Vertices Removal for a Feasible Clustered Travelling Salesman Problem

Master Thesis

December 9, 2020

Author:
Hadas Sayag

Supervisors:
Dr. Nili Beck
Prof. Michal Stern

Academic College of Tel-Aviv Yaffo

School of Computer Science

`https://www.mta.ac.il`

Contents

Section 1

Introduction

Let $G = (V, E)$ be a complete undirected graph with vertex set V and edge set E, where $|V| = n$, such that each edge has a positive length. The **Travelling Salesman Problem** (*TSP*) is to compute the shortest possible path that visits each vertex exactly once. This problem is NP-hard, and it is a well studied problem in graph optimization.
For *TSP*, Christofides shows in [4] an $O(n^3)$ approximation algorithm. The algorithm works on graphs in which the length of the edges satisfies the triangle inequality condition, and is a 3/2-approximation algorithm for solving *TSP*.
Hoogeveen investigates in [11] modifications of Christofides' approximation for the problem of finding a shortest Hamiltonian path. Hoogeveen considers three variants of this problem, depending on the number of prespecified endpoints of the path: zero, one, or two. The algorithm is based on the theorem that a graph contains a Hamiltonian path if and only if exactly two of its vertices have an odd degree. Therefore, if there are prespecified endpoints, they should be included in the set of vertices with odd degree. Hoogeveen algorithm is a 3/2-approximation algorithm for solving *TSP* with zero or one prespecified endpoints, and a 5/3-approximation algorithm for solving *TSP* with two prespecified endpoints.
Frederickson, Hecht and Kim present in [7] an approximation algorithm for Stacker Crane Problem, that is a variation of *TSP*.

Let $G = (V, E)$ be a complete undirected graph with vertex set V and edge set E, such that each edge has a positive length. Let $H =< V, \mathcal{S} >$ be a hypergraph, where $|V| = n, |\mathcal{S}| = m$, $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a set of not necessarily disjoint clusters, $S_i \subseteq V$, $\forall i \in \{1, \ldots, m\}$. The **Clustered Travelling Salesman Problem** (*CTSP*) is to compute the shortest possible path that visits each vertex exactly once, such that the vertices of each

cluster are visited consecutively.
The **Feasibility Clustered Travelling Salesman Problem** (*FCTSP*) is to verify whether there exists a simple path that visits each vertex exactly once, such that the vertices of each cluster are visited consecutively. In the case of non-disjoint clusters, the two problems, *CTSP* and *FCTSP*, might not have a feasible solution.
A lot of work has been done for *CTSP*, where the clusters are disjoint. In [1] Anily, Bramel and Hertz consider the Ordered Clustered Travelling Salesman Problem (*OCTSP*) with disjoint clusters. In this problem, the order of the clusters in the tour (cycle) is prespecified. The authors show a 5/3-approximation algorithm for this problem, which runs in $O(n^3)$ time. The algorithm is an adaptation of Christofides' approximation for *TSP* [4], and can also be applied to the path version of *OCTSP*.
In [8] Guttmann-Beck, Hassin, Khuller and Raghavachari deal with several variants of the *CTSP* with disjoint clusters, depending on whether or not the starting and ending vertices of a cluster have been specified. The authors describe several polynomial time approximation algorithms with a bounded performance ratio.
For hypergraphs with clusters that are not necessarily disjoint, Guttmann-Beck, Knaan and Stern present in [9] a 4-approximation algorithm for *CTSP*. The algorithm uses the PQ-Tree data structure. For special cases of the problem, the authors present algorithms with better approximation ratio.
Moreover, the authors prove the following theorem:

Theorem 1.0.1. *(Guttmann-Beck, Knaan and Stern, [9]) Let* $H = < V, \mathcal{S} >$ *be a hypergraph with vertex set* V *and* $\mathcal{S} = \{S_1, \ldots, S_m\}$ *is a set of not necessarily disjoint clusters,* $S_i \subseteq V$, $\forall i \in \{1, \ldots, m\}$, *such that its intersection graph is connected. If for each cluster* S_i, *when* $i \notin \{j, k\}$, $S_i \not\subseteq (S_j \cup S_k)$, *then* H *has a feasible solution of FCTSP only if the intersection graph is a path.*

An intersection graph of a hypergraph $H = < V, \mathcal{S} >$ is defined to be the graph in which every cluster in H is represented by a node in the intersection graph, and there is an edge between two nodes in the intersection graph if and only if the intersection of the corresponding clusters in H is not empty.

FCTSP is also known as **Consecutive Ones Property** (*COP*). A 0-1 (binary) matrix has *COP* if and only if there is a permutation of its rows so that the ones are consecutive in each column. The hypergraph given for *FCTSP* can be represented by a binary matrix, where each row represents a vertex and each column represents a cluster. The value of each cell is one if and only if the vertex represented by the current row belongs to the

cluster represented by the current column. It can easily be shown that this matrix has *COP* if and only if the given hypergraph has a feasible solution of *FCTSP*.
Booth and Lueker introduce in [2] a data structure called a PQ-Tree. Given a set $U = \{a_1, a_2, \ldots, a_m\}$, the elements of U are represented as leaves in the PQ-Tree. Given a subset $S \subset U$, the authors describe a procedure which uses a sequence of templates to create a PQ-Tree, in which the leaves included in S occur consecutively. Using this procedure, Booth and Lueker also present in [2] an algorithm which finds a feasible solution of *COP*, if such a solution exists, in linear time. Since *FCTSP* is an equivalent problem to *COP*, this algorithm can also be used to determine if the given hypergraph has a feasible solution of *FCTSP*, and if yes - to find a feasible solution. Let $H =< V, \mathcal{S} >$ be a hypergraph with vertex set V and $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a set of not necessarily disjoint clusters, $S_i \subseteq V$, $\forall i \in \{1, \ldots, m\}$. The complexity of Booth and Lueker's Algorithm is $\mathcal{O}(n + m + \sum_{1 \leq i \leq m} |S_i|)$, where $n = |V|, m = |\mathcal{S}|$. Tucker characterizes in [17] matrices with *COP*, by characterizing forbidden submatrices that cannot appear in matrices with that property. Lindzey and McConnell introduce in [12] linear-time algorithm to find a Tucker forbidden submatrix in a binary matrix that does not have *COP*.

For a less restricted problem, consider the Feasibility of Clustered Spanning Tree by Paths. Let $H =< V, \mathcal{S} >$ be a hypergraph with vertex set V and $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a set of not necessarily disjoint clusters, $S_i \subseteq V$, $\forall i \in \{1, \ldots, m\}$. The problem is to find a spanning tree on V which satisfies that each cluster induces a path, when it exists. Swaminathan and Wagner in [16] introduce a polynomial time algorithm, which constructs a solution, if one exists.
Another less restricted problem, considers the Feasibility of Clustered Spanning Tree by Trees, where the clusters are not necessarily disjoint. Let $H =< V, \mathcal{S} >$ be a hypergraph. The problem is to find a spanning tree on V which satisfies that each cluster induces a subtree, when it exists. It is proved in [5], [6], [15] and summarized by T. A. McKee and F. R. McMorris in [14], that a hypergraph has a feasible solution of Clustered Spanning Tree by Trees if and only if it satisfies the Helly property and its intersection graph is chordal. A hypergraph satisfies the Helly property if the following holds: For any set of clusters, if every pair of clusters has a common vertex, then all the clusters of the set have a common vertex. Guttmann-Beck, Sorek and Stern provide in [10] a novel and efficient algorithm which finds a feasible solution tree for H when it exists, or states that no feasible solution exists. The algorithm constructs a weighted graph and checks a special property on a maximum spanning tree for this graph.

Several applications for *FCTSP* in the field of robotics are described in [13]. A possible application is handling customer orders in a warehouse system. The orders may contain several commodities, being picked up by a motorized truck or a robot. The robot can handle several orders at the same time, with a restriction that each order must be handled consecutively. The problem is to find a shortest possible route for picking all the orders, by calculating the optimal sequence of the orders, and a minimum route in each order. In *FCTSP*, we can present a hypergraph such that each order is represented by a cluster and each commodity is represented by a vertex.
Another application described in [13] is managing Numerically Controlled machines. A number of coordinates are given, where one or more operations with different tools must be performed. The restrictions are that the Numerically Controlled machine can handle one tool at a time, and once a tool is in use all of the operations with that tool must be performed consecutively, since the cost of changing a tool is expensive. The problem is to find an optimal order of using the tools, and the minimum route for completing the operations of each tool.
Another possible application is in the area of bioinformatics, the construction of physical maps of chromosomes, described in [3]. Each chromosome is mapped by probes that correspond to unique points on the chromosome, and clones that correspond to intervals of the chromosome. Using an hybridization experiment, it can be determined if a clone contains a specific probe. The problem is to reconstruct the probes in a manner satisfying that all the probes which hybridize to one clone appear consecutively in the solution. Here, each clone is represented by a cluster and each probe is represented by a vertex.

1.1 Our Results

In our research we focus on hypergraphs with non-disjoint clusters, where there is no feasible solution of *FCTSP*. For those instances with no feasible solution, we investigate the removal of vertices from clusters, in order to achieve a feasible solution for the new set of clusters. We present several algorithms which find a removal list of vertices from appropriate clusters, in order to gain feasibility.
In the first part of our research we investigate special and different characteristics of the given hypergraph and its intersection graph. We show that there are clusters with specific attributes which we can ignore while solv-

ing the problem, and we can easily add them afterwards to get a feasible solution for the hypergraph. We look at structures of graph families for the intersection graph, including intersection graphs that are simple paths, chordless cycles, trees, stars, caterpillar trees, bipartite graphs, cliques and graphs that contain a cut edge or a cut node. We introduce algorithms which run in polynomial time and achieve minimum cardinality of vertices removal from clusters of the hypergraph, according to the structure of the intersection graph, in order to gain feasibility. These algorithms are used to remove the appropriate vertices in order to gain a new hypergraph which has a feasible solution. For the new hypergraph that has a feasible solution, in order to find a feasible solution, we can use the algorithm of Booth and Lueker [2].
In the second part of our research we introduce algorithms that are based on PQ-Trees by the algorithm of Booth and Lueker. Those algorithms find a removal list for any given hypergraph, regardless of the structure of its intersection graph. In this part, we design algorithms that run in linear time on any hypergraph.

Organization

In this work, Section 2 introduces definitions that will be used later.
Section 3 introduces minimum cardinality feasible removal list algorithms for special structures and families of the intersection graph. Section 4 presents extensions of Booth and Lueker's algorithm, that are used to find a feasible removal list for a given hypergraph. Section 5 summarizes this work and gives some possible future research.

Section 2

Definitions

Definition 2.0.1. Hypergraph Representation: Let $H =< V, \mathcal{S} >$ be a hypergraph, where $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a set of not necessarily disjoint clusters, $S_i \subseteq V$, $\forall i \in \{1, \ldots, m\}$.
The hypergraph is represented by a binary matrix, denoted by M_H, of size $n \times m$, $n = |V|, m = |\mathcal{S}|$, where each row represents a vertex $v \in V$ and each column represents a cluster $S_i \in \mathcal{S}$. The value of each cell in M_H is 1 if and only if $v \in S_i$, that is, the vertex represented by the current row belongs to the cluster represented by the current column.

Definition 2.0.2. Intersection Graph: Let $H =< V, \mathcal{S} >$ be a hypergraph with vertex set V, and $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a set of not necessarily disjoint clusters, $S_i \subseteq V$, $\forall i \in \{1, \ldots, m\}$.
The intersection graph of H, denoted by $G_{int}(\mathcal{S}) = (V_{int}, E_{int})$, is defined to be the graph in which every cluster S_i in H is represented by a node s_i in G_{int}: $V_{int} = \{s_1, \ldots, s_m\}$, and there is an edge $(s_i, s_j) \in E_{int}$ if and only if $S_i \cap S_j \neq \emptyset$.
$G_{int}(\mathcal{S})$ is represented by:

- M_G - An adjacency matrix, of size $m \times m$, $m = |\mathcal{S}|$, where each row and each column represents a node in $G_{int}(\mathcal{S})$. The value of cell $[s_i, s_j]$ in M_G is equal to $|S_i \cap S_j|$. The value of a cell is greater than zero if and only if there exists an edge between the corresponding nodes, s_i and s_j, in $G_{int}(\mathcal{S})$. Note that M_G is a symmetric matrix.

- V_D - A degree vector, of size $m = |\mathcal{S}|$. The value of $V_D[s_i]$ is the degree of node s_i in $G_{int}(\mathcal{S})$.

- V_S - A clusters' sizes vector, of size $m = |\mathcal{S}|$. The value of $V_S[s_i]$ is $|S_i|$.

Property 2.0.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph with intersection graph $G_{int}(\mathcal{S})$. A leaf s_i in $G_{int}(\mathcal{S})$ corresponds to a cluster S_i which intersects with exactly one cluster.*

Property 2.0.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph. We assume that for $i \neq j$, $S_i \neq S_j$. If two clusters are the same, we can remove one of those clusters.*

Definition 2.0.5. $nc(v)$**:** Let $H =< V, \mathcal{S} >$ be a hypergraph. The number of clusters that vertex v belongs to is denoted by $nc(v)$.

Definition 2.0.6. Induced Hypergraph: Let $H =< V, \mathcal{S} >$ be a hypergraph with vertex set V and $\mathcal{S}$ a set of clusters, and let $\mathcal{S}' \subseteq \mathcal{S}$ be a set of clusters. The induced hypergraph $H[\mathcal{S}'] =< V(\mathcal{S}'), \mathcal{S}' >$ is a hypergraph with vertex set $V(\mathcal{S}')$ such that $V(\mathcal{S}') = \bigcup_{S_i \in \mathcal{S}'} S_i$, and whose cluster set is $\mathcal{S}'$.

Property 2.0.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph, with intersection graph $G_{int}(\mathcal{S})$. The induced graph $G_{int}(\mathcal{S})[\bigcup_{S_i \in \mathcal{S}'} S_i]$, for $\mathcal{S}' \subseteq \mathcal{S}$, is the intersection graph of $H[\mathcal{S}']$, and therefore can be denoted by $G_{int}(\mathcal{S}')$.*

Definition 2.0.8. Induced solution: Let $H =< V, \mathcal{S} >$ be a hypergraph with a feasible solution P of *FCTSP*. $P[X]$, for $X \subseteq V$, is the collection of subpaths induced on the vertices of X. $P[\mathcal{S}']$, for $\mathcal{S}' \subseteq \mathcal{S}$, is the collection of subpaths induced on the vertices of the clusters in $\mathcal{S}'$.

Definition 2.0.9. Containing Cluster: Let $H =< V, \mathcal{S} >$ be a hypergraph. Cluster $S_i \in \mathcal{S}$ is a containing cluster, if every $S_j \in (\mathcal{S} \setminus \{S_i\})$ satisfies $S_j \subseteq S_i$.

Definition 2.0.10. Contained Cluster: Let $H =< V, \mathcal{S} >$ be a hypergraph. Cluster $S_i \in \mathcal{S}$ is a contained cluster, if there exists a cluster $S_j \in (\mathcal{S} \setminus \{S_i\})$ such that $S_i \subseteq S_j$, and for any other cluster $S_k \in (\mathcal{S} \setminus \{S_i, S_j\})$, $S_i \cap S_k = \emptyset$.

Definition 2.0.11. Contained Component: Let $H =< V, \mathcal{S} >$ be a hypergraph. A contained component is a collection of clusters $\mathcal{S}' \subsetneq \mathcal{S}$ whose clusters create one connected component in $G_{int}(\mathcal{S})$ such that $V(\mathcal{S}') \subseteq S_i$, $S_i \in \mathcal{S} \setminus \mathcal{S}'$, and $\forall S_j \in \mathcal{S}', S_k \in \mathcal{S} \setminus (\mathcal{S}' \cup \{S_i\})$ satisfy $S_j \cap S_k = \emptyset$.

Definition 2.0.12. Pairwise Uncontained Clusters: Clusters $S_1, S_2, \ldots, S_m$ satisfy the Pairwise Uncontained Clusters property, denoted by *PUC*, if $\forall i \neq j \neq k$ satisfy $S_k \nsubseteq S_i \cup S_j$. That is, $\forall i \neq j \neq k$ there exists $v \in S_k$ such that $v \notin S_i \cup S_j$.

Definition 2.0.13. Helly Property: Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a family of subsets. $\mathcal{S}$ satisfies the Helly Property if the following holds: For every $\mathcal{S}' \subseteq \mathcal{S}$, if every pair members of $\mathcal{S}'$ has a common element, then all the members of $\mathcal{S}'$ have a common element. In other words, if every $S_i, S_j \in \mathcal{S}'$ satisfy $S_i \cap S_j \neq \emptyset$, then $\bigcap_{S_i \in \mathcal{S}'} S_i \neq \emptyset$.

Definition 2.0.14. Removal List: Let $H =< V, \mathcal{S} >$ be a hypergraph. A removal list L for H is a list containing pairs $(Vertex, Cluster)$, where each pair in the list indicates which vertex to remove from the corresponding cluster in H. Adding $(\{v_1, \ldots, v_i\}, S_j)$ to L is a shortened writing which denotes adding pairs $(v_1, S_j), \ldots, (v_i, S_j)$ to L. $|L| =$ The total number of pairs in L. The hypergraph received after removing L from H is denoted by $H \setminus L$, and its intersection graph is denoted by $G_{int}(\mathcal{S} \setminus L)$.

Definition 2.0.15. Feasible Removal List: Let $H =< V, \mathcal{S} >$ be a hypergraph, and L a removal list for H. L is a feasible removal list if $H \setminus L$ has a feasible solution of *FCTSP*.

Definition 2.0.16. Minimum Cardinality Feasible Removal List: Let $H =< V, \mathcal{S} >$ be a hypergraph, and L a feasible removal list for H. The target function that we will look at is the minimum cardinality, that is, minimizing the total number of changes that are made.
We define - $minChanges(S_i) =$ The number of vertices deleted from cluster S_i. If the same vertex is deleted several times, each change is counted separately.

The target function for minimum cardinality is $min \sum_{i=1}^{m} minChanges(S_i)$.
L^* is a minimum cardinality feasible removal list if $L^* = \underset{(v \in V, S_i \in \mathcal{S})}{\operatorname{argmin}} \{|L| \mid L \text{ is a feasible removal list}\}$.

Section 3

Intersection Graph Algorithms

In this section we present theorems and algorithms that are based on the structure of the intersection graph. We first introduce general theorems regarding hypergraphs and their intersection graph. Then we consider structures of common graph families for the intersection graph, and present several algorithms which find a removal list for the hypergraph, in order to achieve feasibility. After applying these algorithms on the hypergraph, we can use the algorithm of Booth and Lueker [2] to find a feasible solution for the new hypergraph. At the end of this section, we characterize conditions for a feasible solution with a specific cluster at an end of it.

3.1 General Theorems

In this section we first present conditions for a feasible solution of induced hypergraphs. Then we consider cases in which the hypergraph has containing clusters, contained clusters or contained components. We show that in these cases, we can solve the problem without the containing cluster, contained cluster or contained component, and then add them to the solution, in order to get a feasible solution for the hypergraph. At the end of the section we present Theorem 3.1.11 that deals with a special structure of the intersection graph for which there is no feasible solution. These results will be used in the following sections, and can also be used while checking if there exists a feasible solution of *FCTSP*.

Theorem 3.1.1. *Let $H =< V, \mathcal{S} >$ be a hypergraph. Creating an intersection graph for H can be done in $\mathcal{O}(nm^2)$ time complexity, where $n = |V|, m = |\mathcal{S}|$.*

Proof. First we initialize every cell in M_G, V_D, V_S to zero, in $\mathcal{O}(m^2)$ time complexity.
For every row in M_H which represents a vertex $v \in V$:

- We increment the value of $V_S[s_i]$, for every S_i which contains v, in $\mathcal{O}(m)$ time complexity for every $v \in V$. Therefore, calculating V_S can be done in $\mathcal{O}(nm)$ time complexity.
- We increment the value of $V_D[s_i]$, for every two clusters S_i, S_j which contain v and satisfy $M_G[s_i, s_j] = 0$. This represents adding a new edge to the intersection graph. The time complexity of this step is $\mathcal{O}(m^2)$ for every $v \in V$, and therefore, calculating V_D can be done in $\mathcal{O}(nm^2)$ time complexity.
- We increment the value of $M_G[s_i, s_j]$, for every two clusters S_i, S_j which contain v, in $\mathcal{O}(m^2)$ time complexity for every $v \in V$. Therefore, calculating M_G can be done in $\mathcal{O}(nm^2)$ time complexity.

Hence, the total time complexity of creating an intersection graph for H is $\mathcal{O}(nm^2)$. □

Lemma 3.1.2. *Let $H =< V, \mathcal{S} >$ be a hypergraph whose intersection graph $G_{int}(\mathcal{S})$ contains at least two disjoint connected components. If every connected component has a feasible solution of FCTSP, then H has a feasible solution of FCTSP.*

Proof. Since each connected component has a feasible solution, we can connect paths that are feasible solutions for each one of the connected components in a proper way to create a path that is a feasible solution for H. □

Following Theorem 3.1.2, in the algorithms described in this section, we assume that the intersection graph is connected.

Lemma 3.1.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph with a feasible solution P of FCTSP. For every $X = \bigcap_{S_i \in \mathcal{S}'} S_i, \mathcal{S}' \subseteq \mathcal{S}$ in H, $P[X]$ is a consecutive subpath.*

Proof. Suppose that $X = \bigcap_{S_i \in \mathcal{S}'} S_i, \mathcal{S}' \subseteq \mathcal{S}$, and let $\{v, u\} \subseteq X$. In this case, $\forall S_i \in \mathcal{S}'$ it follows that $\{v, u\} \subseteq S_i$. Since P is a feasible solution of *FCTSP*, P contains a path between v and u, such that $\forall S_i \in \mathcal{S}'$ all the vertices in the path are in S_i. Therefore, P contains a path between v and u, such that all the vertices in this path are in X. This follows for every $\{v, u\} \subseteq X$. Hence, $P[X]$ is a consecutive subpath. □

Lemma 3.1.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph with a feasible solution P of FCTSP. For every $\mathcal{S}' \subseteq \mathcal{S}$, if $G_{int}(\mathcal{S}')$ is connected, then $P[\mathcal{S}']$ is a consecutive path.*

Proof. Let k be the number of clusters in $\mathcal{S}'$. The proof is by induction on k.
For $k = 1$, denote $S_1 \in \mathcal{S}'$. since P is a feasible solution, $P[S_1]$ is consecutive. Suppose the assumption of the lemma is correct for $k-1$, and now prove it for k. Denote $S_1, S_2, \ldots, S_k \in \mathcal{S}'$.
Without loss of generality, remove S_k from $\mathcal{S}'$. Denote $\mathcal{S}'' = \mathcal{S}' \setminus \{S_k\}$. If $\mathcal{S}''$ is connected, then vertices $v \in S_k$ that satisfy $nc(v) = 1$ appear consecutively and adjacent to the intersections of S_k with other clusters. If $\mathcal{S}''$ is not connected, since the assumption of the lemma is correct for $k-1$, then for each connected component $C \in \mathcal{S}''$, $P[C]$ is a consecutive path. For each $C \in \mathcal{S}''$, there exists at least one cluster $S_i \in C$ such that $S_i \cap S_k \neq \emptyset$, follows from the connectivity of $G_{int}(\mathcal{S}')$. Since P is a feasible solution, $P[S_k]$ is consecutive, and therefore all the intersections of S_k with other clusters are connected only by vertices of S_k. Hence, $\mathcal{S}'$ is a collection of consecutive paths that are connected by S_k, and thus $P[\mathcal{S}']$ is a consecutive path. □

Theorem 3.1.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If H has a feasible solution P of FCTSP, then for every $\mathcal{S}' \subseteq \mathcal{S}$, $P[\mathcal{S}']$ is a feasible solution for $H[\mathcal{S}']$.*

Proof. Assume that the intersection graph of H is connected. Denote by $P' = P[\mathcal{S}']$ the induced solution for $H[\mathcal{S}']$. According to Lemma 3.1.4, P' is a consecutive path. The induced solution does not change the order of the vertices in the solution, therefore, $P'[S_i] = P[S_i]$. That is, for every $S_i \in \mathcal{S}'$, the entire subpath is in P' and is consecutive, and therefore every cluster in P' is consecutive. Hence, $P[\mathcal{S}']$ is a feasible solution for $H[\mathcal{S}']$. If the intersection graph is not connected, this proof is correct for every connected component, and hence, by Theorem 3.1.2, the theorem is proved. □

Corollary 3.1.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If there exists $\mathcal{S}' \subseteq \mathcal{S}$ such that $H[\mathcal{S}']$ does not have a feasible solution of FCTSP, then H does not have a feasible solution of FCTSP.*

Lemma 3.1.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph that has a feasible solution P of FCTSP. For every set of vertices $U \subseteq S_i$, such that $\forall u \in U$, $nc(u) = 1$, and a removal list $L = \{(U, S_i) \mid S_i \in \mathcal{S}\}$, $P[V \setminus U]$ is a feasible solution for $H \setminus L$.*

Proof. P is a feasible solution, therefore $P[S_i]$ is a consecutive path, and $P[U]$ is a collection of subpaths of this path. Remove from $P[S_i]$ all the subpaths of $P[U]$ and concatenate the remaining subpaths into one path. Denote by P' the induced solution of $P[V \setminus U]$. Clearly, $P'[S_i \setminus U]$ is a consecutive path. Since $U \cap S_j = \emptyset$, for every $j \neq i$, it also holds that

$P'[S_j] = P[S_j]$. Therefore, every cluster in P' is consecutive. Hence, $P[V \setminus U]$ is a feasible solution for hypergraph $H \setminus L$. $\square$

Theorem 3.1.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph, and let $S_i \in \mathcal{S}$ be a containing cluster in H. $H[\mathcal{S} \setminus \{S_i\}]$ has a feasible solution of FCTSP if and only if H has a feasible solution of FCTSP.*

Proof. Suppose $H[\mathcal{S} \setminus \{S_i\}]$ has a feasible solution of *FCTSP*. Denote $\mathcal{S}' = \mathcal{S} \setminus \{S_i\}, H' =< V', \mathcal{S}' >, V' = \bigcup_{S_j \in \mathcal{S}'} S_j$, and P' a feasible solution for H'. Arrange all the vertices of $S_i \setminus \bigcup_{S_j \in \mathcal{S}'} S_j$ in a path, denote this path by P''. Concatenate P' and P'', denote the new path by P. For every $S_j \in \mathcal{S}', P[S_j] = P'[S_j]$, and therefore is a consecutive subpath. P is a path spanning all the vertices of S_i, and since $\forall S_j \in \mathcal{S}'$, $S_j \subseteq S_i$, $P[S_i]$ is a consecutive subpath. Therefore, P is a feasible solution for H.
The correctness of the opposite direction is obtained directly from Theorem 3.1.5. $\square$

Theorem 3.1.9. *Let $H =< V, \mathcal{S} >$ be a hypergraph, and let $S_i \in \mathcal{S}$ be a contained cluster in H. $H[\mathcal{S} \setminus \{S_i\}]$ has a feasible solution of FCTSP if and only if H has a feasible solution of FCTSP.*

Proof. Suppose $H[\mathcal{S} \setminus \{S_i\}]$ has a feasible solution of *FCTSP*, denote this solution by P'. Since S_i is a contained cluster, there exists $S_k \in \mathcal{S}$ such that $S_i \subseteq S_k$. Remove the vertices of $S_k \cap S_i$ from P', except for vertex $v \in (S_k \cap S_i)$ that appears first in P', denote the new path by P''. S_i intersects only with S_k, that is, the vertices of $S_k \cap S_i$ belong to these two clusters only. Therefore, the order of the vertices of $S_j \in (\mathcal{S} \setminus \{S_i, S_k\})$ in P'' is the same as their order in P'. According to Lemma 3.1.7, P'' is a feasible solution.
Replace v in P'' with a subpath that spans all the vertices of S_i, denote this path by P. $S_i = S_k \cap S_i$, that is, all the vertices of the added subpath are contained in S_k, therefore, S_k is consecutive in P. P is a path spanning all the vertices of S_i, and S_i is concatenated consecutively to it. Furthermore, for $S_j \in (\mathcal{S} \setminus \{S_i, S_k\})$, $P[S_j] = P'[S_j]$, and therefore is a consecutive subpath. Therefore, P is a feasible solution for H.
The correctness of the opposite direction is obtained directly from Theorem 3.1.5. $\square$

Theorem 3.1.10. *Let $H =< V, \mathcal{S} >$ be a hypergraph, and let $\mathcal{S}' \subseteq \mathcal{S}$ be a contained component in H, and suppose there is a feasible solution for $H[\mathcal{S}']$. $H[\mathcal{S} \setminus \mathcal{S}']$ has a feasible solution of FCTSP if and only if H has a feasible solution of FCTSP.*

Proof. Suppose $H[\mathcal{S} \setminus \mathcal{S}']$ has a feasible solution of *FCTSP*, denote this solution by P'. Since $\mathcal{S}'$ is a contained component, there exists $S_k \in \mathcal{S}$ such that $\forall S_i \in \mathcal{S}'$, $S_i \subseteq S_k$. $\forall S_i \in \mathcal{S}'$, remove the vertices of $S_k \cap S_i$ from P', except for vertex $v \in (S_k \cap (\bigcup_{S_i \in \mathcal{S}'} S_i)$ that appears first in P', denote the new path by P''. $\forall S_i \in \mathcal{S}'$, S_i intersects only with S_k or with other clusters of the contained component. Therefore, the order of the vertices of $S_j \in (\mathcal{S} \setminus (\mathcal{S}' \cup \{S_k\}))$ in P'' is the same as their order in P'. According to Lemma 3.1.7, P'' is a feasible solution.
Replace v in P'' with a subpath that spans all the vertices of the contained component $\mathcal{S}'$. Denote this path by P. $\forall S_i \in \mathcal{S}'$, $S_i \subseteq S_k$, that is, all the vertices of the added subpath are contained in S_k, therefore, S_k is consecutive in P. P is a path spanning all the vertices of $\mathcal{S}'$, and according to the assumptions of the theorem, the subpath added instead of v is a feasible solution for $H[\mathcal{S}']$. Furthermore, for $S_j \in (\mathcal{S} \setminus (\mathcal{S}' \cup \{S_k\}))$, $P[S_j] = P'[S_j]$, and therefore is a consecutive subpath. Therefore, P is a feasible solution for H.
The correctness of the opposite direction is obtained directly from Theorem 3.1.5. □

Following Theorems 3.1.8, 3.1.9, 3.1.10, in the algorithms described in this section, we assume that H does not include containing clusters, contained clusters and contained components.

