, \forall p \in \{1, \ldots, k\}$. Hence, $v_{j_p} \in S_{i_p} \cap S_{i_{p+1}}$ (the index calculation is performed modulo k). By the minimality of C, edge $(x_{j_p}, s_i) \notin T_L$ if $i \notin \{p, p+1\}$, and therefore $v_{j_p} \notin S_i$. Since T^* is a solution tree for $H\backslash L$, it contains a path between v_{j_k} and v_{j_1} which uses only vertices from S_{i_1}, this path cannot contain any of the vertices $v_{j_2}, \ldots, v_{j_{k-1}}$. In a similar way, there is a path in T^* between v_{j_1} and v_{j_2} which uses only vertices from S_{i_2}, and cannot contain any of the vertices $v_{j_3}, \ldots, v_{j_k}$. We continue in this manner and get that T^* contains the concatenation of paths $v_{j_k} - v_{j_1} - v_{j_2} - \cdots v_{j_{k-1}} - v_{j_k}$ which is a cycle, contradicting the fact that T^* is a tree. Hence, T_L cannot contain a cycle.

Next, we prove that T_L is connected and touches all the cluster-nodes. Suppose otherwise, since T_L is cycle-less we can add at least one edge to T_L without creating any cycle. Let T_L^+ be a spanning tree that contains all the edges of T_L and at least one other edge. Since T_L contains all edges of type (v, x) of G^r, all the edges in $T_L^+\backslash T_L$ are of the form (x, s_i). According to Theorem 3.11 and Lemma 6.7, when Algorithm FindRem uses T_L^+ it will create a feasible removal list with cardinality smaller than L, contradicting the minimality of L.

Hence, T_L is a connected, cycle-less spanning subgraph of G^r which touches all the nodes in V^r, therefore it is a spanning tree. By the way Algorithm FindRem works, if it uses spanning tree T_L it returns the removal list L. ■

Theorem 6.11. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, whose reduction graph G^r is slender, Algorithm FindRem (Fig. 2) finds a minimum feasible removal list if it uses the spanning tree returned by Procedure FindSpanInSlender (Fig. 10).*

Proof. According to Lemma 6.10, there is a minimum feasible removal list which corresponds to a spanning tree of G^r. According to Corollary 6.9, the cardinality of the vertices removal list is equal to $w(G^r) - w(T^r)$, where T^r is a spanning tree of G^r. Since Procedure FindSpanInSlender finds a maximum spanning tree of G^r, its use by Algorithm FindRem results in a list of minimum cardinality . ■

The following corollary is an interesting result concerning a hypergraph with a slender reduction graph.

Corollary 6.12. *Given a hypergraph $H = \langle V, \mathcal{S}\rangle$, whose reduction graph G^r is slender, H has a feasible solution tree if and only if its reduction graph is a tree.*

Proof. According to Theorem 6.11, the spanning tree returned by Procedure FindSpanInSlender is used by Algorithm FindRem and returns a minimum feasible removal list. According to Corollary 6.9, the number of vertices removed is $w(G^r) - w(T^r)$. Hence, the minimum number of vertices to be removed is equal to zero if and only if $w(G^r) = w(T^r)$. Since all the edges of weight zero are the only edges that touch vertex-nodes, according to Property 2.4, they must be included in any spanning tree. Therefore, the weight of the spanning tree returned by Procedure FindSpanInSlender is equal to $w(G^r)$ only when it contains all the edges of the graph, indicating that the reduction graph is a tree. ■

Note that using the conditions presented in [18] it can be proved that when the intersection graph of a given hypergraph is a tree, the hypergraph has a feasible solution tree. However, Corollary 6.12 presents a stronger result as it states that if a hypergraph has a slender reduction graph it has a feasible solution tree if and only if the corresponding intersection graph is a tree.

7. Summary and further research

For the clustered spanning tree problem, if a given hypergraph does not have a feasible solution tree, we consider removing vertices to gain feasibility. We use a special technique for the general case and prove interesting results for some specific cases of a given hypergraph.

We wish to find more special cases where minimum cardinality of the feasible removal list can be achieved. Other minimum objective functions may also be considered, for example minimizing the number of clusters which are being changed. It is also interesting to research whether similar techniques may be applied to other versions of the clustered spanning tree problem.

Another intersecting question might be to research whether it is possible to construct feasible removal lists in a way that guarantees that every cluster still contains at least one vertex, and every vertex is still contained in at least one cluster after the vertices removal.