The following theorem focuses on a special structure of the intersection graph with no feasible solution for the hypergraph.

Theorem 3.1.11. *Let $H =< V, \mathcal{S} >$ be a hypergraph with intersection graph $G_{int}(\mathcal{S})$. If there exists clusters $\{S_i, S_{j_1}, S_{j_2}, S_{j_3}\} \in \mathcal{S}$ such that:*

1. $S_i \cap S_{j_1} \neq \emptyset, S_i \cap S_{j_2} \neq \emptyset, S_i \cap S_{j_3} \neq \emptyset$
2. $S_{j_1} \nsubseteq S_i, S_{j_2} \nsubseteq S_i, S_{j_3} \nsubseteq S_i$
3. $S_{j_1} \cap S_{j_2} = \emptyset, S_{j_2} \cap S_{j_3} = \emptyset, S_{j_1} \cap S_{j_3} = \emptyset$

Then H does not have a feasible solution of FCTSP.
(See Figure 3.1 for the induced intersection graph on $\{S_i, S_{j_1}, S_{j_2}, S_{j_3}\}$.)

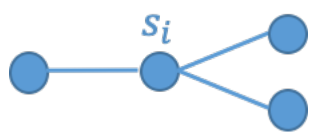

Figure 3.1: The intersection graph of $\{S_i, S_{j_1}, S_{j_2}, S_{j_3}\}$.

Proof. Denote by $X_1 = S_i \cap S_{j_1}, X_2 = S_i \cap S_{j_2}, X_3 = S_i \cap S_{j_3}$. Suppose by contradiction that H has a feasible solution P. According to Theorem 3.1.3, the induced solutions for X_1, X_2, X_3 are consecutive subpaths, denote these subpaths by P_1, P_2, P_3, respectively. Since $S_i \cap S_{j_1} \neq \emptyset, S_i \cap S_{j_2} \neq \emptyset, S_i \cap S_{j_3} \neq \emptyset$, then P_1, P_2, P_3 are non-empty. Also, since $S_{j_1} \cap S_{j_2} = \emptyset, S_{j_2} \cap S_{j_3} = \emptyset, S_{j_1} \cap S_{j_3} = \emptyset$, then P_1, P_2, P_3 are pairwise vertex disjoint.
Without loss of generality, assume that subpaths P_1, P_2, P_3 are ordered, not necessarily consecutively, in P. X_1, X_2, X_3 are contained in S_i, and according to Theorem 3.1.3, $P[S_i]$ is a consecutive subpath. Therefore, every vertex v that appears between P_1, P_2, P_3 in P, satisfies $v \in S_i$.
By condition 2, consider vertex $t \in (S_{j_2} \setminus S_i)$. Vertex t cannot appear in $P[S_i]$. According to the assumptions of the theorem, there exists $u \in X_1$ that satisfies $u \notin S_{j_2}$, and there exists $w \in X_3$ that satisfies $w \notin S_{j_2}$, therefore, S_{j_2} is not consecutive in the solution, as shown in Figure 3.2. Contradicting the fact that P is a feasible solution of *FCTSP*. $\square$

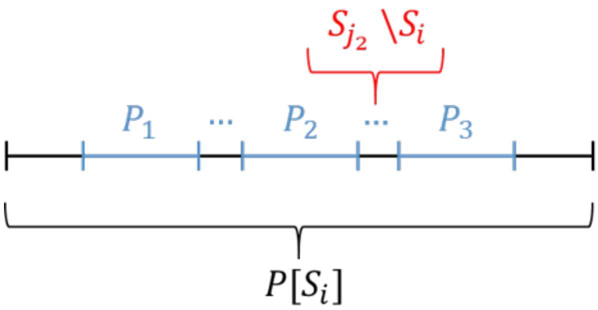

Figure 3.2: Subpath $P[S_i]$ in P, showing where $S_{j_2} \setminus S_i$ should appear in a feasible solution.

3.2 Path Graphs

This section introduces hypergraphs whose intersection graphs are simple paths (see Definition 3.2.1). For these hypergraphs we show that a feasible solution of *FCTSP* always exists, and present Algorithm FindPath (see Figure 3.3) where the input is a hypergraph whose intersection graph is a simple path, and its output is a feasible solution of *FCTSP*.

Definition 3.2.1. Simple path: A path which traverses each node exactly once. This is one connected component such that two nodes have degree 1, and are at the ends of the path, and every other node has degree 2.

Lemma 3.2.2. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a simple path, then there are no cliques of size ≥ 3 in $G_{int}(\mathcal{S})$.*

Proof. Suppose by contradiction that $G_{int}(\mathcal{S})$ is a path graph and there is a clique of size ≥ 3 in $G_{int}(\mathcal{S})$. Denote the nodes of the clique by $s_{i_1}, s_{i_2}, s_{i_3}$. The degree of every node of the clique ≥ 2, since each node is connected to the other two nodes of the clique. We consider two cases:

- $G_{int}(\mathcal{S})$ is a path with exactly three nodes. In this case, all the nodes of $G_{int}(\mathcal{S})$, s_{i_1}, s_{i_2} and s_{i_3}, are part of the clique. Therefore, the degree of every node in $G_{int}(\mathcal{S})$ is 2, contradicting the definition of a path graph.

- $G_{int}(\mathcal{S})$ is a path with more than three nodes. In this case, since $G_{int}(\mathcal{S})$ is a connected graph, there exists a fourth node s_j which is connected to at least one node of the clique, s_{i_1}, s_{i_2} or s_{i_3}. Suppose, without loss of generality, that s_j is connected to s_{i_1}. In this case, s_{i_1} is connected to three other nodes s_{i_2}, s_{i_3}, s_j. Hence, s_{i_1} has degree ≥ 3, contradicting the definition of a path graph.

□

Corollary 3.2.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a simple path, then $\forall v \in V, nc(v) \leq 2$.*

Proof. If $\exists v \in V$ such that $nc(v) \geq 3$, then there exist $S_{i_1}, S_{i_2}, S_{i_3} \in \mathcal{S}$, such that $v \in S_{i_1} \cap S_{i_2} \cap S_{i_3}$. In this case, $G_{int}(\mathcal{S})$ includes a clique of size 3 on the corresponding nodes, contradicting Lemma 3.2.2. □

Lemma 3.2.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph with a feasible solution P of FCTSP, whose intersection graph $G_{int}(\mathcal{S})$ contains a simple path $s_1 - s_2 - \ldots - s_k$. For every $v \in S_{i_1}, u \in S_{i_x}$, $x \in \{1, \ldots, k\}$, the subpath between v and u in P contains only vertices from $\bigcup_{j=1}^{x} S_{i_j}$.*

Proof. The proof is by induction on x.
For $x = 1$, the path between v and u in P contains only vertices from S_{i_1}.
Suppose the assumption of the lemma is correct for $x - 1$, and now prove it for x.
Let $v \in S_{i_1}$, $u \in S_{i_x}$. Since $G_{int}(\mathcal{S})$ contains the edge $(s_{i_{x-1}}, s_{i_x})$, it follows that $S_{i_{x-1}} \cap S_{i_x} \neq \emptyset$. Let $w \in S_{i_{x-1}} \cap S_{i_x}$. Since the assumption of the lemma is correct for $x - 1$, P contains a path between v and w, denote it by P_1, such that $P_1 \subseteq P[\bigcup_{j=1}^{x-1} S_{i_j}]$. Since P is a feasible solution, it contains a path between w and u, denote it by P_2, such that $P_2 \subseteq P[S_{i_x}]$. The union of P_1 and P_2 creates a path, contained in P, between v and u. Since P is a path, according to Property 3.4.3, this is the only path between v and u.

The vertices in P_1 and P_2 are contained in $\bigcup_{i=j}^{x-1} S_{i_j} \cup S_{i_x}$, and thus the path between v and u uses only vertices from $\bigcup_{j=1}^{x} S_{i_j}$. $\square$

Algorithm 1 FindPath: An Algorithm to find a feasible solution for a hypergraph whose intersection graph is a simple path

---

function FINDPATH()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ is a simple path.
The clusters of the simple path are denoted according to their order in the path, by $S_1, S_2, \ldots, S_m$.
Output:
A feasible solution P of *FCTSP* for H.

begin
  Initialize an empty path P.
  Add to P a path spanning the vertices in $S_1 \setminus S_2$.
  for $i = 1, \ldots, m-2$:
    Concatenate to the end of P a path spanning the vertices in $S_i \cap S_{i+1}$.
    Concatenate to the end of P a path spanning the vertices in $S_{i+1} \setminus (S_i \cup S_{i+2})$.
  end for
  Concatenate to the end of P a path spanning the vertices in $S_{m-1} \cap S_m$.
  Concatenate to the end of P a path spanning the vertices in $S_m \setminus S_{m-1}$.
  return P.
end function

Figure 3.3: Algorithm FindPath

Theorem 3.2.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a simple path, then H has a feasible solution of FCTSP.*

Proof. We Prove that Algorithm FindPath in Figure 3.3 finds a feasible solution P for H. According to Lemma 3.2.3, $\forall v \in V, nc(v) \leq 2$. Therefore, we can divide H into the following disjoint subclusters:
$\{S_1 \setminus S_2, S_1 \cap S_2, S_2 \setminus (S_1 \cup S_3), S_2 \cap S_3, S_3 \setminus (S_2 \cup S_4), \ldots\}$.
The algorithm constructs the path P according to this division. It is easy to see that P spans all the vertices of H.
In addition, every cluster appears consecutively in the path:

- The first cluster S_1: This cluster is divided to two subclusters $S_1 \setminus S_2$ and $S_1 \cap S_2$, that appear consecutively in the path.

- $S_i \in \mathcal{S} \setminus \{S_1, S_m\}$: This cluster is divided to three subclusters $S_i \cap S_{i-1}$, $S_i \setminus (S_{i-1} \cup S_{i+1})$ and $S_i \cap S_{i+1}$, that appear consecutively in the path, as shown in Figure 3.4.

- The last cluster S_m: This cluster is divided to two subclusters $S_{m-1} \cap S_m$ and $S_m \setminus S_{m-1}$, that appear consecutively in the path.

Hence, the algorithm constructs a feasible solution of *FCTSP* for H. □

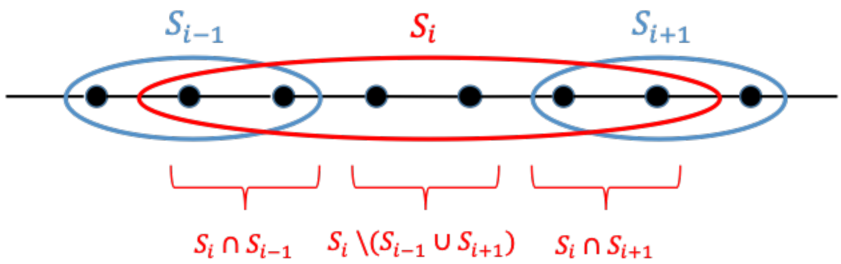

Figure 3.4: The subclusters of cluster S_i in the proof of Theorem 3.2.5.

Theorem 3.2.6. *Verifying whether the intersection graph of a hypergraph $H =< V, \mathcal{S} >$ is a simple path can be performed in $\mathcal{O}(m)$ time complexity, where $m = |\mathcal{S}|$.*

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a simple path, we parse V_D in $\mathcal{O}(m)$ time complexity, and verify that there are exactly two nodes with degree 1, and all other nodes have degree 2. The next step is to verify that $G_{int}(\mathcal{S})$ is connected, using DFS algorithm. The general time complexity of DFS algorithm is $\mathcal{O}(|V| + |E|)$. However, in our graph, we already know that $|E| = \mathcal{O}(|V|) = \mathcal{O}(m)$, and therefore performing DFS on this graph requires $\mathcal{O}(m)$ time complexity. Hence, the total time complexity of verifying that $G_{int}(\mathcal{S})$ is a simple path is $\mathcal{O}(m)$. □

Theorem 3.2.7. *The time complexity of Algorithm FindPath is $\mathcal{O}(m^2+nm)$, $n = |V|, m = |\mathcal{S}|$.*

Proof. In order to find node s_1, we parse V_D in $\mathcal{O}(m)$ time complexity, and search for a node which has degree 1. In order to find s_2, that is, the node which is a neighbour of s_1, we process the row of s_1 in M_G, looking for a cell which has a positive value. This can be done in $\mathcal{O}(m)$ time complexity. After that, we calculate $S_1 \setminus S_2$ and $S_1 \cap S_2$, by processing the columns of S_1, S_2 in M_H, and looking for cells with value 1 only in the column of S_1 or in both columns, respectively. The time complexity of this step is $\mathcal{O}(n)$. Similarly, we find for every cluster S_{i+1}, $i \in \{1, \ldots, m-2\}$, the clusters it intersects with, S_i and S_{i+2}, using M_G, and process the columns of S_i, S_{i+1}, S_{i+2} in M_H in order to calculate $S_i \cap S_{i+1}$ and $S_{i+1} \setminus (S_i \cup S_{i+2})$. The time complexity for calculating which subclusters are relevant to each cluster is $\mathcal{O}(m)$, and the time complexity for calculating the vertices in each subcluster is $\mathcal{O}(n)$. Since there are m clusters to process, the total time complexity of Algorithm FindPath is $\mathcal{O}(m(m+n)) = \mathcal{O}(m^2 + nm)$. □

3.3 Chordless Cycle Graphs

This section introduces hypergraphs whose intersection graphs are chordless cycles (see Definition 3.3.1 and Figure 3.5). For these hypergraphs we show that if the size of the cycle of the intersection graph is $m \geq 4$, then there is no feasible solution of *FCTSP*. We present Algorithm DelCycle (see Figure 3.6) where the input is a hypergraph whose intersection graph is a chordless cycle of size $m \geq 4$, and its output is a minimum cardinality feasible removal list.
A cycle graph with exactly 3 nodes is also a clique of size 3, and will be handled by the algorithm for a clique of size $m = 3$. Therefore, the following section will only deal with chordless cycle graphs with at least 4 nodes.

Definition 3.3.1. Chordless cycle: A connected path where every node has degree 2.

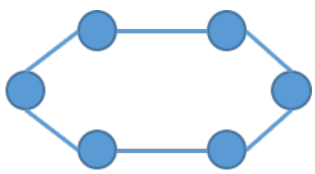

Figure 3.5: An example of a chordless cycle graph with $m = 6$.

Property 3.3.2. *Removing any edge from a chordless cycle, creates a simple path.*

Lemma 3.3.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle, then each cluster in H has a non-empty intersection with exactly two other clusters.*

Proof. This lemma is derived directly from the definition of a chordless cycle. □

Property 3.3.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$, then there are no cliques of size ≥ 3 in $G_{int}(\mathcal{S})$.*

Corollary 3.3.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$, then $\forall v \in V, nc(v) \leq 2$.*

Proof. If $\exists v \in V$ such that $nc(v) \geq 3$, then there exist $S_{i_1}, S_{i_2}, S_{i_3} \in \mathcal{S}$, such that $v \in S_{i_1} \cap S_{i_2} \cap S_{i_3}$. In this case, $G_{int}(\mathcal{S})$ includes a clique of size 3 on the corresponding nodes, contradicting Lemma 3.3.4. □

Corollary 3.3.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$, then there are no $S_i, S_j \in \mathcal{S}$ such that $S_j \subseteq S_i$.*

Proof. Suppose by contradiction that there exists a cluster S_j that is contained in another cluster S_i. According to Corollary 3.3.3, each cluster in H intersects with exactly two other clusters. Therefore, S_j intersects with some other cluster S_k. Since $S_j \subseteq S_i$, $S_j \cap S_k \subseteq S_i$, and therefore $S_i \cap S_j \cap S_k \neq \emptyset$, contradicting Lemma 3.3.4. □

Theorem 3.3.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$, then H does not have a feasible solution of FCTSP.*

Proof. Suppose by contradiction that H has a feasible solution P of *FCTSP*. According to the assumption of the theorem, $G_{int}(\mathcal{S})$ is a chordless cycle. Let the nodes in the cycle be $s_1 - s_2 - \ldots - s_m - s_1$. Denote the labels of the clusters by $S_1, \ldots, S_m$, according to the order of the corresponding nodes in $G_{int}(\mathcal{S})$.
Let $\mathcal{S}' = \{S_1, S_2, S_3\}$. According to Theorem 3.1.5, $P[\mathcal{S}']$ is a feasible solution for $H[\mathcal{S}']$, denote this solution by P'. $G_{int}(\mathcal{S}')$ is a simple path $s_1 - s_2 - s_3$. According to Lemma 3.1.4, P' is a consecutive subpath. According to Lemma 3.2.4, for every $v \in S_1, u \in S_3$, the subpath in P' between v and u contains only vertices from S_1, S_2, S_3. Since $S_1 \cap S_3 = \emptyset$, P' contains at least one vertex from S_2.
Let $\mathcal{S}'' = \{S_1, S_m, S_{m-1}, \ldots, S_4, S_3\}$. According to Theorem 3.1.5, $P[\mathcal{S}'']$ is a feasible solution for $H[\mathcal{S}'']$, denote this solution by P''. $G_{int}(\mathcal{S}'')$ is a simple path $s_1 - s_m - s_{m-1} - \ldots - s_4 - s_3$. According to Lemma 3.1.4, P'' is a consecutive subpath. According to Lemma 3.2.4, for every $v \in$

$S_1, u \in S_3$, the subpath in P'' between v and u contains only vertices from $S_1, S_m, S_{m-1}, \ldots, S_4, S_3$. Since $S_1 \cap S_3 = \emptyset$, P'' contains at least one vertex from $S_4 \cup \ldots \cup S_m$.
$S_2 \cap (S_4 \cup \ldots \cup S_m) = \emptyset$, therefore P' and P'' are two disjoint subpaths, except for the two endpoints. Hence, there are two disjoint subpaths between v and u in P, contradicting Property 3.4.3 of a simple path. □

Lemma 3.3.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$. Removing edge (s_{i_1}, s_{i_2}) from the cycle in $G_{int}(\mathcal{S})$ requires removing intersection $S_{i_1} \cap S_{i_2}$ from H.*

Proof. According to Corollary 3.3.5, each intersection of clusters from H is the intersection of exactly 2 clusters. In addition, according to Definition 2.0.2, there is an edge $(s_{i_1}, s_{i_2}) \in G_{int}(\mathcal{S})$ if and only if $S_{i_1} \cap S_{i_2} \neq \emptyset$. Therefore, removing an edge from the cycle in $G_{int}(\mathcal{S})$ requires choosing $i \neq j$ such that $S_{i_1} \cap S_{i_2} \neq \emptyset$, and removing $S_{i_1} \cap S_{i_2}$ either from S_{i_1} or from S_{i_2}. □

Algorithm 2 DelCycle: An Algorithm to find a minimum cardinality feasible removal list for a hypergraph whose intersection graph is a chordless cycle

function DelCycle()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$.
The clusters of the chordless cycle are denoted according to their order in the cycle, by $S_1, S_2, \ldots, S_m$.
Output:
A minimum cardinality feasible removal list L for H.

begin
  for each cluster $S_i \in \mathcal{S}$:
    Calculate $S_i^{\cap} = S_i \cap S_{(i+1) \bmod m}$.
  end for
  Find $i^* = \underset{1 \leq i \leq m}{\operatorname{argmin}}\{|S_i^{\cap}|\}$.
  $L = (S_{i^*}^{\cap}, S_{i^*})$.
  return L.
end function

Figure 3.6: Algorithm DelCycle

Example 3.3.9. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m = 4$. Figure 3.7 shows an example for Algorithm DelCycle. Intersection $S_4 \cap S_1$ is with minimum cardinality, and therefore $S_4 \cap S_1$ will be removed from S_4. Note that $S_4 \cap S_1$ may be removed from S_1 instead of S_4.*
After the removal, $P = (1, 2, 12, 3, 4, 5, 13, 14, 6, 7, 8, 9, 10, 11, 15)$ is a feasible solution of FCTSP. The removed section is marked by red in the figure.

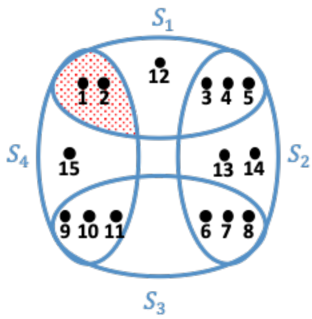

Figure 3.7: Example 3.3.9.

Theorem 3.3.10. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$. Algorithm DelCycle finds a feasible removal list for H.*

Proof. Algorithm DelCycle finds a removal list which removes exactly one edge from $G_{int}(\mathcal{S})$. According to Property 3.3.2, removing one edge from the cycle changes the intersection graph into a simple path. Hence, according to Theorem 3.2.5, $H \setminus L$ has a feasible solution of *FCTSP*. □

Theorem 3.3.11. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a chordless cycle of size $m \geq 4$. Algorithm DelCycle finds a minimum cardinality feasible removal list for H.*

Proof. According to Theorem 3.3.7, H has no feasible solution of *FCTSP* when the intersection graph is a chordless cycle of size $m \geq 4$. Therefore, every feasible removal list has to contain at least one edge from the intersection graph. According to Lemma 3.3.8, that requires choosing $i \neq j$, $i, j \in \{1, \ldots, m\}$, such that $S_i \cap S_j \neq \emptyset$ and removing $S_i \cap S_j$ either from S_i or from S_j. Algorithm DelCycle finds such a pair with minimum cardinality of $S_i \cap S_j$, for those pairs that verify $S_i \cap S_j \neq \emptyset$. Therefore, the algorithm finds a minimum cardinality feasible removal list for H, where the intersection graph of it is a chordless cycle of size $m \geq 4$. □

Theorem 3.3.12. *Verifying whether the intersection graph of a hypergraph $H =< V, \mathcal{S} >$ is a chordless cycle can be performed in $\mathcal{O}(m)$ time complexity, where $m = |\mathcal{S}|$.*

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a chordless cycle, we parse V_D in $\mathcal{O}(m)$ time complexity, and verify that the degree of each node is 2. The next step is to verify that $G_{int}(\mathcal{S})$ is connected, using DFS algorithm. In our graph, we already know that $|E| \leq 2m = \mathcal{O}(m)$, and therefore performing DFS on this graph requires $\mathcal{O}(m)$ time complexity. Hence, the total time complexity of verifying that $G_{int}(\mathcal{S})$ is a chordless cycle is $\mathcal{O}(m)$. □

Theorem 3.3.13. *The time complexity of Algorithm DelCycle is $\mathcal{O}(m^2+n)$, $n = |V|, m = |\mathcal{S}|$.*

Proof. Since each cell in M_G stores the size of the appropriate intersection, finding an intersection with a minimum cardinality can be done by scanning M_G, looking for a cell with a minimum positive value, in $\mathcal{O}(m^2)$ time complexity. Denote by S_i, S_{i+1} the clusters of the minimum intersection. In order to find the vertices in this intersection, we process the columns of S_i, S_{i+1} in M_H, looking for columns with value 1 in both columns, which can be done in $\mathcal{O}(n)$ time complexity. Hence, the total time complexity of Algorithm DelCycle is $\mathcal{O}(m^2 + n)$. □

Remark 3.3.14. *In the following sections, we will also consider removing $S_i \setminus S_j$ from S_i. However, in the case of an intersection graph which is a chordless cycle, there is always a minimum removal list which is not $S_i \setminus S_j$. Since every cluster in H intersects with exactly two other clusters, then for every cluster $S_i \in \mathcal{S}$, $S_i \cap S_{i-1} \subseteq S_i \setminus S_{i+1}$ and $S_i \cap S_{i+1} \subseteq S_i \setminus S_{i-1}$. Hence, removing $S_i \cap S_{i+1}$ or $S_i \cap S_{i-1}$ is at least as good as removing $S_i \setminus S_{i+1}$ or $S_i \setminus S_{i-1}$.*

3.4 Tree Graphs

This section introduces hypergraphs whose intersection graphs are trees (see Definition 3.4.1 and Figure 3.8). For these hypergraphs we show that if the intersection graph is a tree that is not a simple path with no contained clusters, then there is no feasible solution of *FCTSP*. We will use this theorem in the following sections, that include special cases of trees.

Definition 3.4.1. Tree: A connected component that does not contain any cycles.

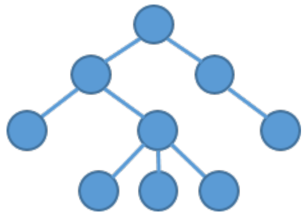

Figure 3.8: An example of a tree graph.

Property 3.4.2. *Removing an edge from a tree splits it into two connected components, each one of them is a tree.*

Property 3.4.3. *In a tree (or a simple path), there is exactly one path between every pair of nodes.*

Property 3.4.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a tree, then there are no cliques of size ≥ 3 in $G_{int}(\mathcal{S})$.*

Corollary 3.4.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph. If its intersection graph $G_{int}(\mathcal{S})$ is a tree, then $\forall v \in V, nc(v) \leq 2$.*

Proof. Suppose by contradiction that $\exists v \in V$ such that $nc(v) \geq 3$. Therefore, there exist $S_{i_1}, S_{i_2}, S_{i_3} \in \mathcal{S}$, such that $v \in S_{i_1} \cap S_{i_2} \cap S_{i_3}$. In this case, $G_{int}(\mathcal{S})$ includes a clique of size 3 on the corresponding nodes, contradicting Lemma 3.4.4. □

Corollary 3.4.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph whose intersection graph $G_{int}(\mathcal{S})$ is a tree. If there are no contained clusters in H, then there are no $S_i, S_j \in \mathcal{S}$ such that $S_j \subseteq S_i$.*

Proof. Suppose by contradiction that there exist clusters $S_i, S_j \in \mathcal{S}$ such that $S_j \subseteq S_i$. Since there are no contained clusters in H, according to Definition 2.0.10, there exists some other cluster $S_k \in (\mathcal{S} \setminus \{S_i, S_j\})$ such that $S_j \cap S_k \neq \emptyset$. Since $S_j \subseteq S_i$, $S_j \cap S_k \subseteq S_i$, and therefore $S_i \cap S_j \cap S_k \neq \emptyset$, contradicting Lemma 3.4.4. □

Lemma 3.4.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph whose intersection graph $G_{int}(\mathcal{S})$ is a tree. If $G_{int}(\mathcal{S})$ is not a simple path, there exists a node in $G_{int}(\mathcal{S})$ with degree ≥ 3.*

Proof. If $G_{int}(\mathcal{S})$ does not have a node with degree 3, it is a simple path, according to Definition 3.2.1. □

Theorem 3.4.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph whose intersection graph $G_{int}(\mathcal{S})$ is a tree. If $G_{int}(\mathcal{S})$ is not a simple path, and there are no contained clusters in $G_{int}(\mathcal{S})$, then H does not have a feasible solution of FCTSP.*

Proof. According to Lemma 3.4.7, there exists a node s_i with degree ≥ 3 in $G_{int}(\mathcal{S})$. s_i has at least 3 neighbours, $s_{j_1}, s_{j_2}, s_{j_3}$. Hence, $S_i \cap S_{j_1} \neq \emptyset, S_i \cap S_{j_2} \neq \emptyset, S_i \cap S_{j_3} \neq \emptyset$. Since there are no contained clusters, and according to Corollary 3.4.6, $S_{j_1} \not\subseteq S_i, S_{j_2} \not\subseteq S_i, S_{j_3} \not\subseteq S_i$. Since $G_{int}(\mathcal{S})$ is a tree and does not contain cycles of size 3, then according to Lemma 3.4.4, $S_{j_1} \cap S_{j_2} = \emptyset, S_{j_2} \cap S_{j_3} = \emptyset, S_{j_1} \cap S_{j_3} = \emptyset$. Hence, according to Theorem 3.1.11, H does not have a feasible solution of *FCTSP*. $\square$

Example 3.4.9. *Let $H =< V, \mathcal{S} >$ be a hypergraph whose intersection graph $G_{int}(\mathcal{S})$ is a tree. Note that if H has contained clusters then it might have a feasible solution of FCTSP even though $G_{int}(\mathcal{S})$ is a tree that is not a simple path. An example is given in Figure 3.9. For this hypergraph, $P = (5, 4, 2, 1, 3, 6)$ is a feasible solution of FCTSP.*

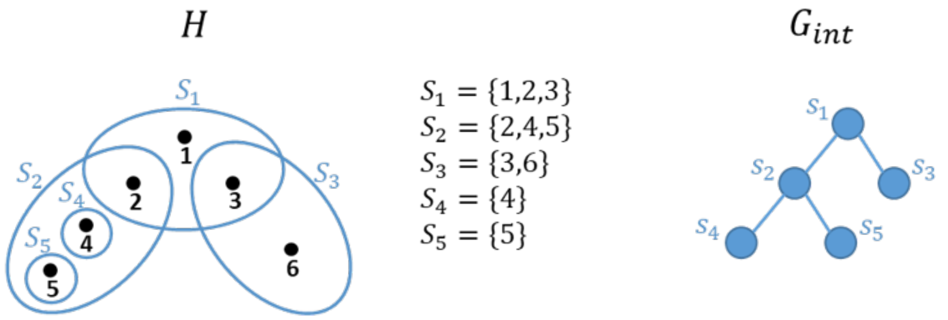

Figure 3.9: Example 3.4.9.

3.4.1 Star Graphs

This section introduces hypergraphs whose intersection graphs are stars (see Definition 3.4.10 and Figure 3.10). A star is a special case of a tree. Since a star is a special case of a tree, according to Theorem 3.4.8, if there are no contained clusters, when the intersection graph of H is a star with $k \geq 3$ leaves, it has no feasible solution of *FCTSP*. We present Algorithm DelStar (see Figure 3.11) where the input is a hypergraph whose intersection graph is a star, and its output is a minimum cardinality feasible removal list.
A star with $k \leq 2$ leaves is also a simple path, and was handled in a previous section. Therefore, the following section will only deal with stars with at least 3 leaves.

Definition 3.4.10. Star: A tree with one internal node and k leaves, for $k \geq 1$. A star is $K_{1,k}$ (see Definition 3.5.1).

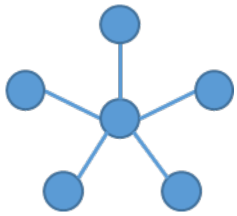

Figure 3.10: An example of a star graph with $k = 5$ leaves.

Algorithm 3 DelStar: An Algorithm to find a minimum cardinality feasible removal list for a hypergraph whose intersection graph is a star

---

function DELSTAR()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S}) = (V_{int}, E_{int})$ is a star with k leaves. Denote by s^* the center of the star.
Output:
A minimum cardinality feasible removal list L for H.

begin
    Initialize an empty list L.
    Initialize a list $Leaves = V_{int} \setminus s^*$.
    for each $s_j \in Leaves$:
        Calculate ${S_j}^{\cap} = S_j \cap S^*$.
        Calculate ${S_j}^{D} = S_j \setminus S^*$.
    end for
    for $i = 1, \ldots, k-2$:
        Find $j^* = \underset{s_j \in Leaves}{\text{argmin}} \{{S_j}^{\cap}, {S_j}^{D}\}$.
        if $|{S_{j^*}}^{\cap}| \leq |{S_{j^*}}^{D}|$:
            Add $({S_{j^*}}^{\cap}, S_{j^*})$ to L.
        else
            Add $({S_{j^*}}^{D}, S_{j^*})$ to L.
        end if
        Remove node s_{j^*} from $Leaves$.
    end for
    return L.
end function

Figure 3.11: Algorithm DelStar

Example 3.4.11. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with $k = 4$. Figure 3.12 shows an example for Algorithm DelStar. A minimum cardinality removal list is removing $S_1 \setminus S^*$ from S_1, transforming S_1 to a contained cluster, and also removing $S_2 \cap S^*$ from S_2, transforming S_2 to a singleton node. After the removal, $P = (14, 15, 12, 13, 1, 2, 4, 8, 9, 10, 11, 5, 6, 7)$ is a feasible solution of FCTSP. Note that after the vertices removal, vertex 3 is not contained in any of the clusters and therefore is not in the solution path. The removed sections are marked by red in the figure.*

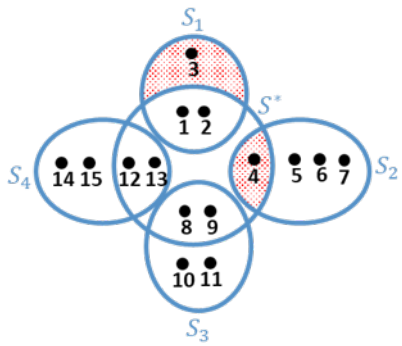

Figure 3.12: Example 3.4.11.

Lemma 3.4.12. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star. If the number of leaves in $G_{int}(\mathcal{S})$ is $k \leq 2$, then $G_{int}(\mathcal{S})$ is a simple path.*

Proof. If there are 2 leaves, then the degree of s^* is 2. Furthermore, the degree of each leaf is 1. Therefore, according to Definition 3.2.1, $G_{int}(\mathcal{S})$ is a simple path, where s^* is at the center of this path. □

Theorem 3.4.13. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star. Algorithm DelStar finds a feasible removal list for H.*

Proof. Let $G_{int}(\mathcal{S} \setminus L)$ be the intersection graph for $H \setminus L$, where L is the output of Algorithm DelStar. At the end of the algorithm, $k - 2$ nodes in $G_{int}(\mathcal{S} \setminus L)$ meet one of the following conditions:

1. For every $S_j \in \mathcal{S}$ such that $S_j \cap S^*$ was removed from S_j, node s_j becomes a singleton node, not connected to any other node in $G_{int}(\mathcal{S} \setminus L)$. Note that $(S_j \setminus (S_j \cap S^*)) \cap S_i = \emptyset$ for every $S_i \in (\mathcal{S} \setminus \{S_j\})$.

2. For every $S_j \in \mathcal{S}$ such that $S_j \setminus S^*$ was removed from S_j, node s_j becomes a contained cluster. Note that $(S_j \setminus (S_j \setminus S^*)) \cap S_i = \emptyset$ for every $S_i \in (\mathcal{S} \setminus \{S_j, S^*\})$ and $(S_j \setminus (S_j \setminus S^*)) \subseteq S^*$.

Therefore, at the end of the algorithm, $G_{int}(\mathcal{S} \setminus L)$ contains at most three nodes that do not correspond to contained clusters or singleton clusters. These three nodes include s^* and two more nodes that were leaves in $G_{int}(\mathcal{S})$. Therefore, according to Lemma 3.4.12, $G_{int}(\mathcal{S} \setminus L)$ is a simple path, and according to Theorems 3.1.2, 3.1.9, 3.2.5, $H \setminus L$ has a feasible solution of *FCTSP*. □

Theorem 3.4.14. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star. Algorithm DelStar finds a minimum cardinality feasible removal list for H.*

Proof. According to Lemma 3.4.12 and to Theorem 3.4.8, if there are no contained clusters in $G_{int}(\mathcal{S})$, and if H has a feasible solution, then the center of the star can be connected to at most two nodes. Therefore, if $k \geq 3$, every feasible removal list has to remove vertices from at least $k-2$ clusters, transforming each one of these clusters either to a singleton cluster or to a contained cluster.
Since every leaf s_j is connected with an edge only to s^*, deleting $S_j \cap S^*$ or $S_j \setminus S^*$ from S_j does not change any other cluster in H. Hence, we can calculate the removal from each cluster independently from the other clusters. The minimum removal from a cluster S_j is removing the minimum of $\{|S_j \cap S^*|, |S_j \setminus S^*|\}$ from S_j.
Algorithm DelStar finds a removal list which removes minimum number of vertices from exactly $k-2$ clusters as described above, and therefore the algorithm finds a minimum cardinality feasible removal list for H, where the intersection graph of it is a star. □

Theorem 3.4.15. *Verifying whether the intersection graph of a hypergraph $H =< V, \mathcal{S} >$ is a star with k leaves can be performed in $\mathcal{O}(m)$ time complexity, where $m = |\mathcal{S}|$.*

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a star with $k > 2$ leaves, we parse V_D in $\mathcal{O}(m)$ time complexity, and check that there is exactly one node whose degree is > 2, and all other nodes have degree 1. The next step is to verify that $G_{int}(\mathcal{S})$ is connected, using DFS algorithm. In our graph, we already know that $|E| \leq 2m = \mathcal{O}(m)$, and therefore performing DFS on this graph requires $\mathcal{O}(m)$ time complexity. Hence, the total time complexity of verifying that $G_{int}(\mathcal{S})$ is a star with k leaves is $\mathcal{O}(m)$. □

Theorem 3.4.16. *The time complexity of Algorithm DelStar is $\mathcal{O}(mn)$, $n = |V|, m = |\mathcal{S}|$.*

Proof. In order to find node s^*, the center of the star, we parse V_D in $\mathcal{O}(m)$ time complexity, and search for a node which has degree > 2. The sizes of the intersections between the leaves and s^* are stored in the row of s^* in M_G. The sizes of the differences between the leaves and s^* can be found by calculating the difference between the size of the appropriate cluster, stored in V_S, and the size of the intersection with s^*, stored in M_G, in $\mathcal{O}(m)$ time complexity.
Next we need to find $k-2$ clusters with minimum cardinality of intersection or difference. Instead, we find the 2 clusters with maximum cardinality. This can be done in $\mathcal{O}(m)$ time complexity, and marks the two clusters which remain connected to s^*. Then, for each of the other $k-2$ clusters, we choose the minimum removal, by comparing the sizes of the intersection and difference of the corresponding cluster with s^*. In order to find the vertices in this removal, we process the columns of S^* and the appropriate cluster in M_H, in $\mathcal{O}(n)$ time complexity. Since there are $\mathcal{O}(m)$ leaves, the time complexity of this step is $\mathcal{O}(mn)$, and therefore, the total time complexity of Algorithm DelStar is $\mathcal{O}(mn)$. □

Remark 3.4.17. *Removing $S_j \cap S^*$ from S_j or from S^* contributes the same number of vertices removal. Therefore, there is a minimum cardinality feasible removal list which does not remove $S_j \cap S^*$ from S^*. Also, since for every S_j, whose corresponding node in the intersection graph is a leaf, $S^* \setminus S_j$ contains $S^* \cap S_k$ for some $k \neq j$, there is a minimum cardinality feasible removal list which does not remove $S^* \setminus S_j$ from S^*. Therefore, all removals can be done not from S^*.*

3.4.2 Star with Paths Graphs

This section introduces hypergraphs whose intersection graphs are stars with paths (see Definition 3.4.18 and Figure 3.13). A star with paths is a special case of a tree, and an extension of a star. Since a star with paths is a special case of a tree, according to Theorem 3.4.8, if there are no contained clusters, when the intersection graph of H is a star with paths and there are $k \geq 3$ paths, it has no feasible solution of *FCTSP*. We present Algorithm DelStarWithPaths (see Figure 3.14), where the input is a hypergraph whose intersection graph is a star with paths, and its output is a minimum cardinality feasible removal list. Algorithm DelStarWithPaths is similar to Algorithm DelStar, with one change. In Algorithm DelStar we remove vertices from the clusters that correspond to leaves in the intersection graph. In algorithm

DelStarWithPaths we remove vertices from clusters that correspond to the nodes that are neighbours of the center of the star in the intersection graph. A star with $k \leq 2$ paths is also a simple path, and was handled in a previous section. Therefore, the following section will only deal with stars with at least 3 paths.

Definition 3.4.18. Star with Paths: A tree with one internal node and k paths (instead of leaves), for $k \geq 1$, such that each path is connected to the internal node at one end of the path.

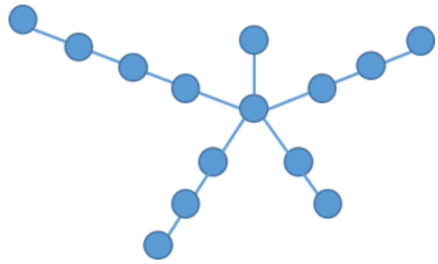

Figure 3.13: An example of a star with paths graph with $k = 5$ paths.

Algorithm 4 DelStarWithPaths: An Algorithm to find a minimum cardinality feasible removal list for a hypergraph whose intersection graph is a star with paths

---

function DELSTARWITHPATHS()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ is a star with k paths. Denote by s^* the center of the star.
Output:
A minimum cardinality feasible removal list L for H.

begin
  Initialize an empty list L.
  Denote $\mathcal{S}' = S^* \cup \{S' \mid S' \in \mathcal{S}, S' \cap S^* \neq \emptyset\}$.
  Denote $H' = H[\mathcal{S}']$.
  $L = DelStar(H')$.
  return L.
end function

Figure 3.14: Algorithm DelStarWithPaths

Example 3.4.19. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with $k = 4$ paths. Figure 3.15 shows an example for Algorithm DelStarWithPaths. A possible minimum cardinality removal list is removing $S_1 \setminus S^*$ from S_1, transforming s_1 to a contained cluster in s^*, that is not connected to the rest of its path. Furthermore, removing $S_3 \cap S^*$ from S_3, transforming s_3 to a cluster that is not connected to the center s^*, but is connected to the rest of its path. After the removals, $P = (19, 14, 15, 12, 13, 1, 2, 3, 4, 8, 9, 10, 11, 16, 20, 5, 6, 7, 17, 18)$ is a feasible solution of FCTSP. Note that after the vertices removal, vertex 21 is not contained in any of the clusters and therefore is not in the solution path. The removed sections are marked by red in the figure.*

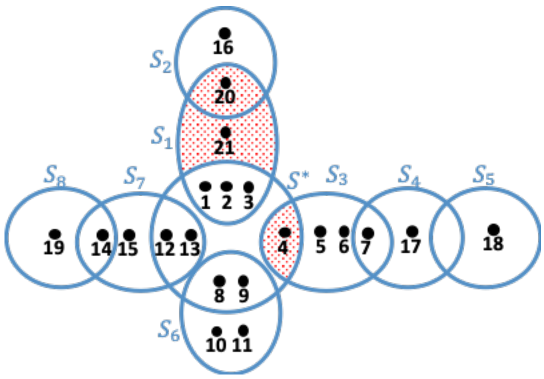

Figure 3.15: Example 3.4.19.

Theorem 3.4.20. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with paths. Algorithm DelStarWithPaths finds a feasible removal list for H.*

Proof. Let $G_{int}(\mathcal{S} \setminus L)$ be the intersection graph for $H \setminus L$, where L is the output of Algorithm DelStarWithPaths. Similarly to the proof of Theorem 3.4.13, at the end of the algorithm we get $k-2$ paths that are disconnected from s^*. After the removal, $k-2$ nodes in $G_{int}(\mathcal{S} \setminus L)$, which are neighbours of s^*, correspond to clusters that meet one of the following conditions:

1. For every $S_j \mid s_j$ is a neighbour of s^*, such that $S_j \cap S^*$ was removed from S_j in Algorithm DelStar, node s_j is disconnected from s^* in $G_{int}(\mathcal{S} \setminus L)$, but is connected to the rest of its path.

2. For every $S_j \mid s_j$ is a neighbour of s^*, such that $S_j \setminus S^*$ was removed from S_j in Algorithm DelStar, node s_j becomes a contained cluster, that is $(S_j \setminus (S_j \setminus S^*)) \subseteq S^*$, and s_j is disconnected from the rest of its path.

In $G_{int}(\mathcal{S} \setminus L)$, s^* has at most two neighbours. Each neighbour is a part of a simple path. It is easy to show that s^* and these two paths create a connected component which is a simple path, where s^* is at the center of this path. Therefore, according to Theorems 3.1.2, 3.1.9 and 3.2.5, $H \setminus L$ has a feasible solution of *FCTSP*. $\square$

Theorem 3.4.21. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with paths. Algorithm DelStarWithPaths finds a minimum cardinality feasible removal list for H.*

Proof. Based on the proof of a star graph of Theorem 3.4.14, if $k \geq 3$, every feasible removal list has to remove vertices from at least $k-2$ clusters. Note that if H does not have a feasible solution, removing vertices from a cluster that corresponds to a node that is not a neighbour of s^* does not affect the feasibility, since s^* is still part of a structure that does not have a feasible solution, according to Theorem 3.1.11. Therefore, we need to remove vertices from at least $k-2$ clusters, that correspond to nodes in the intersection graph that are neighbours of s^*. Hence, we need to transform the star which is composed from s^* and its neighbours. According to Theorem 3.4.14, the algorithm finds a minimum cardinality feasible removal list for this star, which is also a minimum cardinality feasible removal list for H, where the intersection graph of it is a star with paths. $\square$

Theorem 3.4.22. *Verifying whether the intersection graph of a hypergraph $H =< V, \mathcal{S} >$ is a star with k paths can be performed in $\mathcal{O}(m)$ time complexity, where $m = |\mathcal{S}|$.*

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a star with $k > 2$ paths, we parse V_D in $\mathcal{O}(m)$ time complexity, and check that there is exactly one node whose degree is > 2, and all other nodes have degree 1 or 2. We check that $G_{int}(\mathcal{S})$ is a connected graph that contains no cycles, using DFS algorithm. In our graph, we already know that $|E| \leq 3m = \mathcal{O}(m)$, and therefore performing DFS on this graph requires $\mathcal{O}(m)$ time complexity. Hence, the total time complexity of verifying that $G_{int}(\mathcal{S})$ is a star with k paths is $\mathcal{O}(m)$. $\square$

Theorem 3.4.23. *The time complexity of Algorithm DelStarWithPaths is $\mathcal{O}(nm^2)$, $n = |V|, m = |\mathcal{S}|$.*

Proof. In order to find node s^*, the center of the star, we parse V_D in $\mathcal{O}(m)$ time complexity, and search for a node which has degree > 2. Then, we create the induced hypergraph of S^* and its neighbours, by initializing a new matrix ${M_H}'$, which contains the columns of S^* and its neighbours. The neighbours can be found by processing the row of s^* in M_G, looking for cells with a positive value, in $\mathcal{O}(m)$ time complexity, and initializing $M_H{}'$ can be done in $\mathcal{O}(nm)$ time complexity. According to Theorem 3.1.1, creating the induced intersection graph for H' can be performed in $\mathcal{O}(nm^2)$ time complexity. In the last step of the algorithm, we run Algorithm DelStar on the induced hypergraph and intersection graph. According to Theorem 3.4.16, this can be performed in $\mathcal{O}(mn)$ time complexity. Hence, the total time complexity of Algorithm DelStarWithPaths is $\mathcal{O}(nm^2)$. □

3.4.3 Caterpillar Tree Graphs

This section introduces hypergraphs whose intersection graphs are caterpillar trees (see Definition 3.4.24 and Figure 3.16). A caterpillar tree is a special case of a tree. Since a caterpillar tree is a special case of a tree, according to Theorem 3.4.8, if there are no contained clusters, when the intersection graph of H is a caterpillar tree which is not a simple path, it has no feasible solution of *FCTSP*. We present Algorithm DelCaterpillar (see Figure 3.18), which is a dynamic programming algorithm where the input is a hypergraph whose intersection graph is a caterpillar tree, and its output is a feasible removal list.

Definition 3.4.24. Caterpillar Tree: A tree that containes a path, called the central path, and all other nodes are with distance of at most 1 from the central path. The nodes that are not part of the central path are denoted as leaves.

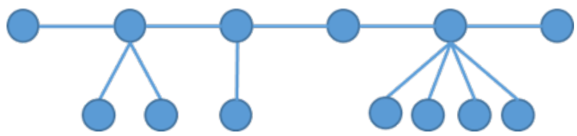

Figure 3.16: An example of a caterpillar tree graph.

Property 3.4.25. *In a caterpillar tree, removing all the leaves and the edges connecting them to the central path creates a path.*

Property 3.4.26. *The central path of a caterpillar tree includes all the nodes with degree ≥ 2 in the graph.*

Property 3.4.27. *A caterpillar tree is a collection of stars, such that their centers are connected by a path. The centers of the stars are the nodes with degree ≥ 2 in the graph.*

Example 3.4.28. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a caterpillar tree with central path $s_1, \ldots, s_k$. An algorithm which considers independently the nodes in the central path of the tree might not give a minimum cardinality result. Consider, for example, the intersection graph described in Figure 3.17. For simplicity, the cost of edge (s_i, s_j) is marked as the number of vertices in $|S_i \cap S_j|$. If we consider only node s_1 we might choose to remove $S_1 \cap S_4$ from S_1 (removal of 2 vertices). If we consider only node s_2 we might choose to remove $S_2 \cap S_5$ from S_2 (removal of 2 vertices). However, removing $S_1 \cap S_2$ from S_2 gives a better overall solution, with a total removal of 3 vertices.*

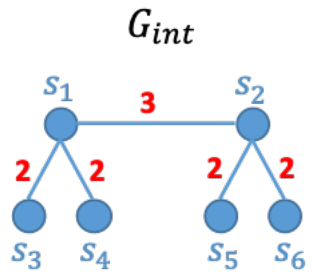

Figure 3.17: Example 3.4.28.

In the case of a caterpillar tree graph, we present a dynamic programming algorithm, which finds a feasible removal list. The dynamic programming algorithm gives better results than an algorithm which considers independently the nodes in the central path of the tree. However, this algorithm might not give a minimum cardinality result, as we show later in this section in Example 3.4.31.

The Dynamic Programming Functions used in the Algorithm:
$G(i)$ = The cost for a feasible removal list of $H[S_1, \ldots, S_i]$, after removing edge (s_i, s_{i+1}).
$F(i)$ = The cost for a feasible removal list of $H[S_1, \ldots, S_i, S_i \cap S_{i+1}]$, when edge (s_i, s_{i+1}) stays in the graph.
We denote:

- $L^{S_i, S_{i+1}}$ = The feasible removal list for disconnecting s_i from s_{i+1} in $G_{int}(\mathcal{S})$, or transforming one of those clusters to a contained cluster in the other cluster. That is, a feasible removal list with a minimum number of vertices considering $\{(S_i \cap S_{i+1}, S_i)\}, \{(S_i \setminus S_{i+1}, S_i)\}, \{(S_{i+1} \setminus S_i, S_{i+1})\}$.

- $cost(s_i, s_j) = |L^{S_i,S_j}|$, that is, the cost of removing edge (s_i, s_j) from $G_{int}(\mathcal{S})$.
- $S(i) = S_i \cup \{S' \mid S' \in (\mathcal{S} \setminus \{S_i\}), S' \cap S_i \neq \emptyset\}$, that is, the subset of S_i and the clusters which correspond to neighbours of s_i in $G_{int}(\mathcal{S})$.
- $H(i) = H[S(i)]$, for $1 \leq i \leq m$.
- $H^{--}(i) = H[S(i) \setminus \{S_{i-1}, S_{i+1}\}]$, for $2 \leq i \leq m-1$. That is, $H(i)$ without clusters S_{i-1}, S_{i+1}, so that edges $(s_i, s_{i-1}), (s_i, s_{i+1})$ are not included in its intersection graph.
- $H^{-+}(i) = H[S(i) \setminus \{S_{i-1}\}]$, for $2 \leq i \leq m$. That is, $H(i)$ without cluster S_{i-1}, so that edge (s_i, s_{i-1}) is not included in its intersection graph. In the intersection graph of $H^{-+}(i)$, for $2 \leq i \leq m-1$, we set $cost(s_i, s_{i+1}) = \infty$.
- $H^{+-}(i) = H[S(i) \setminus \{S_{i+1}\}]$, for $1 \leq i \leq m-1$. That is, $H(i)$ without cluster S_{i+1}, so that edge (s_i, s_{i+1}) is not included in its intersection graph. In the intersection graph of $H^{+-}(i)$, for $2 \leq i \leq m-1$, we set $cost(s_i, s_{i-1}) = \infty$.
- $H^{++}(i) = H[S(i)]$, for $1 \leq i \leq m$. That is, $H^{++}(i)$ includes both neighbours that are part of the central path, s_{i-1} and s_{i+1}, if those neighbours exist in the intersection graph. In the intersection graph of $H^{++}(i)$, for $1 \leq i \leq m-1$, we set $cost(s_i, s_{i+1}) = \infty$, and for $2 \leq i \leq m$, we set $cost(s_i, s_{i-1}) = \infty$.
- $L^{--}(i), L^{-+}(i), L^{+-}(i), L^{++}(i)$ are removal lists for $H^{--}(i), H^{-+}(i), H^{+-}(i), H^{++}(i)$, respectively.