CRediT authorship contribution statement

Nili Guttmann-Beck: Conceptualization, Formal analysis, Methodology, Writing - original draft, Reviewed version. **Roni Rozen:** Conceptualization. **Michal Stern:** Conceptualization, Formal analysis, Methodology, Writing - original draft, Reviewed version.

References

[1] S. Anily, J. Bramel, A. Hertz, A $\frac{5}{3}$-approximation algorithm for the clustered traveling salesman tour and path problems, Oper. Res. Lett. 24 (1–2) (1999) 29–35.
[2] E.M. Arkin, R. Hassin, L. Klein, Restricted delivery problems on a network, Networks 29 (4) (1997) 205–216.
[3] K.S. Booth, G.S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-tree algorithms, J. Comput. System Sci. 13 (3) (1976) 335–379, Working Papers presented at the ACM-SIGACT Symposium on the Theory of Computing (Albuquerque, N. M., 1975).
[4] U. Brandes, S. Cornelsen, B. Pampel, A. Sallaberry, Path-based supports for hypergraphs, J. Discrete Algorithms 14 (2012) 248–261.
[5] J.A. Chisman, The clustered traveling salesman problem, Comput. Oper. Res. 2 (2) (1975) 115–119.
[6] N. Christofides, Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem, Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.
[7] C. Cohen, Node Deletion for Constructing Spanning Tree with Cluster Constraints, Technical report, The Academic College of Tel Aviv Yaffo, 2014.
[8] P. Duchet, Propriété de Helly et problémes de représentation, in: Colloqu. Internat. CNRS, Problemes Conbinatoires et Theorie du Graphs, Orsay, France, Vol. 260, 1976, pp. 117–118.
[9] R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes, J. Assoc. Comput. Mach. 30 (3) (1983) 514–550.
[10] C. Flament, Hypergraphes arborés, Discrete Math. 21 (3) (1978) 223–227.
[11] N. Guttmann-Beck, R. Hassin, S. Khuller, B. Raghavachari, Approximation algorithms with bounded performance guarantees for the clustered traveling salesman problem, Algorithmica 28 (4) (2000) 422–437.
[12] N. Guttmann-Beck, E. Knaan, M. Stern, Approximation algorithms for not necessarily disjoint clustered tsp, J. Graph Algorithms Appl. 22 (2018) 555–575.
[13] N. Guttmann-Beck, Z. Sorek, M. Stern, Clustered spanning tree - conditions for feasibility, Discrete Math. Theor. Comput. Sci. 21 (1) (2019).
[14] B. Klemz, T. Mchedlidze, M. Nöllenburg, Minimum tree supports for hypergraphs and low-concurrency euler diagrams, in: Algorithm Theory – SWAT 2014, Springer International Publishing, Cham, 2014, pp. 265–276.
[15] E. Korach, M. Stern, The clustering matroid and the optimal clustering tree, Math. Program. 98 (1-3, Ser. B) (2003) 385–414.
[16] E. Korach, M. Stern, The complete optimal stars-clustering-tree problem, Discrete Appl. Math. 156 (4) (2008) 444–450.
[17] F.C.J. Lokin, Procedures for travelling salesman problems with additional constraints, European J. Oper. Res. 3 (2) (1979) 135–141.
[18] T.A. McKee, F.R. McMorris, Topics in Intersection Graph Theory, in: SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1999, p. viii+205.
[19] J.Y. Potvin, F. Guertin, A genetic algorithm for the clustered traveling salesman problem with a prespecified order on the clusters, in: Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search, in: Oper. Res./Comput. Sci. Interfaces Ser., vol. 9, Kluwer Acad. bl., Boston, MA, 1998, pp. 287–299.
[20] S. Rafaeli, D. Hutchison, A survey of key management for secure group communication, ACM Comput. Surv. 35 (2003) 309–329.
[21] R. Ramakrishnan, J. Gehrke, Database Management Systems, third ed., McGraw Hill, New York, 2002.
[22] P.J. Slater, A characterization of soft hypergraphs, Canad. Math. Bull. 21 (3) (1978) 335–337.
[23] R. Swaminathan, D.K. Wagner, On the consecutive-retrieval problem, SIAM J. Comput. 23 (2) (1994) 398–414.
[24] A.S. Tanenbaum, D.J. Wetherall, Computer Networks, fifth ed., Prentice Hall, 2011.
[25] R.E. Tarjan, M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Comput. 13 (3) (1984) 566–579.