Note that if edges (s_i, s_{i-1}) or (s_i, s_{i+1}), which are part of the central path, are included in the intersection graph of $H(i)$, the algorithm assigns cost ∞ to those edges, to ensure that these edges will not be removed by Algorithm DelStar.
We also denote:

$$X_g(i) = \begin{cases} - & G(i-1) + |L^{--}(i)| < F(i-1) + |L^{+-}(i)| \\ + & \text{otherwise} \end{cases}$$

$$X_f(i) = \begin{cases} - & G(i-1) + |L^{-+}(i)| < F(i-1) + |L^{++}(i)| \\ + & \text{otherwise} \end{cases}$$

Algorithm 5 DelCaterpillar: A Dynamic Programming Algorithm to find a feasible removal list for a hypergraph whose intersection graph is a caterpillar tree

function DELCATERPILLAR()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ is a caterpillar tree with k nodes in the central path. The clusters corresponding to the nodes of the central path are denoted according to their order in the path, by $S_1, S_2, \ldots, S_k$.
Output:
A feasible removal list L.

begin
  Initialize an empty list L.
  Let $L^{+-}(1) = \mathit{DelStar}(H^{+-}(1) \setminus L^{S_1,S_2}) \cup L^{S_1,S_2}$.
  Let $L^{++}(1) = \mathit{DelStar}(H^{++}(1))$.
  $G(1) = |L^{+-}(1)|$.
  $F(1) = |L^{++}(1)|$.
  Set $X_g(1) = " + ", X_f(1) = " + "$.
  for $i = 2 \ldots k-1$:
    if $L^{S_{i-1},S_i} = \{(S_i \setminus S_{i-1}, S_i)\}$, **then** set $L^{S_i,S_{i+1}} = \emptyset$.
    Let $L^{--}(i) = \mathit{DelStar}(H^{--}(i) \setminus (L^{S_{i-1},S_i} \cup L^{S_i,S_{i+1}})) \cup L^{S_i,S_{i+1}}$.
    Let $L^{+-}(i) = \mathit{DelStar}(H^{+-}(i) \setminus L^{S_i,S_{i+1}}) \cup L^{S_i,S_{i+1}}$.
    Let $L^{-+}(i) = \mathit{DelStar}(H^{-+}(i) \setminus L^{S_{i-1},S_i})$.
    Let $L^{++}(i) = \mathit{DelStar}(H^{++}(i))$.
    $G(i) = \min\{G(i-1) + |L^{--}(i)|, F(i-1) + |L^{+-}(i)|\}$.
    $F(i) = \min\{G(i-1) + |L^{-+}(i)|, F(i-1) + |L^{++}(i)|\}$.
    Calculate the appropriate $X_g(i), X_f(i)$.
  end for
  Let $L^{-+}(k) = \mathit{DelStar}(H^{-+}(k) \setminus L^{S_{k-1},S_k})$.
  Let $L^{++}(k) = \mathit{DelStar}(H^{++}(k))$.
  $F(k) = \min\{G(k-1) + |L^{-+}(k)|, F(k-1) + |L^{++}(k)|\}$.
  Calculate the appropriate $X_f(k)$.
  Let $L = \mathit{RestoreCaterpillar}(L^{--}, L^{-+}, L^{+-}, L^{++}, X_g, X_f)$.
  return L.
end function

Figure 3.18: Algorithm DelCaterpillar

Algorithm 6 RestoreCaterpillar: An Algorithm which restores the feasible removal list from the removal lists constructed by Algorithm DelCaterpillar

function RESTORECATERPILLAR()

Input:
A set of lists $L^{--}, L^{-+}, L^{+-}, L^{++}$ used to construct a feasible removal list for H, and two lists X_g, X_f used to restore the feasible removal list.
Output:
A feasible removal list L.

```
begin
    Initialize an empty list L.
    Define lastStep = "+".
    for i = k, ..., 1 :
        if lastStep == "+" : [1]
            if X_f(i) == "−" :
                Add L^{−+}(i) to L.
                Set lastStep = "−".
            else
                Add L^{++}(i) to L.
                Set lastStep = "+".
            end if
        else
            if X_g(i) == "−" :
                Add L^{−−}(i) to L.
                Set lastStep = "−".
            else
                Add L^{+−}(i) to L.
                Set lastStep = "+".
            end if
        end if
    end for
    return L.
end function
```

Figure 3.19: Algorithm RestoreCaterpillar

Remark 3.4.29. *Note that variable lastStep is set to "+" at the beginning of the algorithm. Since $X_g(k)$ is not defined, this step assures that for $i = k$,* *the first condition (marked by* [1] *in Algorithm RestoreCaterpillar) is true, and therefore, we always look at* $X_f(k)$ *for* $i = k$.

Example 3.4.30. *Let* $H =< V, \mathcal{S} >$ *be a hypergraph, whose intersection graph* $G_{int}(\mathcal{S})$ *is a caterpillar tree. Figure 3.20 shows an example for Algorithm DelCaterpillar. The results of the algorithm are:*
$G(1) = 2 + 0 = 2$, $X_g(1) = "+", L^{+-}(1) = \{18, 19\}$
$F(1) = 1$, $X_f(1) = "+", L^{++}(1) = \{7\}$
$G(2) = 1 + min\{2 + 0, 1 + 0\} = 2$, $X_g(2) = "+", L^{--}(2) = \{20\}, L^{+-}(2) = \{20\}$
$F(2) = min\{2+0, 1+2\} = 2$, $X_f(2) = "-", L^{-+}(2) = \{\}, L^{++}(2) = \{10, 11\}$
$F(3) = min\{2 + 1, 2 + 3\} = 3$, $X_f(3) = "-", L^{-+}(3) = \{31\}, L^{++}(3) = \{31, 12, 13\}$
We now look at $G(i)$ *for* $i = 2$ *as an example, and explain in detail the way the algorithm performs the calculations. For* $G(2)$, *edge* $(s_i, s_{i+1}) = (s_2, s_3)$ *is removed. Note that* $|S_2 \cap S_3| \leq |S_2 \setminus S_3|$, *and also* $|S_2 \cap S_3| \leq |S_3 \setminus S_2|$. $S_2 \cap S_3 = \{20\}$, *and therefore, the cost of removing this edge is* 1. *After removing edge* (s_2, s_3), *there are two options for removals in this step:*

- *Removing edge* (s_1, s_2), *that is, looking at* $G(1)$, *and then checking the removals for a star whose center is* s_2. *In this case, no additional removals from the star are required, therefore, the cost is* $G(1) + 0 = 2 + 0$.

- *Keeping edge* (s_1, s_2), *that is, looking at* $F(1)$ *and then checking the removals for a star whose center is* s_2. *In this case, no additional removals from the star are required, therefore, the cost is* $F(1) + 0 = 1 + 0$.

Hence, the total cost is $G(2) = |S_2 \cap S_3| + min\{G(1) + 0, F(1) + 0\} = 1 + min\{2, 1\} = 2$, *and the corresponding removal list is* $L^{+-}(2) = \{20\}$.
Restoring the minimum cardinality feasible removal list:

- *Since* $X_f(3) = "-"$, *we add* $L^{-+}(3) = \{(S_9 \setminus S_3, S_9)\}$ *to L, and then we look at* $X_g(2)$. S_9 *is transformed to a contained cluster.*

- *Since* $X_g(2) = "+"$, *we add* $L^{+-}(2) = \{(S_2 \cap S_3, S_2)\}$ *to L, and then we look at* $X_f(1)$. *Since we consider a value of* X_g, *edge* (s_2, s_3) *is removed, and we disconnect between* s_2 *and* s_3 *in the central path.*

- *Since* $X_f(1) = "+"$, *we add* $L^{++}(1) = \{(S_1 \cap S_4, S_4)\}$ *to L. Since we consider a value of* X_f, *edge* (s_1, s_2) *is not removed.* S_4 *is transformed to a singleton node.*

After the removal,
$P = (21, 22, 23, 24, 8, 9, 7, 1, 2, 18, 19, 3, 4, 10, 11, 25, 26, 27, 28, 12, 13, 20, 5, 6, 16, 17, 14, 15, 29, 30)$
is a feasible solution of FCTSP. Note that after the vertices removal, vertex 31 *is not contained in any of the clusters and therefore is not in the solution path. The removed sections are marked by red in the figure.*

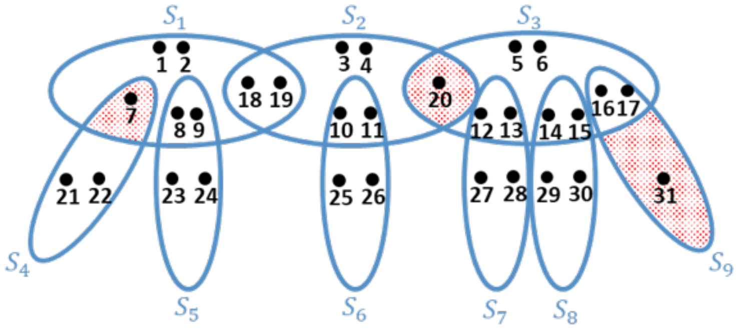

Figure 3.20: Example 3.4.30.

Example 3.4.31. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a caterpillar tree. An algorithm which considers the minimum of $\{(S_i \cap S_{i+1}, S_i)\}, \{(S_i \setminus S_{i+1}, S_i)\}, \{(S_{i+1} \setminus S_i, S_{i+1})\}$ for disconnecting s_i from s_{i+1} in $G_{int}(\mathcal{S})$ might not give a minimum cardinality result. Consider, for example, the intersection graph described in Figure 3.4.31. The results of Algorithm DelCaterpillar are:*
$G(1) = 5 + 0 = 5$*, since we remove $S_1 \cap S_2$ from S_1.*
$F(1) = 9$*, since we remove $S_3 \setminus S_1$ from S_3.*
$F(2) = min\{5 + 2, 9 + 2 + 2\} = 7$*, for $G(1)$ we remove $S_2 \cap S_5$ from S_5, for $F(1)$ we remove $S_2 \cap S_5$ from S_5 and $S_2 \cap S_6$ from S_6.*
Therefore, the algorithm removes $S_1 \cap S_2$ from S_1 and $S_2 \cap S_5$ from S_5, with a total removal of 7 *vertices. The removed sections are marked by red in the left figure.*
However, removing $S_2 \cap S_5$ from S_5, $S_2 \cap S_6$ from S_6 and $S_2 \cap S_7$ from S_7, gives a better overall solution, with a total removal of 6 *vertices. The removed sections are marked by red in the right figure.*

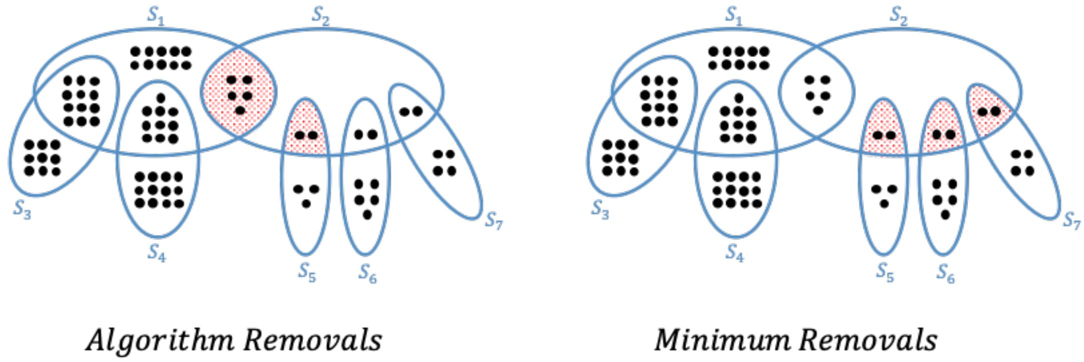

Figure 3.21: Example 3.4.31.

Theorem 3.4.32. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a caterpillar tree. Algorithm DelCaterpillar finds a feasible removal list for H.*

Proof. The clusters corresponding to the nodes of the central path are denoted according to their order in the path, by $S_1, S_2, \ldots, S_k$. Let S_i be the cluster currently processed by the algorithm. The proof is by induction on i. For $i = 1$, the structure of S_1 and its neighbours is a star graph. Since we call Algorithm DelStar on $H[S(1)]$, according to Theorem 3.4.13, the algorithm finds a feasible removal list for $H[S(1)]$.
Suppose the assumption of the lemma is correct for i, that is, the algorithm finds a feasible removal list for $H[S(1) \cup \ldots \cup S(i)]$, and now prove it for $i+1$, $1 \leq i \leq k-1$. Denote by P^+ a feasible solution for $H[S(1) \cup \ldots \cup S(i)]$, and by P^- a feasible solution for $H[S(1) \cup \ldots \cup (S(i) \setminus S_{i+1})]$. Note that after calling Algorithm DelStar on $H[S_i]$, according to Theorem 3.4.13, s_i has at most two neighbours which are not contained clusters, and s_i and those neighbours create a simple path in $G_{int}(\mathcal{S})$.
There are two options while processing S_{i+1}:

- The algorithm keeps s_{i+1} connected to s_i. When calling Algorithm DelStar on $H[S_{i+1}]$, the algorithm assures that at most one of the neighbours of s_{i+1} is kept connected to it. If s_i is also connected to s_{i-1}, P^+ contains subpaths spanning the vertices of $(S_{i-1} \setminus S_i, S_{i-1} \cap S_i, S_i \setminus (S_{i-1} \cup S_{i+1}), S_i \cap S_{i+1}, S_{i+1} \setminus S_i)$, in this order. If s_i is not connected to s_{i-1} but is connected to a leaf, we get a similar path, with some leaf s_j instead of s_{i-1}. In both cases, we can concatenate the neighbour that was kept connected to s_{i+1} adjacent to $(S_{i+1} \setminus S_i)$ in

P^+. Therefore, $H[S(1) \cup \ldots \cup S(i) \cup S(i+1)] \setminus L$ has a feasible solution of *FCTSP*.

- The algorithm disconnects s_{i+1} and s_i. When calling Algorithm DelStar on $H[S_{i+1}]$, the algorithm assures that at most two of the neighbours of s_{i+1} are kept connected to it. s_{i+1} and those neighbours create a new connected component which is a simple path, and therefore, according to Theorems 3.1.2 and 3.2.5, $H[S(1) \cup \ldots \cup S(i) \cup S(i+1)] \setminus L$ has a feasible solution of *FCTSP*.

□

Theorem 3.4.33. *Verifying whether the intersection graph of a hypergraph* $H =< V, \mathcal{S} >$ *is a caterpillar tree can be performed in* $\mathcal{O}(m^2)$ *time complexity, where* $m = |\mathcal{S}|$.

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a caterpillar tree, we remove the leaves from the intersection graph, and check that the new intersection graph is a simple path. We first parse V_D to find all the vertices whose degree is 1 in $\mathcal{O}(m)$ time complexity. We remove these leaves from $G_{int}(\mathcal{S})$ and update M_G and V_D in $\mathcal{O}(m^2)$ time complexity. According to Theorem 3.2.6, verifying that the updated intersection graph is a simple path can be performed in $\mathcal{O}(m)$ time complexity. Hence, the total time complexity of verifying that $G_{int}(\mathcal{S})$ is a caterpillar tree is $\mathcal{O}(m^2)$. □

Theorem 3.4.34. *The time complexity of Algorithm DelCaterpillar is* $\mathcal{O}(nm^3)$, $n = |V|, m = |\mathcal{S}|$.

Proof. In order to find the nodes in the central path, we parse V_D in $\mathcal{O}(m)$ time complexity, and look for nodes with degree > 1. For each node in the central path we:

1. Create the four induced hypergraphs and appropriate intersection graphs. According to Theorem 3.1.1, this can be done in $\mathcal{O}(nm^2)$ time complexity.

2. Run Algorithm DelStar on the induced hypergraphs. According to Theorem 3.4.16, this can be performed in $\mathcal{O}(mn)$ time complexity.

3. Calculate $G(i)$ and $F(i)$. This can be done in $\mathcal{O}(1)$ time complexity.

Hence, for each node, the required time complexity is $\mathcal{O}(nm^2)$. Since there are $\mathcal{O}(m)$ nodes in the central path, the total time complexity of Algorithm DelCaterpillar is $\mathcal{O}(nm^3)$. □

Theorem 3.4.35. *The time complexity of Algorithm RestoreCaterpillar is* $\mathcal{O}(nm^2)$*,* $n = |V|, m = |\mathcal{S}|$*.*

Proof. We process lists X_g, X_f, and at each step, we add the appropriate removal list. The size of X_g and X_f is $\mathcal{O}(m)$ each. The size of the removal list at each step is $\mathcal{O}(mn)$. Hence, the total time complexity of Algorithm RestoreCaterpillar is $\mathcal{O}(nm^2)$. □

3.5 Bipartite Graphs

This section introduces hypergraphs whose intersection graphs are bipartite graphs (see Definitions 3.5.1 and 3.5.2). For these hypergraphs we show that if $G_{int}(\mathcal{S})$ is isomorphic to $K_{x,y}$, $x \geq 2, y \geq 2$, then there is no feasible solution of *FCTSP*. A bipartite graph might also be a path, cycle, tree, or star graph, as shown in Figure 3.22. All the theorems and algorithms introduced in previous sections are also valid for a bipartite graph with the special structure of a path, cycle, tree, or star. We present Algorithm DelBipartite (see Figure 3.25) where the input is a hypergraph whose intersection graph is a bipartite graph, and its output is a feasible removal list. Algorithm DelBipartite does not find a minimum cardinality feasible removal list, and therefore, the order in which we process the clusters is not relevant.

Definition 3.5.1. Bipartite Graph: A graph whose nodes can be divided into two disjoint sets, such that for each two nodes u, v in the same set, there does not exist an edge (u, v).

Definition 3.5.2. Complete Bipartite Graph: A bipartite graph, denoted by $K_{x,y}$, whose nodes are divided to sets V_x, V_y, $|V_x| = x, |V_y| = y$, such that for each two nodes $u \in V_x, v \in V_y$, there exists an edge (u, v).

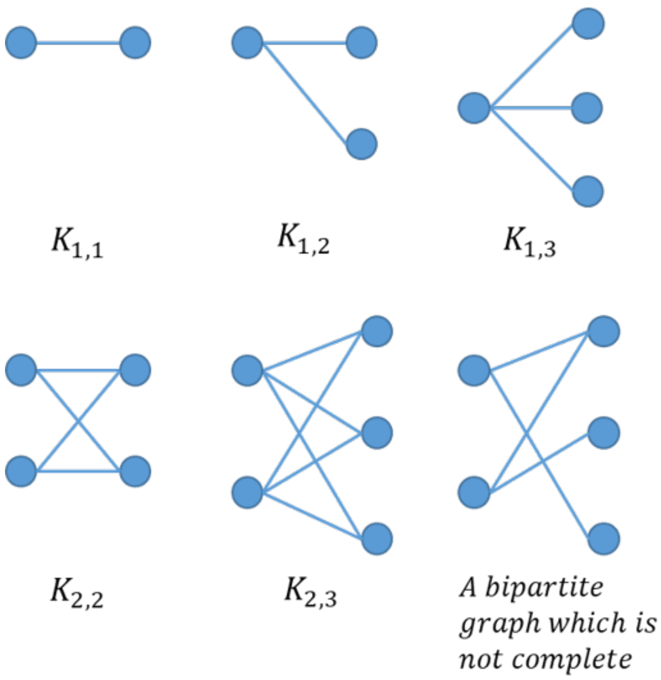

Figure 3.22: Examples of bipartite graphs.

Lemma 3.5.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. If $G_{int}(\mathcal{S})$ is isomorphic to $K_{1,1}$ or $K_{1,2}$, then H has a feasible solution of FCTSP.*

Proof. If the intersection graph is $K_{1,1}$ or $K_{1,2}$, then it is a simple path (see Figure 3.22). According to Theorem 3.2.5, H has a feasible solution of *FCTSP*. □

Lemma 3.5.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. If $G_{int}(\mathcal{S})$ is isomorphic to $K_{1,y}$, $y \geq 3$, and there are no contained clusters in $G_{int}(\mathcal{S})$, then H does not have a feasible solution of FCTSP.*

Proof. If the intersection graph is $K_{1,y}$, $y \geq 3$, then it is a star with $k \geq 3$ (see Figure 3.22). According to Theorem 3.4.8, if there are no contained clusters, H does not have a feasible solution of *FCTSP*. □

Example 3.5.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. Note that if H has contained clusters, then H might have a feasible solution of FCTSP. An example is given in Figure 3.23. In this example, H is a hypergraph whose intersection graph is a complete bipartite graph $K_{1,4}$, that has contained clusters. For this hypergraph, $P = (1, 2, 3, 4)$ is a feasible solution of FCTSP.*

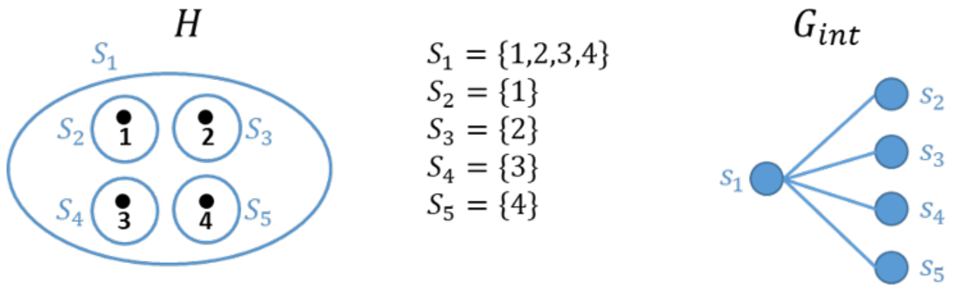

Figure 3.23: Example 3.5.5.

Lemma 3.5.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. If $G_{int}(\mathcal{S})$ is isomorphic to $K_{2,2}$, then H does not have a feasible solution of FCTSP.*

Proof. If the intersection graph is $K_{2,2}$, then it is a chordless cycle of size 4 (see Figure 3.22). According to Theorem 3.3.7, H does not have a feasible solution of *FCTSP*. □

Lemma 3.5.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S}^x)$ is a bipartite graph. If $G_{int}(\mathcal{S})$ is isomorphic to $K_{x,y}$, $x \geq 2, y \geq 2$, then there are no $S_i, S_j \in \mathcal{S}$ such that $S_j \subseteq S_i$.*

Proof. Denote the two sets of nodes of the bipartite graph by $\mathcal{S}^x, \mathcal{S}^y$, $|\mathcal{S}^x| = x, |\mathcal{S}^y| = y$. Suppose by contradiction that there exist $S_i, S_j \in \mathcal{S}$ such that $S_j \subseteq S_i$. Therefore, there exists an edge (s_i, s_j). According to the definition of a bipartite graph, s_i, s_j belong to different sets of nodes. Without loss of generality, assume that $s_i \in \mathcal{S}^x, s_j \in \mathcal{S}^y$. Since $x \geq 2$, there exists a node $s_k \in \mathcal{S}^x, s_k \neq s_i$, such that there exists an edge (s_j, s_k). In this case, $S_j \cap S_k \neq \emptyset$, and since $S_j \subseteq S_i$, also $S_i \cap S_k \neq \emptyset$. Hence, there exists an edge (s_i, s_k) between two nodes that belong to the same set of nodes, contradicting the definition of a bipartite graph. □

Theorem 3.5.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. If $G_{int}(\mathcal{S})$ is isomorphic to $K_{x,y}$, $x \geq 2, y \geq 2$, then H does not have a feasible solution of FCTSP.*

Proof. If $G_{int}(\mathcal{S})$ is isomorphic to $K_{2,2}$, then according to Lemma 3.5.6, H does not have a feasible solution of *FCTSP*.
Suppose that $G_{int}(\mathcal{S})$ is isomorphic to $K_{x,y}$, and $x > 2, y \geq 2$ or $x \geq 2, y > 2$. Denote the two sets of nodes of the bipartite graph by $\mathcal{S}^x, \mathcal{S}^y$, $|\mathcal{S}^x| = x, |\mathcal{S}^y| = y$. Without loss of generality, assume that $x > 2$. Therefore, there exists some node $s_k \in \mathcal{S}^y$, that has at least 3 neighbours $s_{i_1}, s_{i_2}, s_{i_3} \in \mathcal{S}^x$. These nodes satisfy:

1. $s_{i_1}, s_{i_2}, s_{i_3}$ are neighbours of s_k, therefore $S_k \cap S_{i_1} \neq \emptyset, S_k \cap S_{i_2} \neq \emptyset, S_{i_3} \cap S_k \neq \emptyset$.

2. According to Lemma 3.5.7, there are no $S_i, S_j \in \mathcal{S}$ such that $S_j \subseteq S_i$. Therefore, $S_{i_1} \nsubseteq S_k, S_{i_2} \nsubseteq S_k, S_{i_3} \nsubseteq S_k$.

3. $s_{i_1}, s_{i_2}, s_{i_3}$ belong to the same set of nodes, therefore $S_{i_1} \cap S_{i_2} = \emptyset, S_{i_2} \cap S_{i_3} = \emptyset, S_{i_1} \cap S_{i_3} = \emptyset$.

According to Theorem 3.1.11, H does not have a feasible solution of *FCTSP*.

□

Example 3.5.9. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. Note that if $G_{int}(\mathcal{S})$ is not a complete bipartite graph, then H might have a feasible solution of FCTSP. An example is given in Figure 3.24. For this hypergraph, $P = (1, 2, 3, 4, 5, 6)$ is a feasible solution of FCTSP.*

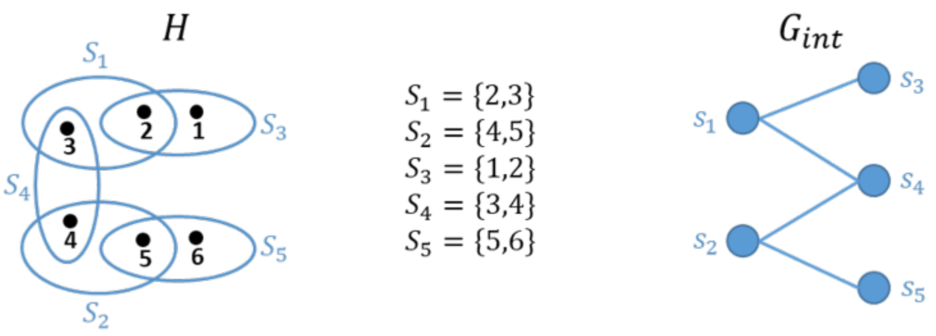

Figure 3.24: Example 3.5.9.

Algorithm 7 DelBipartite: An Algorithm to find a feasible removal list for a hypergraph whose intersection graph is a bipartite graph

function DelBipartite()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph.
Output:
A feasible removal list L for H.

begin
  Initialize an empty list L.
  while There exists a node s_i with at least 3 neighbours in $G_{int}(\mathcal{S})$:
    Define $H(i) = H[S_i \cup \{S' \mid S' \in \mathcal{S}, S' \cap S_i \neq \emptyset\}]$.
    $L_{star} = DelStar(H(i))$.
    Add L_{star} to L.
    $H = H \setminus L_{star}$.
    Remove contained clusters from H.
  end while
  while There exists a cycle C in $G_{int}(\mathcal{S})$:
    Denote the nodes of C by $s_1, \ldots, s_k$.
    $L_{cycle} = DelCycle(H[S_1, \ldots, S_k])$.
    Add L_{cycle} to L.
    $H = H \setminus L_{cycle}$.
    Remove contained clusters from H.
  end while
  return L.
end function

Figure 3.25: Algorithm DelBipartite

Example 3.5.10. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. Figure 3.26 shows an example for Algorithm DelBipartite. The algorithm first handles nodes with at least* 3 *neighbours, there are two nodes which satisfy this condition, s_1 and s_2. Suppose that s_1 is chosen by the algorithm, therefore, $S_1 \cap S_5$ is removed from S_5. After the removal, S_5 is a contained cluster, and therefore removed from the hypergraph. After this step, there are no nodes that have at least* 3 *neighbours, but there is a cycle $s_1 - s_4 - s_2 - s_3$. Therefore, $S_2 \cap S_4$ is removed from S_4. At the end of the algorithm, $P = (12, 3, 4, 5, 1, 2, 11, 6, 7, 13, 8, 9, 10)$*

is a feasible solution of FCTSP. The removed sections are marked by red in the figure.

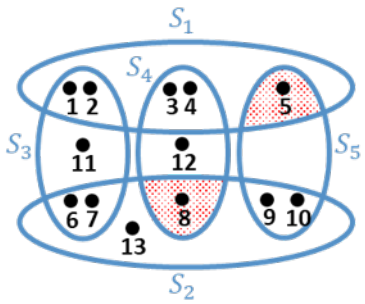

Figure 3.26: Example 3.5.10.

Theorem 3.5.11. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. Algorithm DelBipartite finds a feasible removal list for H.*

Proof. At the end of the algorithm, the degree of each node, without the contained clusters, is at most 2. Also, there are no cycles in $G_{int}(\mathcal{S} \setminus L)$. Therefore, $G_{int}(\mathcal{S} \setminus L)$ is a collection of simple paths, and according to Theorems 3.2.5 and 3.1.2, $H \setminus L$ has a feasible solution of *FCTSP*. □

Theorem 3.5.12. *Verifying whether the intersection graph of a hypergraph $H =< V, \mathcal{S} >$ is a bipartite graph can be performed in $\mathcal{O}(m^2)$ time complexity, where $m = |\mathcal{S}|$.*

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a connected bipartite graph, we use DFS algorithm, and verify that the intersection graph is 2-colorable and connected. Since there are at most m^2 edges in $G_{int}(\mathcal{S})$, performing DFS on this graph requires $\mathcal{O}(|V|+|E|) = \mathcal{O}(m+m^2) = \mathcal{O}(m^2)$ time complexity. □

Lemma 3.5.13. *Let $H =< V, \mathcal{S} >$ be a hypergraph with intersection graph $G_{int}(\mathcal{S})$. The time complexity of removing contained clusters from H is $\mathcal{O}(m^2 + nm)$.*

Proof. Cluster S_i is a contained cluster if it satisfies all the following conditions:

1. The corresponding node is a leaf in the intersection graph.

2. There exists a cluster S_j which contains all the vertices of S_i.

3. For every $v \in S_i$, $v \notin S_k$, $\forall S_k \neq S_i, S_k \neq S_j$.

We parse V_D in $\mathcal{O}(m)$ time complexity, looking for clusters which are leaves in $G_{int}(\mathcal{S})$. For each leaf, we verify that conditions two and three are satisfied by processing the row of this cluster in M_G to find the cluster it intersects with, in $\mathcal{O}(m)$ time complexity, and then processing the appropriate columns of these clusters in M_H, in $\mathcal{O}(n)$ time complexity. We update M_H, M_G and V_D, in order to remove the contained cluster, in $\mathcal{O}(n+m)$ time complexity. Since there are $\mathcal{O}(m)$ contained clusters, the total time complexity of removing all the contained clusters from H is $\mathcal{O}(m^2 + nm)$. □

Theorem 3.5.14. *The time complexity of Algorithm DelBipartite is* $\mathcal{O}(nm^3)$, $n = |V|, m = |\mathcal{S}|$.

Proof. In the first part of the algorithm, as long as there exists a node which has at least 3 neighbours in the intersection graph we:

1. Parse V_D in order to find such a node, in $\mathcal{O}(m)$ time complexity.

2. Create an induced hypergraph and intersection graph. According to Theorem 3.1.1, this can be done in $\mathcal{O}(nm^2)$ time complexity.

3. Run Algorithm DelStar on the induced hypergraph. According to Theorem 3.4.16, this can be performed in $\mathcal{O}(mn)$ time complexity.

4. Update the hypergraph and intersection graph, in $\mathcal{O}(nm^2)$ time complexity.

5. Remove contained clusters. According to Lemma 3.5.13, this can be performed in $\mathcal{O}(m^2 + nm)$ time complexity.

Since there are $\mathcal{O}(m)$ nodes which have at least 3 neigbours, the total time complexity of the first part of Algorithm DelBipartite is $\mathcal{O}(nm^3)$.
In the second part of the algorithm, as long as there exists a cycle in the intersection graph we:

1. Find a cycle using DFS algorithm, in $\mathcal{O}(m^2)$ time complexity.

2. Create an induced hypergraph and intersection graph for this cycle. According to Theorem 3.1.1, this can be done in $\mathcal{O}(nm^2)$ time complexity.

3. Run Algorithm DelCycle on the induced hypergraph. According to Theorem 3.3.13, this can be performed in $\mathcal{O}(m^2 + n)$ time complexity.

4. Update the hypergraph and intersection graph, in $\mathcal{O}(nm^2)$ time complexity.
5. Remove contained clusters. According to Lemma 3.5.13, this can be performed in $\mathcal{O}(m^2 + nm)$ time complexity.

Since there are at most $\mathcal{O}(m)$ cycles, the total time complexity of the second part of Algorithm DelBipartite is $\mathcal{O}(nm^3)$.
Hence, the total time complexity of Algorithm DelBipartite is $\mathcal{O}(nm^3)$. □

Example 3.5.15. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a bipartite graph. Figure 3.27 shows an example of a case where the algorithm does not find a minimum cardinality feasible removal list. The algorithm first handles nodes with at least 3 neighbours. Only node s_1 satisfies this condition, therefore, $S_4 \setminus S_1$ is removed from S_4. After the removal, S_4 is a contained cluster, and therefore removed from the hypergraph. After this step, there are no nodes that have at least 3 neighbours, but there is a cycle $s_1 - s_5 - s_2 - s_3$. All the intersections of clusters in the cycle are of size 2. Suppose that intersection $S_1 \cap S_5$ is chosen by the algorithm, and is removed from S_5. At the end of the algorithm, $P = (5, 6, 3, 4, 1, 2, 11, 12, 7, 8, 16, 9, 10, 14, 15)$ is a feasible solution of FCTSP. Note that after the vertices removal, vertex 13 is not contained in any of the clusters and therefore is not in the solution path. The removed sections are marked by red in the left figure.*
Considering removing only $S_1 \cap S_5$ from S_5, is sufficient in order to gain feasibility in this example.
After this removal, $P = (13, 3, 4, 5, 6, 1, 2, 11, 12, 7, 8, 16, 9, 10, 14, 15)$ is a feasible solution of FCTSP. Hence, the algorithm does not necessarily find a minimum cardinality feasible removal list. The removed sections for the minimum cardinality feasible removal list are marked by red in the right figure.

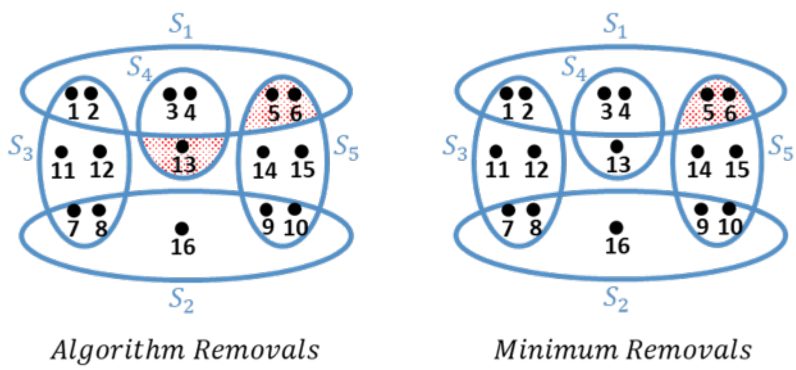

Figure 3.27: Example 3.5.15.

3.6 Clique Graphs

This section introduces hypergraphs whose intersection graphs are cliques (see Definition 3.6.1 and Figure 3.28). The first theorems in this section characterize conditions for a feasible solution for a hypergraph whose intersection graph is a clique of size $m = 3$. We present Algorithm DelClique3 (see Figure 3.31) where the input is a hypergraph whose intersection graph is a clique of size $m = 3$, and its output is a minimum cardinality feasible removal list. At the end of the section we characterize conditions for cases where there is no feasible solution for a hypergraph whose intersection graph is a clique of size $m \geq 3$. The theorems in this section uses the PUC property, defined in 2.0.12. A set of clusters satisfy the PUC property, if no cluster in the set is contained in the union of a pair of two other clusters in the set.

Definition 3.6.1. Clique: A clique is a set of nodes C, for which $\forall v_i, v_j \in C$, $i \neq j$, there exists an edge (v_i, v_j).

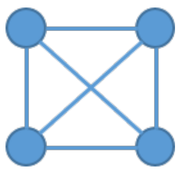

Figure 3.28: An example of a clique graph with $m = 4$.

Property 3.6.2. *A clique of size k satisfies the Helly Property if* $\bigcap_{i=1}^{k} S_i \neq \emptyset$.

Corollary 3.6.3. *Let* $H =< V, \mathcal{S} >$ *be a hypergraph, whose intersection graph* $G_{int}(\mathcal{S})$ *is a clique. If* H *satisfies the PUC property, then* H *does not have a feasible solution of FCTSP.*

Proof. According to Theorem 1.0.1, when the intersection graph is connected, and the hypergraph satisfies the PUC property, it has a feasible solution of $FCTSP$ only if the intersection graph is a path. Since a clique graph with $m \geq 3$ is not a simple path, if H satisfies the PUC property, it has no feasible solution of $FCTSP$ (see Example 1 in Figure 3.29). $\square$

Example 3.6.4. *Let* $H =< V, \mathcal{S} >$ *be a hypergraph, whose intersection graph* $G_{int}(\mathcal{S})$ *is a clique of size* $m = 3$. *If* H *does not satisfy the PUC property, there are cases where there is a feasible solution and cases where there is no feasible solution.*
Example 1 in Figure 3.29 shows a hypergraph that satisfies the PUC property, and therefore does not have a feasible solution of FCTSP.

Example 2 in Figure 3.29 shows a hypergraph that does not satisfy the PUC property, and has a feasible solution of FCTSP. For this hypergraph, $P = (1, 2, 3, 4, 5)$ is a feasible solution.
Example 3 in Figure 3.29 shows a hypergraph that does not satisfy the PUC property. We will prove in Theorem 3.6.6, that this hypergraph does not have a feasible solution of FCTSP.

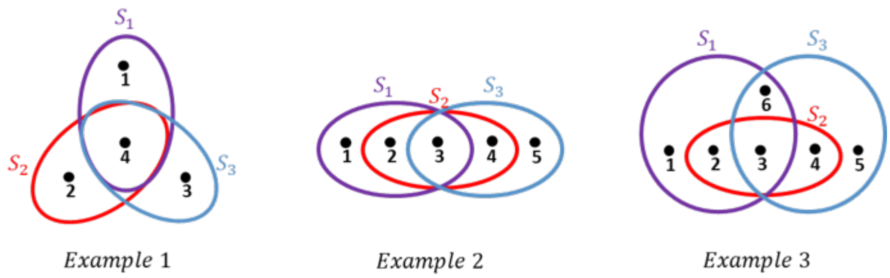

Figure 3.29: Example 3.6.4.

3.6.1 Clique Graphs of size $m = 3$

In this section, we will focus on the conditions that must be met in order to have a feasible solution, if the intersection graph is a clique of size $m = 3$. Denote the nodes of the clique by s_1, s_2, s_3, corresponding to clusters S_1, S_2, S_3. Divide the clusters into disjoint subclusters, denoted by:

- ${S_1}', {S_2}', {S_3}'$: Subclusters that contain vertices which satisfy $nc(v) = 1$. That is, ${S_1}' = S_1 \setminus (S_2 \cup S_3), {S_2}' = S_2 \setminus (S_1 \cup S_3), {S_3}' = S_3 \setminus (S_1 \cup S_2)$.
- ${S_{12}}'', {S_{23}}'', {S_{13}}''$: Subclusters that contain vertices which satisfy $nc(v) = 2$. That is, ${S_{12}}'' = (S_1 \cap S_2) \setminus S_3, {S_{23}}'' = (S_2 \cap S_3) \setminus S_1, {S_{13}}'' = (S_1 \cap S_3) \setminus S_2$.
- ${S_{123}}'''$: A subcluster that contain vertices which satisfy $nc(v) = 3$. That is, ${S_{123}}''' = S_1 \cap S_2 \cap S_3$.

Also denote, for $i, j, k \in \{1, 2, 3\}$, such that i, j, k are 3 different indices:
${S_i}^D = S_i \setminus (S_j \cup S_k)$, that is, the vertices that belong to exactly one cluster, S_i.
${S_{ij}}^\cap = S_i \cap S_j$.
${S_{ij}}^D = (S_i \cap S_j) \setminus (S_1 \cap S_2 \cap S_3)$, that is, the vertices that belong to exactly two clusters, S_i, S_j.
We use these notations in the theorems and algorithm for a hypergraph whose intersection graph is a clique of size $m = 3$.

Theorem 3.6.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m = 3$. If $S_1 \cap S_2 \cap S_3 = \emptyset$, then H does not have a feasible solution of FCTSP.*
In other words, if $\mathcal{S}$ does not satisfy the Helly Property, then H does not have a feasible solution of FCTSP.

Proof. Denote by $X_1 = S_1 \cap S_2, X_2 = S_2 \cap S_3, X_3 = S_3 \cap S_1$. Suppose by contradiction that H has a feasible solution P. According to Theorem 3.1.3, the induced solutions for X_1, X_2, X_3 are consecutive subpaths, denote these subpaths by P_1, P_2, P_3, respectively. Since the intersection graph is a clique, $X_1 \neq \emptyset, X_2 \neq \emptyset, X_3 \neq \emptyset$, and therefore P_1, P_2, P_3 are non-empty. Also, since $S_1 \cap S_2 \cap S_3 = \emptyset$, then P_1, P_2, P_3 are pairwise vertex disjoint.
Without loss of generality, assume that P_1, P_2, P_3 is the order of these subpaths, not necessarily consecutively, in P. Note that $X_1 \subseteq S_1$, $X_3 \subseteq S_1$, but $X_2 \cap S_1 = \emptyset$. Therefore, $P[S_1]$ is not consecutive in P, contradicting the fact that this solution is a feasible solution for H.
Another way to prove this theorem is by using known theorems for the Clustered Spanning Tree by Trees problem. T. A. McKee and F. R. McMorris specify in [14] that a hypergraph has a feasible solution of Clustered Spanning Tree by Trees problem if and only if it satisfies the Helly property and its intersection graph is chordal. Since *FCTSP* is a restricted case of Clustered Spanning Tree by Trees problem, and $\mathcal{S}$ does not satisfy the Helly Property, then H does not have a feasible solution of *FCTSP*. □

Theorem 3.6.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m = 3$. H has a feasible solution of FCTSP if and only if H does not satisfy the PUC property, and there exists a pair of clusters S_i, S_j, $i, j \in \{1, 2, 3\}$, $i \neq j$, that satisfies $S_i \cap S_j = S_1 \cap S_2 \cap S_3$.*

Proof. Suppose that H does not satisfy the *PUC* property, and there exists a pair of clusters S_i, S_j that satisfies $S_i \cap S_j = S_1 \cap S_2 \cap S_3$. Without loss of generality, assume that $S_3 \subseteq S_1 \cup S_2$.
Denote for $i, j \in \{1, 2, 3\}$, $i \neq j$:

- P_i': A subpath that spans consecutively the vertices of S_i'.
- P_{ij}'': A subpath that spans consecutively the vertices of S_{ij}''.
- P_{123}''': A subpath that spans consecutively the vertices of S_{123}'''.

Note that, since $S_3 \subseteq S_1 \cup S_2$, $S_3' = \emptyset$.
Consider three cases, depending on the values of the indices i and j, such that $S_i \cap S_j = S_1 \cap S_2 \cap S_3$:

- $S_1 \cap S_2 = S_1 \cap S_2 \cap S_3$: In this case, $S_{12}'' = \emptyset$. Therefore, we can construct P in the following way: $P_1', P_{13}'', P_{123}''', P_{23}'', P_2'$.
  P spans all the vertices of V, and for each cluster S_i, $P[S_i]$ is consecutive. Therefore, P is a feasible solution for H of *FCTSP*.

- $S_1 \cap S_3 = S_1 \cap S_2 \cap S_3$: In this case, $S_{13}'' = \emptyset$. Therefore, we can construct P in the following way: $P_1', P_{12}'', P_{123}''', P_{23}'', P_2'$.
  P spans all the vertices of V, and for each cluster S_i, $P[S_i]$ is consecutive. Therefore, P is a feasible solution for H of *FCTSP*.

- $S_2 \cap S_3 = S_1 \cap S_2 \cap S_3$: This case is symmetrical to the previous case, therefore, there is a feasible solution for H of *FCTSP*.

Figure 3.30 shows the general structure of the solution, for the first case described above.
We now prove the correctness of the opposite direction, by showing that if H satisfies the *PUC* property, or if there is no pair of clusters S_i, S_j, $i, j \in \{1, 2, 3\}$, $i \neq j$, that satisfies $S_i \cap S_j = S_1 \cap S_2 \cap S_3$, then H does not have a feasible solution of *FCTSP*.
If H satisfies the *PUC* property, then according to Theorem 3.6.3, H does not have a feasible solution of *FCTSP*.
If there is no pair of clusters S_i, S_j that satisfies $S_i \cap S_j = S_1 \cap S_2 \cap S_3$, then $S_{12}'' \neq \emptyset, S_{23}'' \neq \emptyset, S_{13}'' \neq \emptyset$. Consider two cases:

- $S_{123}''' = \emptyset$: In this case, according to Theorem 3.6.5, H does not have a feasible solution of *FCTSP*.

- $S_{123}''' \neq \emptyset$: Suppose by contradiction that H has a feasible solution P. According to Theorem 3.1.3, each of the subpaths $P_i', P_{ij}'', P_{123}'''$ is consecutive in P. Since P is a feasible solution, $P[S_2]$ is also a consecutive subpath. Therefore, $P_2', P_{12}'', P_{23}'', P_{123}'''$ appear consecutively in P, not necessarily in this order. Note that subclusters $S_{12}'', S_{23}'', S_{123}'''$ are non-empty, and that P_{123}''' must appear between P_{12}'' and P_{23}'' in P. Without loss of generality, assume that these subclusters are ordered $P_{12}'', P_{123}''', P_{23}''$. Consider vertex $v \in S_{13}''$. By definition, $v \notin S_2$, and therefore $v \notin P[S_2]$. If v appears on one side of $P[S_2]$ in P, then $P[S_3]$ is not consecutive, since $S_{12}'' \cap S_3 = \emptyset$. If v appears on the other side of $P[S_2]$ in P, then $P[S_1]$ is not consecutive, since $S_{23}'' \cap S_1 = \emptyset$. Contradicting the fact that P is a feasible solution of *FCTSP*.

□

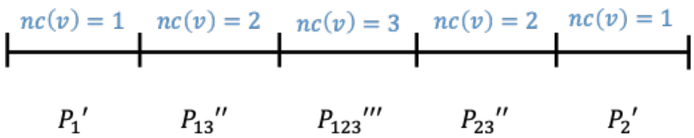

Figure 3.30: An example of a solution according to Theorem 3.6.6.

Corollary 3.6.7. *Let* $H =< V, \mathcal{S} >$ *be a hypergraph, whose intersection graph* $G_{int}(\mathcal{S})$ *is a clique of size* $m = 3$*. If there exists a pair of clusters* S_i, S_j*,* $i, j \in \{1, 2, 3\}$*,* $i \neq j$*, that satisfies* $S_j \subseteq S_i$*, then* H *has a feasible solution of FCTSP.*

Proof. Since $S_j \subseteq S_i$, H does not satisfy the *PUC* property. Without loss of generality, assume that $S_2 \subseteq S_1$. $\forall v \in S_2 \cap S_3$, $v \in S_1 \cap S_2 \cap S_3$. Also, $\forall w \notin S_2 \cap S_3$, it is also true that $w \notin S_1 \cap S_2 \cap S_3$. Hence, $S_2 \cap S_3 = S_1 \cap S_2 \cap S_3$, and according to Theorem 3.6.6, H has a feasible solution of *FCTSP*. $\square$

Algorithm 8 DelClique3: An Algorithm to find a minimum cardinality feasible removal list for a hypergraph whose intersection graph is a clique of size $m = 3$

function DelClique3()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m = 3$, denote the nodes of the clique by s_1, s_2, s_3.
Output:
A minimum cardinality feasible removal list L for H.

begin
  Initialize an empty list L.
  Calculate ${S_1}^D = S_1 \setminus (S_2 \cup S_3)$.
  Calculate ${S_2}^D = S_2 \setminus (S_1 \cup S_3)$.
  Calculate ${S_3}^D = S_3 \setminus (S_1 \cup S_2)$.
  for each Pair $S_i, S_j \in \mathcal{S}$:
    Calculate ${S_{ij}}^{\cap} = S_i \cap S_j$.
    Calculate ${S_{ij}}^D = (S_i \cap S_j) \setminus (S_1 \cap S_2 \cap S_3)$.
  end for
  Find $k^* = \underset{k \in \{1,2,3\}}{\operatorname{argmin}} \{{S_k}^D\}$.
  Find $i^*, j^* = \underset{i,j \in \{1,2,3\}}{\operatorname{argmin}} \{{S_{ij}}^D\}$.
  Find $i', j' = \underset{i,j \in \{1,2,3\}}{\operatorname{argmin}} \{{S_{ij}}^{\cap}\}$.
  if $(|{S_{k^*}}^D| + |{S_{i^*j^*}}^D|) \leq |{S_{i'j'}}^{\cap}|$:
    Add $({S_{k^*}}^D, S_{k^*})$ to L.
    Add $({S_{i^*j^*}}^D, S_{i^*})$ to L.
  else
    Add $({S_{i'j'}}^{\cap}, S_{i'})$ to L.
  end if
  return L.
end function

Figure 3.31: Algorithm DelClique3

Example 3.6.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m = 3$. Figure 3.32 shows three examples for Algorithm DelClique3.*
In Example 1 of Figure 3.32, a minimum cardinality removal list is removing ${S_3}^D$ from S_3 and removing ${S_{12}}^D$ from S_1, in order to satisfy the conditions

of Theorem 3.6.6. After the removal, $P = (1, 2, 5, 6, 9, 10, 7, 8, 3, 4, 11)$ *is a feasible solution of FCTSP. Note that after the vertices removal, vertex* 12 *is not contained in any of the clusters and therefore is not in the solution path. Therefore, the structure of the solution is:* $P_1{}', P_{13}{}'', P_{123}{}''', P_{23}{}'', P_2{}'$.
In Example 2 of Figure 3.32, a minimum cardinality removal list is removing $S_{12}{}^{\cap}$ *from* S_1*, in order to "break" the clique. After the removal,* $P = (1, 2, 3, 14, 15, 7, 8, 9, 12, 13, 10, 4, 5, 6, 11)$ *is a feasible solution of FCTSP. The structure of the solution is:* $P_1{}', P_{13}{}'', P_3{}', P_{23}{}'', P_2{}'$.
In Example 3 of Figure 3.32, a minimum cardinality removal list is removing $S_{23}{}^{\cap}$ *from* S_2*, in order to "break" the clique. Note that in this example,* $S_1 \cap S_2 \cap S_3 = \emptyset$*. After the removal,* $P = (3, 4, 1, 2, 9, 10, 7, 8, 6, 5)$ *is a feasible solution of FCTSP. The structure of the solution is:* $P_2{}', P_{12}{}'', P_1{}', P_{13}{}'', P_3{}'$.
The removed sections in each example are marked by red in the figure.

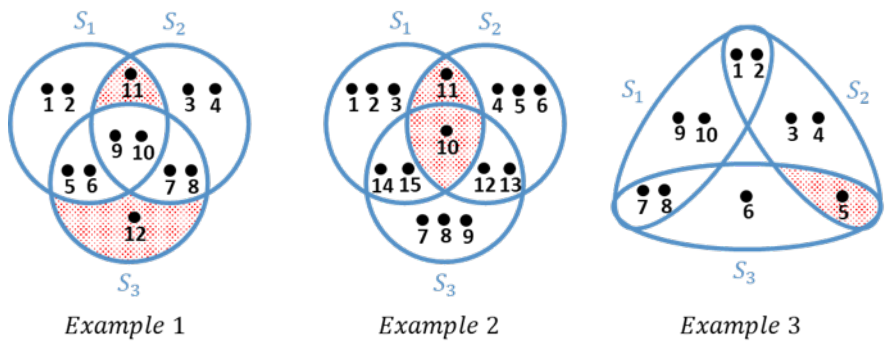

Figure 3.32: Example 3.6.8.

Theorem 3.6.9. *Let* $H =< V, \mathcal{S} >$ *be a hypergraph, whose intersection graph* $G_{int}(\mathcal{S})$ *is a clique of size* $m = 3$*. Algorithm DelClique3 finds a feasible removal list for* H*.*

Proof. Let $G_{int}(\mathcal{S} \setminus L)$ be the intersection graph for $H \setminus L$, L is the output of Algorithm DelClique3. At the end of the algorithm, $G_{int}(\mathcal{S} \setminus L)$ meets one of the following cases:

1. Suppose the algorithm chooses to remove $S_{i'} \cap S_{j'}$ from $S_{i'}$. In this case, $G_{int}(\mathcal{S} \setminus L)$ is a simple path. According to Theorem 3.2.5, $H \setminus L$ has a feasible solution of *FCTSP*.

2. Suppose the algorithm removes $S_k \setminus (S_i \cup S_j)$ from S_k and $(S_{i^*} \cap S_{j^*}) \setminus (S_1 \cap S_2 \cap S_3)$ from S_{i^*}. Denote the new clusters by $S_1{}^n, S_2{}^n, S_3{}^n$. These clusters satisfy that there exists $S_k{}^n$ such that $S_k{}^n \subseteq (S_i{}^n \cup S_j{}^n)$,

and there exists a pair of clusters ${S_{i^*}}^n, {S_{j^*}}^n$ such that $({S_{i^*}}^n \cap {S_{j^*}}^n) = ({S_1}^n \cap {S_2}^n \cap {S_3}^n)$. According to Theorem 3.6.6, $H \setminus L$ has a feasible solution of *FCTSP*.

$\square$

Remark 3.6.10. *Note that if $S_1 \cap S_2 \cap S_3 = \emptyset$, then we have to "break" the clique in order to gain feasibility. In this case, using the notation in Algorithm DelClique3 (see Figure 3.31), each pair S_i, S_j satisfies $S_i \cap S_j = (S_i \cap S_j) \setminus (S_1 \cap S_2 \cap S_3)$, that is, ${S_{i'j'}}^{\cap} = {S_{i^*j^*}}^D$. Therefore, the algorithm always removes an intersection $S_i \cap S_j$, since $|{S_{i'j'}}^{\cap}| \leq (|{S_{k^*}}^D| + |{S_{i^*j^*}}^D|)$.*

Theorem 3.6.11. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m = 3$. Algorithm DelClique3 finds a minimum cardinality feasible removal list for H.*

Proof. Since $G_{int}(\mathcal{S})$ is a clique of size $m = 3$, according to Theorem 3.6.6, H has a feasible solution of *FCTSP* if and only if H does not satisfy the *PUC* property, and there exists a pair of clusters S_i, S_j, $i, j \in \{1, 2, 3\}$, $i \neq j$, that satisfies $S_i \cap S_j = S_1 \cap S_2 \cap S_3$. If H does not satisfy these conditions, then there are two options to gain feasibility, and find a removal list accordingly:

1. Transform $G_{int}(\mathcal{S})$ to a clique which satisfies the conditions of Theorem 3.6.6. In order to satisfy the conditions of Theorem 3.6.6, choose $k \in \{1, 2, 3\}$ and remove $S_k \setminus (S_i \cup S_j)$ from S_k, and choose $i^*, j^* \in \{1, 2, 3\}$ and remove $(S_{i^*} \cap S_{j^*}) \setminus (S_1 \cap S_2 \cap S_3)$ from S_{i^*}.
2. "Break" the clique, that is, transform $G_{int}(\mathcal{S})$ to a graph which is not a clique. In order to "break" the clique, choose $i', j' \in \{1, 2, 3\}$ and remove $S_{i'} \cap S_{j'}$ from $S_{i'}$.

Since the algorithm considers all options and chooses the minimum one, the algorithm finds a minimum cardinality feasible removal list for H, where the intersection graph of it is a clique of size $m = 3$. $\square$

Theorem 3.6.12. *Verifying whether the intersection graph of a hypergraph $H =< V, \mathcal{S} >$ is a clique of size $m = 3$ can be performed in $\mathcal{O}(1)$ time complexity.*

Proof. In order to verify that $G_{int}(\mathcal{S})$ is a clique of size $m = 3$, we verify that there are exactly three nodes in the intersection graph, and the degree of each node is 2, in $\mathcal{O}(1)$ time complexity. $\square$

Theorem 3.6.13. *The time complexity of Algorithm DelClique3 is $\mathcal{O}(n)$, $n = |V|$.*

Proof. In the first part of the algorithm, we calculate the sizes of subclusters in the hypergraph. This can be calculated by comparing the columns of the clusters in M_H, and checking for each vertex v the number of clusters which contain v. Since there is a constant number of subclusters of each type, the time complexity of calculating the subclusters and their size is $\mathcal{O}(n)$.
Next we look for the minimum combination of subclusters to remove. Since there is a constant number of subclusters, this requires $\mathcal{O}(1)$ time complexity. In order to find the vertices in the appropriate subclusters, we process the columns of the clusters in M_H, in $\mathcal{O}(n)$ time complexity. Hence, the total time complexity of Algorithm DelClique3 is $\mathcal{O}(n)$. $\square$

3.6.2 Clique Graphs of size $m \geq 3$

Now we consider the general case of a hypergraph whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m \geq 3$.

Corollary 3.6.14. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m \geq 3$. If there exists a set of clusters S_i, S_j, S_k, $i, j, k \in \{1, \ldots, m\}$, i, j, k are 3 different indices, that satisfies $S_i \cap S_j \cap S_k = \emptyset$, then H does not have a feasible solution of FCTSP.*

Proof. Denote $\mathcal{S}' = \{S_i, S_j, S_k\}$. According to Theorem 3.6.5, $H[\mathcal{S}']$ does not have a feasible solution of *FCTSP*. Therefore, according to Corollary 3.1.6, H does not have a feasible solution of *FCTSP*. $\square$

Theorem 3.6.15. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m \geq 3$. If the intersection of at least three clusters in $\mathcal{S}$ is empty, then H does not have a feasible solution of FCTSP. In other words, if there exists a set of x clusters in the clique, $3 \leq x \leq m$, that does not satisfy the Helly Property, then H does not have a feasible solution of FCTSP.*

Proof. T. A. McKee and F. R. McMorris specify in [14] that a hypergraph has a feasible solution of Clustered Spanning Tree by Trees problem if and only if it satisfies the Helly property and its intersection graph is chordal. Therefore, if there exists a set of x clusters in the clique, $3 \leq x \leq m$, that does not satisfy the Helly Property, then H does not have a feasible solution of Clustered Spanning Tree by Trees problem. Since *FCTSP* is a restricted case of Clustered Spanning Tree by Trees problem, then H does not have a feasible solution of *FCTSP*. $\square$

3.7 Cut Edges

This section considers hypergraphs whose intersection graphs have a cut edge, denoted by (s_1, s_2) (see Definition 3.7.1 and Figure 3.33).
The algorithm we introduce in this section reduces the complexity of finding a feasible solution for H. This is done by removing the cut edge from the intersection graph, and thus solving each connected component separately with some restrictions.
For these hypergraphs we show that there is a feasible solution for the hypergraph if and only if there is a feasible solution for the connected components created after removing the cut edge, with S_1 and S_2 at an end of the corresponding solutions for these components. We present Algorithm DelCutEdge (see Figure 3.34) where the input is a hypergraph whose intersection graph has a cut edge, and its output is a minimum cardinality feasible removal list.

Definition 3.7.1. Cut Edge: An edge that when removed from the graph turns it to a disconnected graph. That is, removing a cut edge from a connected graph, splits it into two connected components.

Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2). We compute the minimum cardinality removal of vertices between three possible options:

1. Removing $(S_1 \cap S_2)$ from S_1 or S_2.
2. Removing $(S_1 \setminus S_2)$ from S_1.
3. Removing $(S_2 \setminus S_1)$ from S_2.

Denote by $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathcal{S}$ clusters sets which correspond to the disjoint connected components, created after removing the cut edge from $G_{int}(\mathcal{S})$, such that $s_1 \in \mathcal{S}_1, s_2 \in \mathcal{S}_2$, and removing the chosen vertices.
For component $H[\mathcal{S}_i]$, $i \in \{1, 2\}$, denote:
$mR(\mathcal{S}_i)$ = The minimum number of vertices removal for component $H[\mathcal{S}_i]$, $i \in \{1, 2\}$.
$mRE(\mathcal{S}_i)$ = The minimum number of vertices removal for component $H[\mathcal{S}_i]$, $i \in \{1, 2\}$, where the subpath spanning S_i is at an end of the feasible solution of the corresponding component.
$L^{mR}(\mathcal{S}_i)$ = A removal list for component $H[\mathcal{S}_i]$, $i \in \{1, 2\}$.
$L^{mRE}(\mathcal{S}_i)$ = A removal list for component $H[\mathcal{S}_i]$, where the subpath spanning S_i is at an end of the feasible solution of the component, $i \in \{1, 2\}$.
Also denote:
L^{REd} = A removal list for components $H[\mathcal{S}_1], H[\mathcal{S}_2]$, after the removal of

the cut edge, including the removal list for removing the cut edge from the intersection graph.
L^{KEd} = A removal list for components $H[\mathcal{S}_1], H[\mathcal{S}_2]$, when the cut edge is kept in the intersection graph, and the subpath spanning S_i is at an end of the feasible solution of the component, $i \in \{1, 2\}$.
We use these notations in the theorems and algorithm for a hypergraph whose intersection graph contains a cut edge.

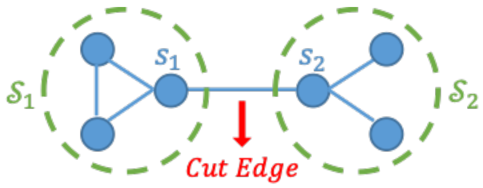

Figure 3.33: An example of a graph with a cut edge.

Theorem 3.7.2. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2). If $S_1 \subseteq S_2$ then s_1 is a leaf in $G_{int}(\mathcal{S})$, $\mathcal{S}_1 = \{S_1\}$, and S_1 is a contained cluster. Otherwise, if $S_1 \not\subseteq S_2$, then there is no $S' \in \mathcal{S} \setminus \{S_1\}$ such that $S_1 \subseteq S'$.*

Proof. Suppose that $S_1 \subseteq S_2$, and suppose by contradiction that s_1 is not a leaf in $G_{int}(\mathcal{S})$. Since s_1 is not a leaf, s_1 is connected to at least two nodes in $G_{int}(\mathcal{S})$. Therefore, s_1 is connected to some node s_i in $G_{int}(\mathcal{S})$, such that $s_i \neq s_2$. Since (s_1, s_2) is a cut edge, and there exists an edge (s_1, s_i), then $S_i \in \mathcal{S}_1$. $S_1 \subseteq S_2$ and $S_1 \cap S_i \neq \emptyset$, therefore, $S_2 \cap S_i \neq \emptyset$. That is, there exists an edge (s_2, s_i) which connects a node from $\mathcal{S}_1$ to s_2, contradicting the definition of $\mathcal{S}_1$. Hence, s_1 is a leaf in $G_{int}(\mathcal{S})$, and $\mathcal{S}_1 = \{S_1\}$.
Suppose that $S_1 \not\subseteq S_2$, and suppose by contradiction that there exists $S' \in \mathcal{S} \setminus \{S_1, S_2\}$ such that $S_1 \subseteq S'$. Since there exists an edge (s_1, s'), according to the definition of $\mathcal{S}_1$, $S' \in \mathcal{S}_1$. Since $S_1 \cap S_2 \neq \emptyset$, then $S_2 \cap S' \neq \emptyset$. That is, there exists an edge (s_2, s') which connects a node from $\mathcal{S}_1$ to s_2, contradicting the definition of $\mathcal{S}_1$. Hence, there is no $S' \in \mathcal{S} \setminus \{S_1\}$ such that $S_1 \subseteq S'$. □

Following Theorems 3.1.9 and 3.7.2, in the theorems and algorithms for cut edge and cut node, we assume that the clusters that are part of the cut edge or that are connected to the cut node, are not contained clusters.

Theorem 3.7.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2), and suppose that S_1, S_2 are not contained clusters. H has a feasible solution of FCTSP if and only if $H[\mathcal{S}_i]$, $i \in \{1, 2\}$, has a feasible solution P^i with $P^i[S_i]$ at an end of this solution.*

Proof. Suppose that H has a feasible solution P of *FCTSP*. Before removing the cut edge, $S_1 \cap S_2 \neq \emptyset$. $P[S_1]$ is the concatenation of $P[S_1 \setminus S_2], P[S_1 \cap S_2]$. $P[S_2]$ is the concatenation of $P[S_2 \setminus S_1], P[S_1 \cap S_2]$. Since P is a feasible solution, $P[S_1], P[S_2]$ are consecutive subpaths, and therefore P contains a subpath $(P[S_1 \setminus S_2], P[S_1 \cap S_2], P[S_2 \setminus S_1])$. Note that after removing the cut edge, $\bigcup_{S_i \in \mathcal{S}_1} S_i$ and $\bigcup_{S_j \in \mathcal{S}_2} S_j$ are vertex disjoint.
Since S_1, S_2 are not contained clusters, according to Theorem 3.7.2, $S_1 \nsubseteq S_2$, $S_2 \nsubseteq S_1$. Therefore, $S_1 \setminus S_2 \neq \emptyset$ and $P[S_1 \setminus S_2]$ is non-empty. Similarly, $S_2 \setminus S_1 \neq \emptyset$ and $P[S_2 \setminus S_1]$ is non-empty. Note that $(S_1 \setminus S_2) \subseteq \mathcal{S}_1$, $(S_2 \setminus S_1) \subseteq \mathcal{S}_2$, and $(S_1 \cap S_2) \nsubseteq (\mathcal{S}_1 \cup \mathcal{S}_2)$. Since $H[\mathcal{S}_1], H[\mathcal{S}_2]$ are connected components, according to Lemma 3.1.4, $P[\mathcal{S}_1], P[\mathcal{S}_2]$ are consecutive subpaths. Therefore, $P = (P[\bigcup_{S_i \in (\mathcal{S}_1 \setminus S_1)} S_i], P[S_1 \setminus S_2], P[S_1 \cap S_2], P[S_2 \setminus S_1], P[\bigcup_{S_j \in (\mathcal{S}_2 \setminus S_2)} S_j])$.
Consider $\mathcal{S}_1$. Denote by P^1 a solution path for $H[\mathcal{S}_1]$. We get that $P^1 = (P[\bigcup_{S_i \in (\mathcal{S}_1 \setminus S_1)} S_i], P[S_1 \setminus S_2])$, and spans all the vertices in $H[\mathcal{S}_1]$. For $S_k \in (\mathcal{S}_1 \setminus \{S_1\})$, $P^1[S_k] = P[S_k]$, and is therefore a consecutive subpath. Also $P[S_1 \setminus S_2]$ is at an end of P^1. Hence, P^1 is a feasible solution for $H[\mathcal{S}_1]$ with $P[S_1]$ at its end.
Similarly, the proof that $H[\mathcal{S}_2]$ has a feasible solution P^2 with $P^2[S_2]$ at an end of this solution is the same.
For the other direction, suppose that $H[\mathcal{S}_i]$, $i \in \{1, 2\}$, has a feasible solution P^i with $P^i[S_i]$ at an end of this solution. Therefore, we can construct a feasible solution for H, denoted by P, in the following way: $(P^1[\bigcup_{S_i \in (\mathcal{S}_1 \setminus S_1)} S_i]$, $P^1[S_1 \setminus S_2], P''[S_1 \cap S_2]$, $P^2[S_2 \setminus S_1], P^2[\bigcup_{S_j \in (\mathcal{S}_2 \setminus S_2)} S_j])$, where P'' is a consecutive subpath spanning the vertices in $S_1 \cap S_2$. Obviously, P spans all the vertices in H, and each cluster has a consecutive subpath in P. Hence, H has a feasible solution of *FCTSP*. $\square$

Theorem 3.7.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2), and suppose that S_1, S_2 are not contained clusters. The size of a minimum cardinality feasible removal list for H is:*
$\min\{\min\{|S_1 \cap S_2|, |S_1 \setminus S_2|, |S_2 \setminus S_1|\} + mR(\mathcal{S}_1) + mR(\mathcal{S}_2)$,
$mRE(\mathcal{S}_1) + mRE(\mathcal{S}_2)\}$.

Proof. According to Theorem 3.7.3, if H has a feasible solution and S_1, S_2 are not contained clusters, then $H[\mathcal{S}_1]$ has a feasible solution with $P[S_1]$ at an end of $P[\mathcal{S}_1]$, and $H[\mathcal{S}_2]$ has a feasible solution with $P[S_2]$ at an end of $P[\mathcal{S}_2]$. Therefore, if H does not have a feasible solution, then at least one of the following conditions is satisfied:

1. There exists a connected component $H[\mathcal{S}_i]$, $i \in \{1, 2\}$, such that $H[\mathcal{S}_i]$ does not have a feasible solution.

2. There exists a connected component $H[\mathcal{S}_i]$, $i \in \{1,2\}$, such that $H[\mathcal{S}_i]$ has a feasible solution, but there is no feasible solution P with $P[S_i]$ at an end of it.

There are two options for removals, in order to gain feasibility:

- Remove the cut edge, and solve each connected component independently. That is, remove edge (s_1, s_2), and find a removal list for each connected component, $H[\mathcal{S}_1], H[\mathcal{S}_2]$, independently. The removal of the edge is done by removing $S_1 \cap S_2$ from S_1, or by removing $S_1 \setminus S_2$ from S_1 or $S_2 \setminus S_1$ from S_2. Removing $S_1 \cap S_2$ from S_1 obviously removes edge (s_1, s_2) from $G_{int}(\mathcal{S})$. Let $H_1 = H[\mathcal{S}_1] \setminus L_1$ with $L_1 = (S_1 \setminus S_2, S_1)$ and ${S_1}' = S_1 \setminus S_2$. In H_1, ${S_1}'$ is a contained cluster. According to Theorem 3.1.9, H_1 has a feasible solution if and only if $H_1[\mathcal{S}_1 \setminus {S_1}']$ has a feasible solution. Therefore, we can solve the problem without S_1, and edge (s_1, s_2) is removed from $G_{int}(\mathcal{S})$. The number of vertices removal for this option is $\min\{|S_1 \cap S_1|, |S_1 \setminus S_2|, |S_2 \setminus S_1|\} + mR(\mathcal{S}_1) + mR(\mathcal{S}_2)$.
- Keep the cut edge, and solve each connected component $H[\mathcal{S}_i]$, $i \in \{1,2\}$, with a condition that $P[S_i]$ is at an end of the solution. That is, find removal lists L_i, $i \in \{1,2\}$, such that $H[\mathcal{S}_i] \setminus L_i$ has a feasible solution with $P[S_i]$ at an end of it. According to Theorem 3.7.3, $H \setminus (L_1 \cup L_2)$ has a feasible solution of *FCTSP*. After removing the cut edge, $\bigcup_{S_i \in \mathcal{S}_1} S_i$ and $\bigcup_{S_j \in \mathcal{S}_2} S_j$ are vertex disjoint, therefore, L_1, L_2 are vertex disjoint. The number of vertices removal for this option is $mRE(\mathcal{S}_1) + mRE(\mathcal{S}_2)$.

Hence, a minimum cardinality feasible removal list is:
$\min\{\min\{|S_1 \cap S_2|, |S_1 \setminus S_2|, |S_2 \setminus S_1|\} + mR(\mathcal{S}_1) + mR(\mathcal{S}_2)$,
$mRE(\mathcal{S}_1) + mRE(\mathcal{S}_2)\}$. $\square$

Algorithm 9 DelCutEdge: An Algorithm to find a minimum cardinality feasible removal list for a hypergraph whose intersection graph has a cut edge

function DELCUTEDGE()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2).
Output:
A minimum cardinality feasible removal list L for H.

begin
  Calculate $L^{KEd} = L^{mRE}(\mathcal{S}_1) + L^{mRE}(\mathcal{S}_2)$.
  Calculate $L^{REd} = L^{mR}(\mathcal{S}_1) + L^{mR}(\mathcal{S}_2) +$
    $\min\{|S_1 \cap S_2|, |S_1 \setminus S_2|, |S_2 \setminus S_1|\}$.
  if $|L^{KEd}| \leq |L^{REd}|$:
    Set $L = L^{KEd}$.
  else
    Set $L = L^{REd}$.
  end if
  return L.
end function

Figure 3.34: Algorithm DelCutEdge

Example 3.7.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2). The calculation of $H[\mathcal{S}_i]$ with $P[S_i]$ at an end, can be done using theorems for known ends of path in Section 3.9. Furthermore, if $G_{int}(\mathcal{S}_i)$ has a structure with a known solution, we can use algorithms described in this work in order to find a feasible removal list for $H[\mathcal{S}_i]$.*
Figure 3.35 shows an example for the use of algorithm DelCutEdge. In this example, $G_{int}(\mathcal{S}_1)$ is a star and $G_{int}(\mathcal{S}_2)$ is a clique of size 3. Therefore, we can use Algorithm DelStar (Figure 3.11), Algorithm DelClique3 (Figure 3.31), and results in Section 3.9 in order to find a feasible removal list for H.

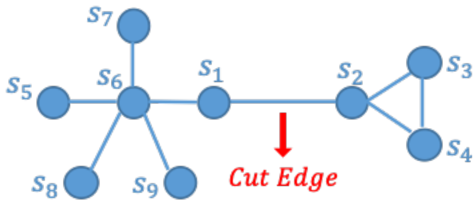

Figure 3.35: Example 3.7.5.

Example 3.7.6. *Let* $H =< V, \mathcal{S} >$ *be a hypergraph, whose intersection graph* $G_{int}(\mathcal{S})$ *has a cut edge, denoted by* (s_1, s_2)*. Figure 3.36 shows An example for Algorithm DelCutEdge.* $G_{int}(\mathcal{S}_1)$ *is a simple path, and therefore, according to Theorems 3.2.5 and 3.9.6, has a feasible solution with* $P[S_1]$ *at an end of it.* $G_{int}(\mathcal{S}_2)$ *is a clique of size 3 which does not satisfy the PUC property, and therefore, according to Theorem 3.6.6, does not have a feasible solution.*
The calculations made by the algorithm:
The minimum removals for component $H[\mathcal{S}_1]$ *is* $mR(\mathcal{S}_1) = 0$.
The minimum removals for component $H[\mathcal{S}_2]$ *is* $mR(\mathcal{S}_2) = 1$, $L^{mR}(\mathcal{S}_2) = \{(\{9\}, S_2)\}$.
The minimum removals for component $H[\mathcal{S}_1]$*, where* $P[S_1]$ *is at an end of the solution, is* $mRE(\mathcal{S}_1) = 0$.
The minimum removals for component $H[\mathcal{S}_2]$*, where* $P[S_2]$ *is at an end of the solution, is* $mRE(\mathcal{S}_2) = 1$, $L^{mRE}(\mathcal{S}_2) = \{(\{9\}, S_2)\}$.
The cost of removing edge (s_1, s_2) *is* $\min\{|S_1 \cap S_1|, |S_1 \setminus S_2|, |S_2 \setminus S_1|\} = \min\{2, 2, 3\} = 2$.
Hence, compute:
$\min\{\min\{|S_1 \cap S_1|, |S_1 \setminus S_2|, |S_2 \setminus S_1|\} + mR(\mathcal{S}_1) + mR(\mathcal{S}_2),$
$mRE(\mathcal{S}_1) + mRE(\mathcal{S}_2)\} =$
$\min\{2 + 0 + 1, 0 + 1\} = 1.$
Therefore, a minimum cardinality feasible removal list is $L = \{(\{9\}, S_2)\}$*. After the removal,* $P = (1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 9, 14, 12, 13)$ *is a feasible solution of FCTSP. The removed section is marked by red in the figure.*

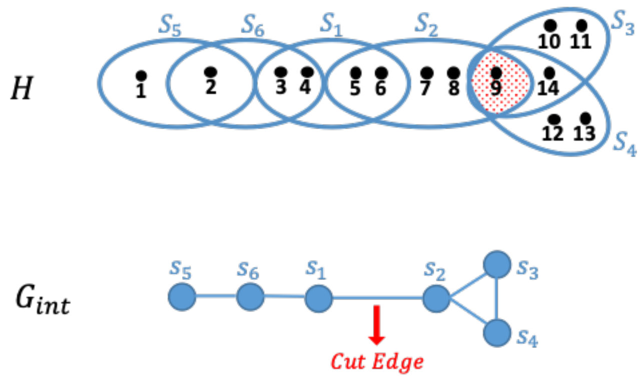

Figure 3.36: Example 3.7.6.

Theorem 3.7.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2). Algorithm DelCutEdge finds a feasible removal list for H.*

Proof. Algorithm DelCutEdge finds a removal list which satisfies one of the cases:

1. For each connected component, $H[\mathcal{S}_i] \setminus L^{mR}(\mathcal{S}_i)$, $i \in \{1, 2\}$, has a feasible solution, and the cut edge is removed. Therefore, according to Theorem 3.1.2, $H \setminus L^{REd}$ has a feasible solution of *FCTSP*.

2. For each connected component, $H[\mathcal{S}_i] \setminus L^{mRE}(\mathcal{S}_i)$, $i \in \{1, 2\}$, has a feasible solution with $P[S_i]$ at an end of it. Therefore, according to Theorem 3.7.3, $H \setminus L^{KEd}$ has a feasible solution of *FCTSP*.

□

Corollary 3.7.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut edge, denoted by (s_1, s_2). Algorithm DelCutEdge finds a minimum cardinality feasible removal list for H.*

Proof. By Theorem 3.7.7, Algorithm DelCutEdge finds a feasible removal list, and by Theorem 3.7.4, Algorithm DelCutEdge finds a minimum cardinality feasible removal list for H. □

3.8 Cut Nodes

This section considers hypergraphs whose intersection graphs have a cut node, denoted by s^* (see Definition 3.8.1 and Figure 3.37). For these hypergraphs we characterize conditions for a feasible solution, according to the

number of connected components which are connected to s^* in the intersection graph. We present Algorithm DelCutNode (see Figure 3.39) where the input is a hypergraph whose intersection graph has a cut node, and its output is a minimum cardinality feasible removal list.
Similarly to a cut edge, the algorithm we introduce in this section for a cut node makes it possible to reduce the complexity of the problem. This is done by removing the cut node and its edges from the intersection graph and running the algorithm separately on each connected component. Note that a cut node may be connected with several edges to each connected component, and therefore, removing the cut node and its edges may cause many removals of vertices from the hypergraph.

Definition 3.8.1. Cut Node: A node that when removed from the graph, with the edges containing this node, turns it to a disconnected graph. That is, removing a cut node from a connected graph, splits it into two or more connected components.

Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. We compute the minimum cardinality removal of vertices between three possible options, considering clusters $S_i \in \mathcal{S}_i$ which satisfy $S_i \cap S^* \neq \emptyset$:

1. Removing $(S_i \cap S^*)$ from S_i or S^*.
2. Removing $(S_i \setminus S^*)$ from S_i.
3. Removing $(S^* \setminus S_i)$ from S^*.

The minimum cardinality removal list for disconnecting the components might be a combination of those options. However, Similarly to the case of a star graph, we will not consider removals from S^*.
Let P^* be a subpath spanning the vertices in S^*.
Denote by $\mathcal{S}_1, \ldots, \mathcal{S}_k \subseteq \mathcal{S}$, $k \geq 2$, clusters sets which correspond to the disjoint connected components, created after removing the cut node and its edges from $G_{int}(\mathcal{S})$, and removing the chosen vertices for disconnecting the components.
For component $H[\mathcal{S}_i]$, $1 \leq i \leq k$, denote:
$mR(\mathcal{S}_i \cup \{S^*\})$ = The minimum number of vertices removal for component $H[\mathcal{S}_i \cup \{S^*\}]$, $1 \leq i \leq k$.
$mR(\mathcal{S}_i)$ = The minimum number of vertices removal for component $H[\mathcal{S}_i]$, $1 \leq i \leq k$, where S^* is not part of the component, including the minimum number of vertices removal for disconnecting S^* from the component.
$mRE(\mathcal{S}_i \cup \{S^*\})$ = The minimum number of vertices removal for component

$H[\mathcal{S}_i \cup \{S^*\}]$, $1 \leq i \leq k$, where P^* is at an end of the feasible solution of the corresponding component.
$L^{mR}(\mathcal{S}_i \cup \{S^*\})$ = A removal list for component $H[\mathcal{S}_i \cup \{S^*\}]$, $1 \leq i \leq k$.
$L^{mR}(\mathcal{S}_i)$ = A removal list for component $H[\mathcal{S}_i]$, where S^* is not part of the component, including the removal list for disconnecting S^* from the component, $1 \leq i \leq k$.
$L^{mRE}(\mathcal{S}_i \cup \{S^*\})$ = A removal list for component $H[\mathcal{S}_i \cup \{S^*\}]$, where P^* is at an end of the feasible solution of the component, $1 \leq i \leq k$.
Also denote:
$cost^{2con} = \min\limits_{1 \leq i,j \leq k} \{mRE(\mathcal{S}_i \cup \{S^*\}) + mRE(\mathcal{S}_j \cup \{S^*\}) + \sum_{t \neq i,j} mR(\mathcal{S}_t)\}$.
That is, finding solutions P^i, P^j for two components, with $P^i[S^*], P^j[S^*]$ at an end of them, and removing S^* from the other $(k-2)$ components.
$cost^{1con} = \min\limits_{1 \leq i \leq k} \{mR(\mathcal{S}_i \cup \{S^*\}) + \sum_{t \neq i} mR(\mathcal{S}_t)\}$. That is, finding a solution P for one component, with $P[S^*]$ not necessarily at its end, and removing S^* from the other $(k-1)$ components.
We use these notations in the theorems and algorithm for a hypergraph whose intersection graph contains a cut node.

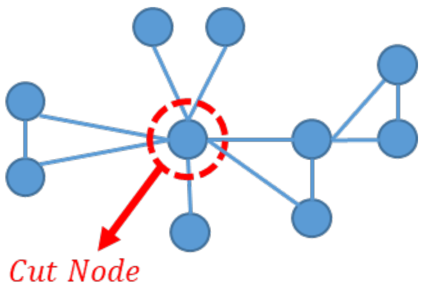

Figure 3.37: An example of a graph with a cut node.

Theorem 3.8.2. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. If there exists $S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2, S_3 \in \mathcal{S}_3$ such that $S_i \cap S^* \neq \emptyset$, $S_i \not\subseteq S^*$, $i \in \{1, 2, 3\}$, then H does not have a feasible solution of FCTSP (see Figure 3.38).*

Proof. S_1, S_2, S_3 belong to different connected components, and therefore, $S_1 \cap S_2 = \emptyset, S_1 \cap S_3 = \emptyset, S_2 \cap S_3 = \emptyset$. Also, according to the assumptions in the theorem, $S_i \cap S^* \neq \emptyset$, $S_i \not\subseteq S^*$, $i \in \{1, 2, 3\}$. Hence, according to Theorem 3.1.11, H does not have a feasible solution of *FCTSP*. □

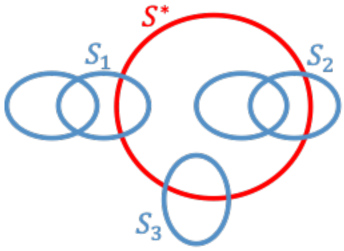

Figure 3.38: The structure of components $H[\mathcal{S}_1], H[\mathcal{S}_2], H[\mathcal{S}_3]$ in Theorem 3.8.2.

Corollary 3.8.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. If H has a feasible solution P of FCTSP, then there are at most two components, $H[\mathcal{S}_1], H[\mathcal{S}_2]$, such that there exist $S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2$ which satisfy $S_i \cap S^* \neq \emptyset$, $S_i \nsubseteq S^*$, $i \in \{1, 2\}$.*

Theorem 3.8.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. The size of a minimum cardinality feasible removal list for H is:* $\min\{cost^{2con}, cost^{1con}\}$.

Proof. According to Corollary 3.8.3, if H has a feasible solution P of $FCTSP$, then there are at most two components, $H[\mathcal{S}_1], H[\mathcal{S}_2]$, such that there exist $S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2$ which satisfy $S_i \cap S^* \neq \emptyset$, $S_i \nsubseteq S^*$, $i \in \{1, 2\}$. In other words, if H does not have a feasible solution, then there exist at least three components, $H[\mathcal{S}_1], H[\mathcal{S}_2], H[\mathcal{S}_3]$, such that $S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2, S_3 \in \mathcal{S}_3$, $S_i \cap S^* \neq \emptyset$, $S_i \nsubseteq S^*$, $i \in \{1, 2, 3\}$. Therefore, a minimum cardinality feasible removal list for H can be found by considering the following options:

- Finding solutions P^i, P^j for two components, with $P^i[S^*], P^j[S^*]$ at an end of them, and removing S^* from the other $(k-2)$ components. There are $\binom{k}{2}$ options to choose these two components.
  Calculate $\mathcal{S}_{i^*}, \mathcal{S}_{j^*} = \underset{1 \leq i,j \leq k}{\text{argmin}}\{$ The cost for finding solutions P^i, P^j for $H[\mathcal{S}_i], H[\mathcal{S}_j]$, with $P^i[S^*], P^j[S^*]$ at an end of them, and disconnecting S^* from the other $(k-2)$ components $\}$.
  The number of vertices removal for this option is:
  $cost^{2con} = \min\limits_{1 \leq i,j \leq k}\{mRE(\mathcal{S}_i \cup \{S^*\}) + mRE(\mathcal{S}_j \cup \{S^*\}) + \sum_{t \neq i,j} mR(\mathcal{S}_t)\}$.

- Finding a solution P for one component, with $P[S^*]$ not necessarily at its end, and removing S^* from the other $(k-1)$ components. There are $\binom{k}{1} = k$ options to choose this component.
  Calculate $\mathcal{S}_{i^*} = \underset{1 \leq i \leq k}{\text{argmin}}\{$ The cost for finding a solution P for $H[\mathcal{S}_i]$,

with $P[S^*]$ not necessarily at its end, and disconnecting S^* from the other $(k-1)$ components }.
The number of vertices removal for this option is:
$cost^{1con} = \min\limits_{1 \leq i \leq k} \{mR(\mathcal{S}_i \cup \{S^*\}) + \sum_{t \neq i} mR(\mathcal{S}_t)\}$.

Note that the option of removing S^* from all the components is dominated by the second option of calculating $cost^{1con}$. Let $H[\mathcal{S}_i]$ be the component that is kept connected to S^* in the second option. While calculating $mR(\mathcal{S}_i \cup \{S^*\})$, the algorithm may disconnect S^* from the rest of the component in this option.
Hence, the size of a minimum cardinality feasible removal list is:
$\min\{cost^{2con}, cost^{1con}\}$. □

Algorithm 10 DelCutNode: An Algorithm to find a minimum cardinality feasible removal list for a hypergraph whose intersection graph has a cut node

function DELCUTNODE()

Input:
A hypergraph $H =< V, \mathcal{S} >$ whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*.
Output:
A minimum cardinality feasible removal list L for H.

begin
  for Every $\{i, j\} \subseteq \{1, \ldots, k\}$, $i \neq j$:
    Calculate $L^{2con}(i, j) = L^{mRE}(\mathcal{S}_i \cup \{S^*\}) + L^{mRE}(\mathcal{S}_j \cup \{S^*\}) + \bigcup_{t \neq i,j} L^{mR}(\mathcal{S}_t)$.
  end for
  for Every $i \in \{1, \ldots, k\}$:
    Calculate $L^{1con}(i) = L^{mR}(\mathcal{S}_i \cup \{S^*\}) + \bigcup_{t \neq i} L^{mR}(\mathcal{S}_t)$.
  end for
  Calculate $(i^*, j^*) = \underset{1 \leq i,j \leq k}{\operatorname{argmin}}\{|L^{2con}(i, j)|\}$.
  Calculate $i' = \underset{1 \leq i \leq k}{\operatorname{argmin}}\{|L^{1con}(i)|\}$.
  if $|L^{2con}(i^*, j^*)| \leq |L^{1con}(i')|$:
    Set $L = L^{2con}(i^*, j^*)$.
  else
    Set $L = L^{1con}(i')$.
  end if
  return L.
end function

Figure 3.39: Algorithm DelCutNode

Remark 3.8.5. *Similarly to Algorithm DelCutEdge, the calculation of $H[\mathcal{S}_i]$ with $P[S_i]$ at an end can be done using theorems in Section 3.9. Furthermore, if $G_{int}(\mathcal{S}_i)$ has a structure with a known solution, we can use algorithms described in this work in order to find a feasible removal list for $H[\mathcal{S}_i]$.*

Example 3.8.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. Figure 3.40 shows an example for Algorithm DelCutNode. After removing the cut node and its edges, there are three connected components, denoted by $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$. Since there exist $S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2, S_3 \in \mathcal{S}_3$ such that $S_i \cap S^* \neq \emptyset$, $S_i \not\subseteq S^*$, $i \in \{1, 2, 3\}$,*

according to Theorem 3.8.2, H does not have a feasible solution. The calculations made by the algorithm:
$mR(\mathcal{S}_1 \cup \{S^*\}) = 0.$
$mR(\mathcal{S}_2 \cup \{S^*\}) = 0.$
$mR(\mathcal{S}_3 \cup \{S^*\}) = 0.$
$mR(\mathcal{S}_1) = 2,\ L^{mR}(\mathcal{S}_1) = \{(\{4,5\}, S_1)\}.$
$mR(\mathcal{S}_2) = 2,\ L^{mR}(\mathcal{S}_2) = \{(\{9,10\}, S_2)\}.$
$mR(\mathcal{S}_3) = 1,\ L^{mR}(\mathcal{S}_3) = \{(\{1\}, S_3)\}.$
$mRE(\mathcal{S}_1 \cup \{S^*\}) = 0.$
$mRE(\mathcal{S}_2 \cup \{S^*\}) = 0.$
$mRE(\mathcal{S}_3 \cup \{S^*\}) = 0.$
$cost^{2con} =$

$$\min_{1\le i,j\le k}\{mRE(\mathcal{S}_i \cup \{S^*\}) + mRE(\mathcal{S}_j \cup \{S^*\}) + \textstyle\sum_{t\neq i,j} mR(\mathcal{S}_t)\} =$$
$$mRE(\mathcal{S}_1 \cup \{S^*\}) + mRE(\mathcal{S}_2 \cup \{S^*\}) + mR(\mathcal{S}_3) = 0 + 0 + 1 = 1.$$

$cost^{1con} =$

$$\min_{1\le i\le k}\{mR(\mathcal{S}_i \cup \{S^*\}) + \textstyle\sum_{t\neq i} mR(\mathcal{S}_t)\} =$$
$$mR(\mathcal{S}_1 \cup \{S^*\}) + mR(\mathcal{S}_2) + mR(\mathcal{S}_3) = 0 + 2 + 1 = 3.$$

Therefore, a minimum cardinality feasible removal list is $L = \{(S^* \cap S_3, S_3)\} = \{(\{1\}, S_3)\}$. The solution keeps S^* at an end of two components and removes it from the other component. After the removal, $P = (8,7,6,4,5,1,14,9,10,11,12,13,2,3)$ is a feasible solution of FCTSP. The removed section is marked by red in the figure.

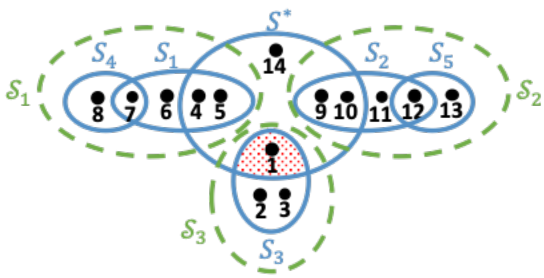

Figure 3.40: Example 3.8.6.

Theorem 3.8.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. Algorithm DelCutNode finds a feasible removal list for H.*

Proof. Algorithm DelCutNode finds a removal list which satisfies one of the cases:

1. $L = L^{2con}(i,j)$: For two connected components, $H[\mathcal{S}_i] \setminus L^{mRE}(\mathcal{S}_i \cup \{S^*\}), H[\mathcal{S}_j] \setminus L^{mRE}(\mathcal{S}_j \cup \{S^*\})$ have feasible solutions P^i, P^j, with $P^i[S^*], P^j[S^*]$ at an end of them. Since $H[\mathcal{S}_i], H[\mathcal{S}_j]$ are connected by S^*, they create a single connected component. We can construct a feasible solution for this component in the following way: $(P^i[\bigcup_{S_k \in (\mathcal{S}_i \setminus S_i)} S_k], P^i[S_i \setminus S^*], P^*, P^j[S_j \setminus S^*], P^j[\bigcup_{S_k \in (\mathcal{S}_j \setminus S_j)} S_k])$. For each of the other components $H[\mathcal{S}_t]$, $1 \leq t \leq k$, $t \neq i, j$, $H[\mathcal{S}_t] \setminus L^{mR}(\mathcal{S}_t)$ has a feasible solution where S^* is not part of the component. Therefore, according to Theorem 3.1.2, $H \setminus L$ has a feasible solution of *FCTSP*.

2. $L = L^{1con}(i')$: For one connected component, $H[\mathcal{S}_{i'}] \setminus L^{mR}(\mathcal{S}_{i'} \cup \{S^*\})$ has a feasible solution $P^{i'}$, with $P^{i'}[S^*]$ not necessarily at its end. For each of the other components $H[\mathcal{S}_t]$, $1 \leq t \leq k$, $t \neq i'$, $H[\mathcal{S}_t] \setminus L^{mR}(\mathcal{S}_t)$ has a feasible solution where S^* is not part of the component. Therefore, according to Theorem 3.1.2, $H \setminus L$ has a feasible solution of *FCTSP*.

□

Corollary 3.8.8. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ has a cut node, denoted by s^*. Algorithm DelCutNode finds a minimum cardinality feasible removal list for H.*

Proof. Follows from Theorems 3.8.7 and 3.8.4. □

3.9 Theorems for Known Ends of Path

In this section we introduce several theorems for cases where there exists a feasible solution for a hypergraph with a known cluster at an end of the solution path. We also present theorems for specific structures of the intersection graph with a known cluster at an end of the solution, for hypergraphs whose intersection graph is a path, a clique of size 3, or a star. These theorems can be used for algorithms to solve hypergraphs with intersection graphs which have a cut edge or a cut node.

Lemma 3.9.1. *Let $H =< V, \mathcal{S} >$ be a hypergraph that has a feasible solution P of FCTSP, with intersection graph $G_{int}(\mathcal{S})$. Suppose $\mathcal{S}$ contains clusters $S_i, S_j, S_k \in \mathcal{S}$, where i, j, k are 3 different indices, such that $S_i \cap S_j \neq \emptyset$, $S_i \cap S_k \neq \emptyset$, and vertices $v_j \in S_j \setminus (S_i \cup S_k)$ and $v_k \in S_k \setminus (S_i \cup S_j)$, then there is no feasible solution P with $P[S_i]$ at its end (see Figure 3.41).*

Proof. Suppose by contradiction that $P[S_i]$ is at an end of P. $S_i, \{v_j\}, \{v_k\}$ are pairwise vertex disjoint, therefore, $v_j \notin P[S_i], v_k \notin P[S_i]$. Since $S_i \cap S_j \neq \emptyset$ and $S_i \cap S_k \neq \emptyset$, $P[S_i]$ contains vertices from S_j and vertices from S_k. Since $P[S_i]$ is at an end of P, without loss of generality, suppose that v_j appears, relative to v_k, closer to $P[S_i]$ in P. Since $S_i \cap S_k \subseteq S_i$, v_j is between $P[S_i \cap S_k]$ and v_k in P (see Figure 3.42). Hence, $P[S_k]$ is not consecutive, contradicting the fact that P is a feasible solution for H of *FCTSP*. $\square$

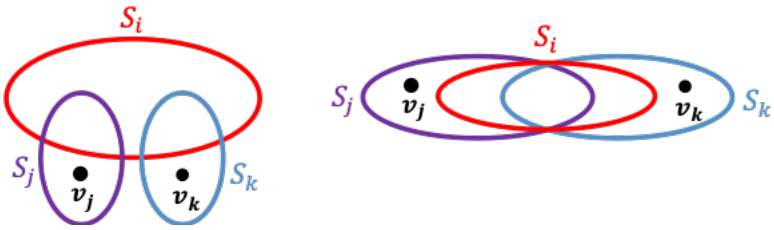

Figure 3.41: Examples of the structure of clusters S_i, S_j, S_k in Lemma 3.9.1.

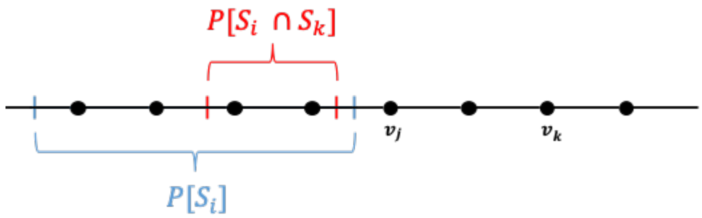

Figure 3.42: The structure of the path in the proof of Lemma 3.9.1.

Lemma 3.9.2. *Let $H =< V, \mathcal{S} >$ be a hypergraph that has a feasible solution P of FCTSP, with intersection graph $G_{int}(\mathcal{S})$. If there exists $\mathcal{S}' \subseteq \mathcal{S}$ such that there exists $v \in \bigcap_{S' \in \mathcal{S}'} S'$, and there is a feasible solution P for H with v at an end of P, then every $S_j, S_k \in \mathcal{S}'$ satisfy that either $S_j \subseteq S_k$ or $S_k \subseteq S_j$ (see Figure 3.43).*

Proof. Let $S_j, S_k \in \mathcal{S}'$, $j \neq k$, and suppose by contradiction that $S_j \nsubseteq S_k$ and $S_k \nsubseteq S_j$. Furthermore, $S_j \setminus S_k \neq \emptyset, S_k \setminus S_j \neq \emptyset, S_j \cap S_k \neq \emptyset$, and are pairwise vertex disjoint.
According to Theorem 3.1.3, $P[S_j \cap S_k]$ is a consecutive subpath. Since v is at an end of P, and $v \in (S_j \cap S_k)$, then $P[S_j \cap S_k]$ is at an end of P. Without loss of generality, suppose that $P[S_j \setminus S_k]$ appears closer to v, relative to $P[S_k \setminus S_j]$, in P. Since $S_k \setminus S_j \neq \emptyset$ and $v \in S_k$, $P[S_k]$ appears on both sides

of $P[S_j \setminus S_k]$. Contradicting the fact that P is a feasible solution for H of *FCTSP*. $\square$

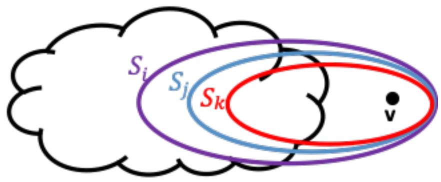

Figure 3.43: The structure of clusters in Lemma 3.9.2.

We now consider cases where a node s_i, corresponding to a cluster S_i, is a leaf in the intersection graph, and characterize when there is a feasible solution with S_i at an end of it.

Theorem 3.9.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph with intersection graph $G_{int}(\mathcal{S})$. Let $S_i, S_j \in \mathcal{S}$ such that s_i is a leaf in $G_{int}(\mathcal{S})$, and $S_i \subseteq S_j$. H has a feasible solution P of FCTSP with $P[S_i]$ at an end of P if and only if H has a feasible solution P with $P[S_j]$ at an end of P (see Figure 3.44).*

Proof. Assume that H has a feasible solution P with $P[S_i]$ at an end of P. Since $S_i \subseteq S_j$, then $S_i = S_i \cap S_j$, and therefore $P[S_i \cap S_j]$ is at an end of P. Hence, P is a feasible solution for H with $P[S_j]$ at its end.
Assume that H has a feasible solution P with $P[S_j]$ at an end of P. Since s_i is a leaf in $G_{int}(\mathcal{S})$ and $S_i \subseteq S_j$, then S_i is a contained cluster. Since H has a feasible solution, according to Theorem 3.1.9, $H[\mathcal{S} \setminus \{S_i\}]$ has a feasible solution, denote this solution by P'. Since s_i is a leaf, S_i intersects only with S_j, and every vertex $v \in S_i \cap S_j$ satisfies $v \notin \mathcal{S} \setminus \{S_i, S_j\}$. Let $U = S_i \cap S_j$ and $L = \{(U, S_j)\}$. According to Lemma 3.1.7, $P'[V \setminus \{U\}]$ is a feasible solution for $H[\mathcal{S} \setminus \{S_i\}] \setminus L$, denote the new path by P''. Concatenate $P[S_i]$ to the end of P'', adjacent to $P[S_j \setminus S_i]$, denote this path by P^N. $P^N[S_i]$ is obviously a consecutive path. $P^N[S_j]$ is the concatenation of $P''[S_j \setminus S_i]$ and $P^N[S_i]$, and is therefore a consecutive path. Also, for $S_k \in (\mathcal{S} \setminus \{S_i, S_j\})$, $P^N[S_k] = P'[S_k]$, and is therefore a consecutive path. Hence, P^N is a feasible solution for H with $P^N[S_i]$ at its end, and thus the theorem is proved. $\square$

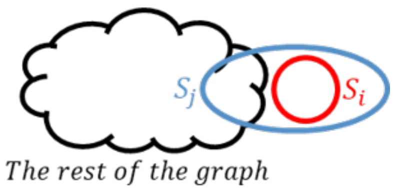

Figure 3.44: The structure of clusters S_i, S_j in Theorem 3.9.3.

Theorem 3.9.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph that has a feasible solution of FCTSP, with intersection graph $G_{int}(\mathcal{S})$. Let $S_i, S_j \in \mathcal{S}$ such that s_i is a leaf in $G_{int}(\mathcal{S})$, $S_i \cap S_j \neq \emptyset$, and $S_i \not\subseteq S_j$. There exists a feasible solution P for H with $P[S_i]$ at an end of P if and only if there exists a feasible solution P' for $H[\mathcal{S} \setminus \{S_i\}]$ with $P'[S_j]$ at an end of P' (see Figure 3.45).*

Proof. Suppose that there exists a feasible solution P for H with $P[S_i]$ at an end of P. Since s_i is a leaf in $G_{int}(\mathcal{S})$ and S_i intersects only with S_j, we can arrange the vertices of $P[S_i]$ in the following way: $(P[S_i \setminus S_j], P[S_i \cap S_j])$. Denote by $P^{V\setminus ij}$ a subpath that spans the vertices of $V \setminus (S_i \cup S_j)$. Since $P[S_i]$ is at an end of P, and $P[S_j]$ is consecutive in P, $P = (P[S_i \setminus S_j], P[S_i \cap S_j], P[S_j \setminus S_i], P^{V\setminus ij}])$. According to Theorem 3.1.5, $P[\mathcal{S} \setminus \{S_i\}]$ is a feasible solution for $H[\mathcal{S} \setminus \{S_i\}]$. The structure of this solution is: $P[S_i \cap S_j], P[S_j \setminus S_i], P^{V\setminus ij}]$. Hence, there is a feasible solution for $H[\mathcal{S} \setminus \{S_i\}]$ with $P[S_j]$ at an end of $P[\mathcal{S} \setminus \{S_i\}]$.
Suppose that there exists a feasible solution P' for $H[\mathcal{S} \setminus \{S_i\}]$ with $P'[S_j]$ at an end of P'. S_i intersects only with S_j, that is, the vertices of $S_i \cap S_j$ belong to these two clusters only in H. Therefore, $nc(S_i \cap S_j) = 1$ in $H[\mathcal{S} \setminus \{S_i\}]$. Denote $L = \{(S_i \cap S_j, S_j)\}$. According to Lemma 3.1.7, $H[\mathcal{S} \setminus \{S_i\}] \setminus L$ has a feasible solution, denote this path by P''. Let $P^{i\setminus j}, P^{i\cap j}, P^{j\setminus i}, P^{V\setminus ij}$ be subpaths which respectively spans $S_i \setminus S_j, S_i \cap S_j, S_j \setminus S_i, V \setminus (S_i \cup S_j)$. Denote $P^N = (P^{i\setminus j}, P^{i\cap j}, P^{j\setminus i}, P^{V\setminus ij})$. Note that $P'' = (P^{j\setminus i}, P^{V\setminus ij})$. That is, P^N is a new path where $P^{i\cap j}$ is concatenated at an end of P'', and $P^{i\setminus j}$ is concatenated adjacent to $P^N[S_j]$. $P^N[S_i]$ is the concatenation of $P^N[S_i \setminus S_j]$ and $P^N[S_i \cap S_j]$, and is therefore a consecutive path. $P^N[S_j]$ is arranged consecutively in P^N. Also, for $S_k \in (\mathcal{S} \setminus \{S_i, S_j\})$, $P^N[S_k] = P'[S_k]$, and therefore is a consecutive subpath. Hence, P^N is a feasible solution for H with $P^N[S_i]$ at an end of P^N. $\square$

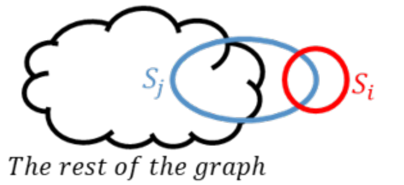

Figure 3.45: The structure of clusters S_i, S_j in Theorem 3.9.4.

Example 3.9.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph with intersection graph $G_{int}(\mathcal{S})$, and there exist $S_i, S_j \in \mathcal{S}$ such that $S_i \subseteq S_j$ and s_i is not a leaf in $G_{int}(\mathcal{S})$. There are cases in which there exists a solution with $P[S_i]$ at an end of P, and cases in which no such solution exists. Figure 3.46 shows examples of such cases. In both cases, $S_i \subseteq S_j$, such that s_i is not a leaf.*
Example 1 in Figure 3.46 shows an example for a case in which there exists a solution with $P[S_i]$ at an end of P. In this example, $P = (2, 3, 4, 1)$ is a feasible solution of FCTSP, and $P[S_i] = (2, 3)$ is at an end of this solution.
Example 2 in Figure 3.46 shows an example for a case in which there is no solution with $P[S_i]$ at an end of P.
Note that Lemma 3.9.1 is another example for a case in which s_i is not a leaf, and there is no feasible solution P with $P[S_i]$ at its end.

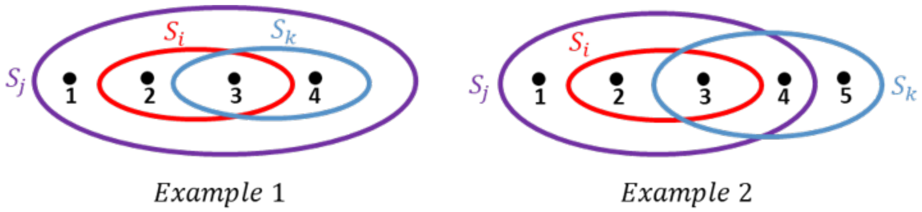

Figure 3.46: Example 3.9.5.

We now present theorems for specific structures of the intersection graph, that have a known cluster at an end of a feasible solution path. First we consider a case in which the intersection graph is a simple path.

Theorem 3.9.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a simple path, and $S_i \in \mathcal{S}$. If there are no contained clusters in H, there exists a feasible solution P of FCTSP with $P[S_i]$ at an end of P if and only if s_i is a leaf in $G_{int}(\mathcal{S})$.*

Proof. According to the definition of a simple path, there exist two nodes that have degree 1 in $G_{int}(\mathcal{S})$, and every other node in $G_{int}(\mathcal{S})$ has degree 2.

Suppose that there exists a feasible solution P for H with $P[S_i]$ at an end of P, and suppose by contradiction that s_i is not a leaf in $G_{int}(\mathcal{S})$, that is, S_i intersects with at least two clusters. Since S_i intersects with two clusters, there are $S_j, S_k \in \mathcal{S}$ such that $S_i \cap S_j \neq \emptyset, S_i \cap S_k \neq \emptyset$. According to Lemma 3.2.2, there are no cliques of size ≥ 3 in $G_{int}(\mathcal{S})$, and therefore $S_j \cap S_k = \emptyset$. If the degree of s_j in $G_{int}(\mathcal{S})$ is 1, since there are no contained clusters, $S_j \nsubseteq S_i$. If the degree of s_j is 2, since S_j intersects with some cluster that is not S_i, and since there are no cliques of size ≥ 3 in $G_{int}(\mathcal{S})$, $S_j \nsubseteq S_i$. Similarly, $S_k \nsubseteq S_i$. Therefore, according to Lemma 3.9.1, there is no feasible solution P with $P[S_i]$ at its end. Hence, if there is a feasible solution with $P[S_i]$ at an end of P, then S_i intersects with exactly one cluster, and s_i is a leaf in $G_{int}(\mathcal{S})$.
Suppose that s_i is a leaf in $G_{int}(\mathcal{S})$, that is, cluster $S_i \in \mathcal{S}$ intersects with exactly one cluster. The degree of s_i in $G_{int}(\mathcal{S})$ is 1, and therefore, according to the definition of a simple path, s_i is at an end of the path which represents the intersection graph. According to Theorem 3.2.5, if the intersection graph $G_{int}(\mathcal{S})$ is a simple path, then H has a feasible solution, denoted by P. This Theorem is based on Algorithm FindPath at Figure 3.3, that finds a feasible solution in which the order of the clusters in P is arranged according to the order of the corresponding nodes in the path of $G_{int}(\mathcal{S})$, and therefore $P[S_i]$ is at an end of P. $\square$

Corollary 3.9.7. *Let $H =< V, \mathcal{S} >$ be a hypergraph, with a connected intersection graph $G_{int}(\mathcal{S})$, and $S_i \in \mathcal{S}$. If H has a feasible solution of FCTSP, and H satisfies the PUC property, then there exists a feasible solution P with $P[S_i]$ at an end of P if and only if S_i intersects with exactly one cluster.*

Proof. According to Theorem 1.0.1, when the intersection graph is connected, and the hypergraph satisfies the *PUC* property, it has a feasible solution of *FCTSP* only if the intersection graph is a path. Therefore, according to Theorem 3.9.6, there exists a feasible solution P with $P[S_i]$ at an end of P if and only if s_i is a leaf in $G_{int}(\mathcal{S})$, and thus S_i intersects with exactly one cluster. $\square$

We now consider a case where the intersection graph is a clique of size $m = 3$.

Lemma 3.9.8. *Let $H =< V, \mathcal{S} = \{S_i, S_j, S_k\} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a clique of size $m = 3$. If there exists a pair of clusters S_i, S_j which satisfies $S_j \subseteq S_i$, then H has a feasible solution P of FCTSP such that $P[S_i]$ is at an end of P (see Figures 3.47 and 3.48).*

Proof. According to Corollary 3.6.7, since $S_j \subseteq S_i$, H has a feasible solution of *FCTSP*. Also, $S_j \cap S_k = S_i \cap S_j \cap S_k$. Therefore, $(S_j \cap S_k) \setminus S_i = \emptyset$, $S_j \setminus (S_i \cup S_k) = \emptyset$. Hence, we can construct P by concatenating the subpaths spanning: $(S_i \setminus (S_j \cup S_k), (S_i \cap S_j) \setminus S_k, S_i \cap S_j \cap S_k, (S_i \cap S_k) \setminus S_j, S_k \setminus (S_i \cup S_j))$, and $P[S_i]$ is at an end of P. □

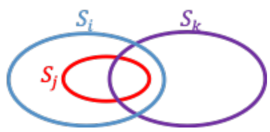

Figure 3.47: The structure of clusters S_i, S_j, S_k in Lemma 3.9.8.

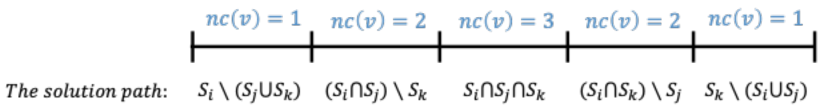

Figure 3.48: The path of the solution in Lemma 3.9.8.

Finally we consider a case where the intersection graph is a star. Note that the following corollary is obtained directly from Theorem 3.9.3.

Corollary 3.9.9. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with k leaves, $k \geq 2$. Denote by s^* the center of the star, and s_i, corresponding to $S_i \in \mathcal{S}$, a leaf in $G_{int}(\mathcal{S})$. If $S_i \subseteq S^*$, then there exists a feasible solution of FCTSP P with $P[S_i]$ at an end of P if and only if there exists a feasible solution P' with $P'[S^*]$ at an end of P'.*

Lemma 3.9.10. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with k leaves. Denote by s^* the center of the star. If $k \geq 2$, and there are no contained clusters in H, then there is no feasible solution of FCTSP P with $P[S^*]$ at its end.*

Proof. Denote by s_i, s_j two leaves in $G_{int}(\mathcal{S})$. Hence, $S^* \cap S_i \neq \emptyset$, $S^* \cap S_j \neq \emptyset$, $S_i \cap S_j = \emptyset$. Since there are no contained clusters, there exist vertices $v_i \in S_i \setminus (S^* \cup S_j)$ and $v_j \in S_j \setminus (S^* \cup S_i)$. Hence, according to Lemma 3.9.1, there is no feasible solution P with $P[S^*]$ at its end. □

Theorem 3.9.11. *Let $H =< V, \mathcal{S} >$ be a hypergraph, whose intersection graph $G_{int}(\mathcal{S})$ is a star with k leaves, $k \geq 2$. Denote by s^* the center of the star, and s_i, corresponding to $S_i \in \mathcal{S}$, a leaf in $G_{int}(\mathcal{S})$. If H has a feasible solution of FCTSP, and $S_i \not\subseteq S^*$, then there exists a feasible solution of FCTSP P with $P[S_i]$ at an end of P.*

Proof. According to Theorem 3.4.8, since H has a feasible solution, the number of leaves which are not contained clusters is $k \leq 2$. According to our assumption, $S_i \nsubseteq S^*$, and therefore s_i is one of those leaves. Note that $S_i \cap S^* \neq \emptyset$. In this case, $H[\mathcal{S} \setminus \{S_i\}]$ is a star with at most one leaf which is not a contained cluster. Denote this leaf by s_j. Therefore, $S_j \setminus S^* \neq \emptyset, S_j \cap S^* \neq \emptyset, S^* \setminus S_j \neq \emptyset$ and are pairwise vertex disjoint. Let $P^{j\setminus *}, P^{j\cap *}, P^{*\setminus j}$ be subpaths which respectively spans $S_j \setminus S^*, S_j \cap S^*, S^* \setminus S_j$. The concatenation of these subpaths, $(P^{j\setminus *}, P^{j\cap *}, P^{*\setminus j})$, is a feasible solution for $H[\mathcal{S} \setminus \{S_i\}]$, with the subpath spanning S^* at an end of it. Furthermore, According to Theorem 3.1.9, $H[\mathcal{S} \setminus \{S_i\}]$ has a feasible solution with the contained clusters also. Hence, according to Theorem 3.9.4, H has a feasible solution P with $P[S_i]$ at its end. $\square$

Section 4

Booth and Lueker Adjusted Algorithms

In this section we present two algorithms that uses PQ-Trees (see Definition 4.0.1) and are based on the algorithm of Booth and Lueker introduced in [2]. These algorithms can be used on any hypergraph, and they are not based on the intersection graph of the hypergraph.

Let $H =< V, \mathcal{S} >$ be a hypergraph. The algorithm of Booth and Lueker uses the structure of a PQ-Tree to decide if there is a permutation of the vertices of V, such that each cluster $S \in \mathcal{S}$ is consecutive in this permutation. The algorithm can be used to find all the permutations which satisfy that $\forall S \in \mathcal{S}$, S is consecutive in this permutation.

Definition 4.0.1. PQ-Tree: Given a universal set $U = \{a_1, a_2, \ldots, a_m\}$, the class of PQ-Trees over that set is defined to be all rooted ordered trees that meet the conditions below. This data structure is introduces by Booth and Lueker in [2], and is used in the algorithm presented by the authors, which finds a feasible solution of *COP*.
The leaves are elements of U, such that every element $a_i \in U$ appears exactly once as a leaf. A leaf is denoted by L-Node, and is drawn as a triangle. The internal (nonleaf) nodes are distinguished as being either P-Nodes or Q-Nodes:

- A P-Node is drawn as a circle, and its children can appear in any order. Each P-Node has at least two children.

- A Q-Node is drawn as a rectangle, and its children can only be reversed. Each Q-Node has at least three children.

The permutations of this PQ-Tree are constructed by all the possible and legal orders of the children of the P-Nodes and Q-Nodes in the tree.

Definition 4.0.2. Let $H =< V, \mathcal{S} >$ be a hypergraph, and $S \in \mathcal{S}$ the cluster that is currently being processed by Booth-Lueker Algorithm.
A leaf $v \in V$ in the PQ-Tree is $full$ if $v \in S$.
A leaf $v \in V$ in the PQ-Tree is *empty* if $v \notin S$.
An internal node in the PQ-Tree is $full$ if all of its descendants are $full$.
An internal node in the PQ-Tree is *empty* if all of its descendants are *empty*.
An internal node in the PQ-Tree is *partial* if some of its descendants are $full$ and some of its descendants are *empty*. Note that the type of a *partial* node is always a Q-Node. If a P-Node needs to be marked as a *partial* node, it is transformed to be a Q-Node.
Figure 4.1 shows the symbols we will use later in this section to draw PQ-Trees.

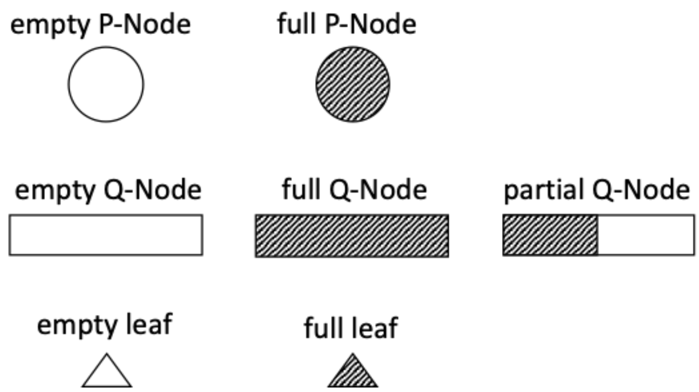

Figure 4.1: PQ-Tree symbols.

Booth-Lueker's Algorithm has two main procedures, which are applied on each one of the clusters. Both procedures scan the PQ-Tree bottom-up. The first procedure, called *BUBBLE*, marks the nodes in the PQ-Tree that need to be checked while processing the current cluster $S \in \mathcal{S}$. The marked nodes are all the nodes included in the minimal connected subtree which contains all the leaves of S. The purpose is to avoid processing the entire tree while processing one specific cluster. The second procedure, called *REDUCE*, processes the relevant nodes, that were marked by the *BUBBLE* procedure, and tries to match to these nodes one template from a given set of templates. When a template matches the current node and its descendants, the procedure performs an update on the structure of the tree, which is appropriate for this template.

Let $H =< V, \mathcal{S} >$ be a hypergraph, and T the PQ-Tree constructed after Booth-Lueker Algorithm processed all of the clusters in $\mathcal{S}$. If Booth-Lueker Algorithm ended successfully, the appropriate permutations of the leaves of T represent the possible feasible solutions for H of *FCTSP*. The path of each solution is defined by the order of the leaves, according to the legal orders of the children of the P-Nodes and Q-Nodes in T.

Example 4.0.3. *Let $H =< V, \mathcal{S} >$ be a hypergraph. In Figure 4.2 a PQ-Tree is constructed for the given hypergraph. The figure shows the templates that are matched during the REDUCE procedure on each of the clusters. The order in which the clusters are processed is S_1 and then S_2. Figure 4.3 shows the final PQ-Tree for H, and the appropriate permutations of the leaves in this PQ-Tree.*

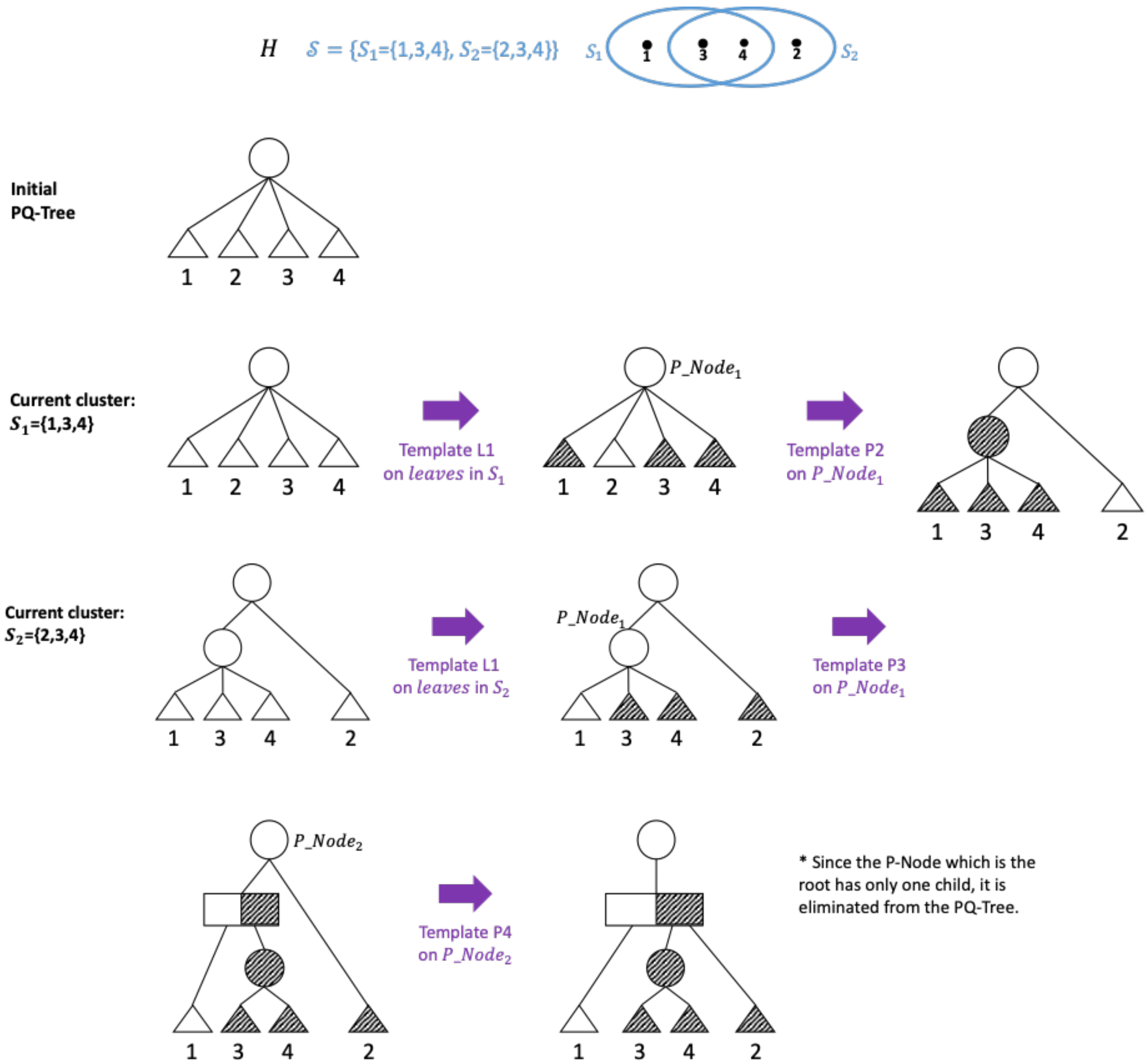

Figure 4.2: Constructing PQ-Tree for Example 4.0.3.

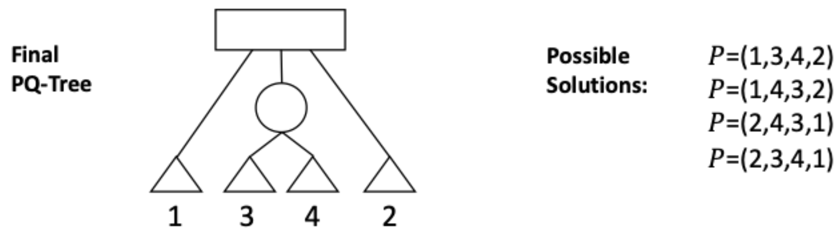

Figure 4.3: Final PQ-Tree and allowed permutations for Example 4.0.3.

4.1 Extended Booth-Lueker Algorithm

We present an Extended Booth-Lueker Algorithm, which finds a feasible removal list, based on Booth-Lueker Algorithm presented in [2], in linear time. Booth-Lueker Algorithm uses several tamplates that are described in [2]. At each step, the algorithm tries to match one of the templates to the current node and its descendants, and arrange them according to the relevant template. If there is no matching template, then the algorithm stops and returns that there is no feasible solution.
In Extended Booth-Lueker Algorithm, at each step, if there is no matching template, then the algorithm changes the current node and its descendants, by removing vertices from the processed cluster, in order to match one of the templates.

Let $H =< V, \mathcal{S} >$ be a hypergraph, and $S \in \mathcal{S}$ the cluster that is currently being processed by Booth-Lueker Algorithm. A removal from S can be done by transforming nodes to be *empty*, such that after this transformation the current node and its descendants will match one of Booth-Lueker templates. In order to transform a node to be *empty*, we remove vertices from S. Removing a vertex v from S transforms the corresponding leaf to be an *empty* node, and might also transform the ancestors of this leaf to be *empty* nodes. Note that this change might also transform the ancestors of v to be *partial* nodes.

Example 4.1.1. *Figure 4.4 shows an example for a node which does not have a feasible solution, since the $full$ nodes are not consecutive in the partial Q-Node. There are several ways to change this node, by removing vertices from the current cluster. For example. we can remove vertices* 2 *and* 4 *from the current cluster. After this removal, leaves* 2 *and* 4 *are empty, and the structure of the node matches Template* $Q0$.

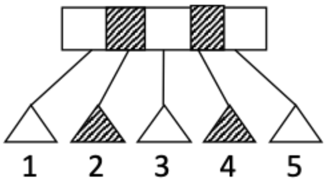

Figure 4.4: Example 4.1.1.

Remark 4.1.2. *Note that it is also possible to change the current node and its descendants by transforming nodes to be $full$. In order to transform a node to be $full$, we add vertices to S. Adding a vertex v to S transforms the corresponding leaf to be a $full$ node, and might also transform the ancestors of this leaf to be $full$ nodes. There are cases in which the minimum change is adding vertices to the cluster, and cases in which the minimum change is removing vertices from the cluster. However, this is beyond the scope of this work, and our algorithm only removes vertices from clusters.*

A general description of Extended Booth-Lueker Algorithm, based on the necessary conditions for matching relevant templates, is as follows. The algorithm processes all of the clusters, according to Algorithm Booth-Lueker. If there is no matching template for some node and Algorithm Booth-Lueker stops, then a local removal is made on the current node and its subtree, according to the conditions described in Tables 4.1, 4.2 and 4.3, presented later on. The removal is made recursively on the current node and its children. After the removal, we continue to run Booth-Lueker Algorithm on the current cluster. For efficiency of Extended Booth-Lueker Algorithm, as shown in Theorem 4.1.7, when changing a specific node in the algorithm, we transform all of its descendants to be *empty* nodes. We do not consider removals which transform a $full$ node to be a *partial* node.

The following ExtendedBLP5 Algorithm is part of the offered Extended Booth-Lueker Algorithm. This part is suitable for the case when Booth-Lueker Algorithm tries to match a P-Node to template $P5$. In this template, the current node is not the root of the minimal subtree which contains all the vertices of the current cluster, and it has one *partial* child.
Furtermore, we also present here EmptyNode Algorithm, that is a recursive function used to remove vertices from the current cluster. EmptyNode Algorithm is used when Extended Booth-Lueker Algorithm decides to transform a node to be *empty*. The transformation is made recursively on the node and its descendants.

Algorithm 11 ExtendedBLP5: A section of Extended Booth-Lueker Algorithm, for Template $P5$

function EXTENDEDBLP5()

Input:
A PQ-Tree T for hypergraph $H =< V, \mathcal{S} >$.
A cluster $S \in \mathcal{S}$ that is the current cluster processed by the algorithm.
The node that is currently processed by the algorithm.
Output:
An updated PQ-Tree and a cluster S' constructed after the removal of vertices from S.
Assumptions:
The current node processed by the algorithm is a P-Node.
The current node is not the root of the minimal subtree which contains all the vertices of the current cluster.
The current node has one *partial* child.

begin
    The relevant template to check is Template $P5$.
    Denote $currNode$ = The node currently processed by the algorithm.
    if $currNode$ has children which are $full$ nodes :
        if The *empty* children of the *partial* are not on one side of it :
            Denote $partNode$ = The *partial* child of $currNode$.
            Let $S' = EmptyNode(T, S, partNode)$.
            Check Templates $P1, P3$, which are relevant for 0 $partial$.
        end if
    end if
    return S' and the updated PQ-Tree T.
end function

Figure 4.5: Algorithm ExtendedBLP5

Algorithm 12 EmptyNode: A recursive function that removes vertices from the current cluster in order to transform a node to be *empty*

function EMPTYNODE()

Input:
A PQ-Tree T for hypergraph $H =< V, \mathcal{S} >$.
A cluster $S \in \mathcal{S}$ that is the current cluster processed by the algorithm.
A node in the PQ-Tree.
Output:
An updated PQ-Tree and a cluster S' constructed after the removal of vertices from S.

begin
  Set $S' = S$.
  Denote $currNode$ = The node that is transformed to be *empty*.
  Set the label of $currNode$ to be *empty*.
  if The type of $currNode$ is L-Node :
    Denote v = The corresponding vertex represented by the leaf.
    if $v \in S$:
      Set $S' = S' \setminus \{v\}$.
    end if
  else
    Denote $childFull$ = A list of the $full$ children of $currNode$.
    Denote $childPartial$ = A list of the *partial* children of $currNode$.
    for each *child* in $(childFull \cup childPartial)$:
      Let $S' = EmptyNode(T, S', child)$.
    end for
  end if
  return S' and the updated PQ-Tree T.
end function

Figure 4.6: Algorithm EmptyNode

Example 4.1.3. *Figure 4.7 shows a possible run of Extended Booth-Lueker Algorithm, on the given hypergraph. In this example, the order in which we process the clusters is S_1, S_2 and then S_3. While processing cluster S_3, the appropriate PQ-Tree is shown in the bottom-right part of the figure. Since there are 3 partial nodes, there is no matching template for the Q-Node which is the root of the tree. Therefore, according to Extended Booth-Lueker Algorithm, transform one of the partial nodes to be an empty node. Suppose* *the algorithm transforms the left partial node to be an empty node, that is, we remove vertex* 2 *from cluster* S_3*. This removal is demonstrated in Figure 4.8. Note that the left partial Q-Node is transformed to be a P-Node, since at the end of the algorithm, each Q-Node should have at least three children. After this removal, there are* 2 *partial nodes with a valid structure, and the Q-Node, which is the root of the tree, matches Template* $Q3$*. Figure 4.8 shows the final PQ-Tree for* H'*, and the appropriate permutations of the leaves in this PQ-Tree.*

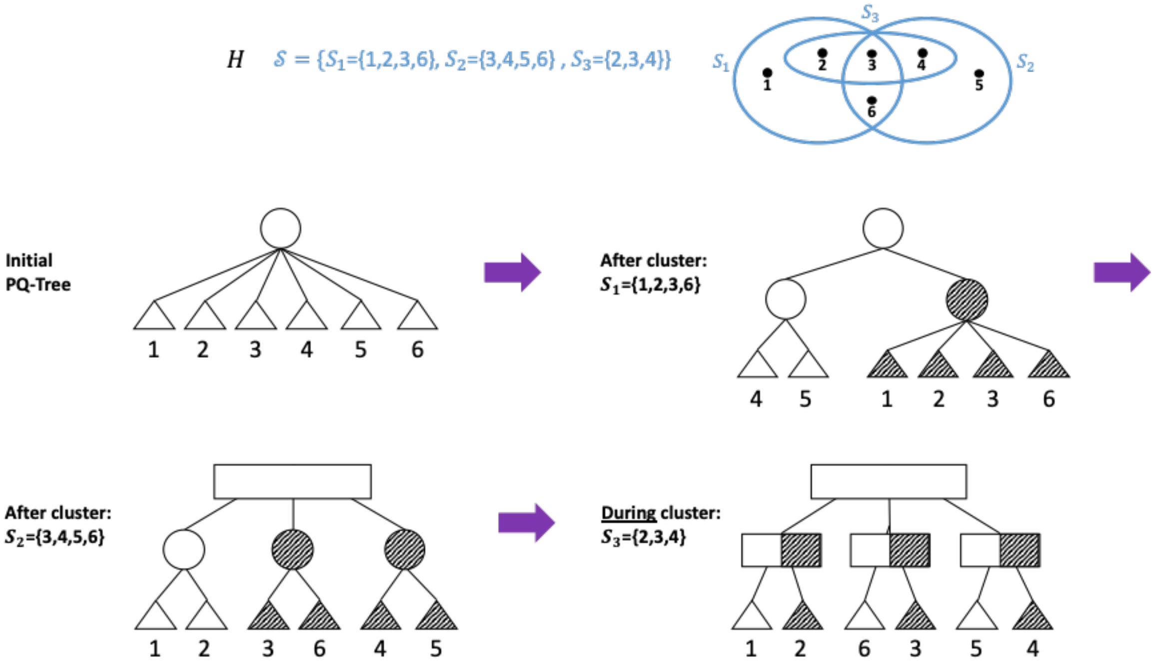

Figure 4.7: Constructing PQ-Tree for Example 4.1.3.

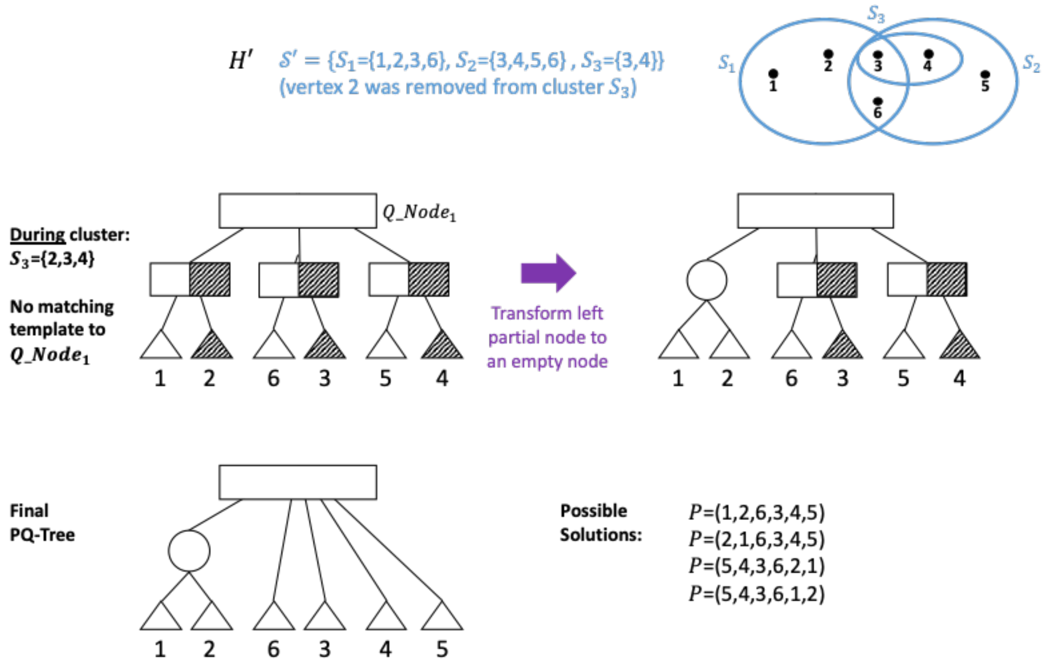

Figure 4.8: Removal and final PQ-Tree for Example 4.1.3.

Lemma 4.1.4. *Let $H =< V, \mathcal{S} >$ be a hypergraph. Let $S_1, \dots, S_i \in \mathcal{S}$ be the clusters which were already processed by the algorithm, and denote by T the PQ-Tree constructed for these clusters. Let $S' \in \mathcal{S} \backslash \{S_1, \dots, S_i\}$ be the cluster that is currently being processed by Extended Booth-Lueker Algorithm, and T' the PQ-Tree after processing S'. If $S_1, \dots, S_i$ have consecutive subpaths in all the appropriate permutations of the leaves in T, then $S_1, \dots, S_i$ also have consecutive subpaths in all the appropriate permutations of the leaves in T'.*

Proof. Booth and Lueker prove in [2] that at the and of their algorithm, if the algorithm ended successfully, then each cluster has a consecutive subpath. The algorithm constructs the PQ-Tree, such that after processing a cluster $S' \in \mathcal{S}$, this cluster has a consecutive subpath, which stays consecutive when processing other clusters. Therefore, after processing S', if there is a feasible solution with S', clusters $S_1, \dots, S_i$ which were processed before S' still have consecutive subpaths in all the appropriate permutations of the leaves in T'. Removing vertices from S' in Extended Booth-Lueker Algorithm, does not change the structure of the PQ-Tree, but only the "color" of the leaves and internal nodes from *full* or *partial* to *empty*. Since the structure of the PQ-Tree is not changed, the appropriate permutations of the leaves are also not changed. Therefore, removing vertices from S' does not change the subpaths already constructed for previous clusters. After the appropriate removals by Extended Booth-Lueker Algorithm, changes to the PQ-Tree structure are made only by the Booth-Lueker templates. Hence, according to the proof of Booth-Lueker Algorithm, if $S_1, \ldots, S_i$ have consecutive subpaths in all the appropriate permutations of the leaves in T, then $S_1, \ldots, S_i$ also have consecutive subpaths in all the appropriate permutations of the leaves in T', even after removing vertices from S'. $\square$

Theorem 4.1.5. *Let $H =< V, \mathcal{S} >$ be a hypergraph, and $S_i \in \mathcal{S}$. Denote $S_i{}' = S_i \setminus L_i$, where L_i is the removal list constructed by Extended Booth-Lueker Algorithm for cluster S_i. Denote $H' =< V, \mathcal{S}' >$, such that $\mathcal{S}' = S_1{}', \ldots, S_m{}'$. Extended Booth-Lueker Algorithm finds a feasible solution for H' of FCTSP.*

Proof. Let T be the PQ-Tree constructed after Extended Booth-Lueker Algorithm processed all of the clusters in $\mathcal{S}$. Extended Booth-Lueker Algorithm verifies that each cluster is processed successfully by the algorithm, by removing vertices when needed, in order to match one of the known templates. Therefore, the algorithm ends successfully, after processing all the clusters. According to Lemma 4.1.4, $\forall S' \in \mathcal{S}'$, S' has a consecutive subpath in all the appropriate permutations of the leaves in T. Hence, T represents a feasible solution for H' of *FCTSP*. $\square$

Since the changes of Extended Booth-Lueker Algorithm are made locally on the current cluster, the order in which we process the clusters affects the number of changes that are made by the algorithm, as shown in Example 4.1.6.

Example 4.1.6. *Let $H =< V, \mathcal{S} >$ be a hypergraph, such that $\mathcal{S} = \{S_1, S_2, S_3\}$.*

(a) If the order in which the clusters are processed is S_1, S_2, S_3, when processing cluster S_3, there is no matching template for the Q-Node marked in red (see Figure 4.9). Therefore, according to Extended Booth-Lueker Algorithm, we transform at least one of the partial nodes to be an empty node. The minimum removal is to transform the left partial node to be an empty node, that is, removing vertex 1 from cluster S_3. After the removal, the marked Q-Node matches Template $Q2$.
At the end of Extended Booth-Lueker Algorithm,
$Path = (2, 3, 4, 1, 5, 6, 7, 8, 9, 10, 11, 12)$ is a feasible solution for H' of FCTSP, and the number of removals is 1.

(b) *If the order in which the clusters are processed is S_1, S_3, S_2, when processing cluster S_2, there is no matching template for the Q-Node marked in red (see Figure 4.10). Therefore, according to Extended Booth-Lueker Algorithm, we transform at least one of the partial nodes to be an empty node. The minimum removal is to transform the left partial node to be an empty node, that is, removing vertices $\{5, 6\}$ from cluster S_2. After the removal, the marked Q-Node matches Template Q2.*
At the end of Extended Booth-Lueker Algorithm,
$Path = (2, 3, 4, 5, 6, 1, 12, 9, 10, 11, 7, 8)$ is a feasible solution for H' of FCTSP, and the number of removals is 2.

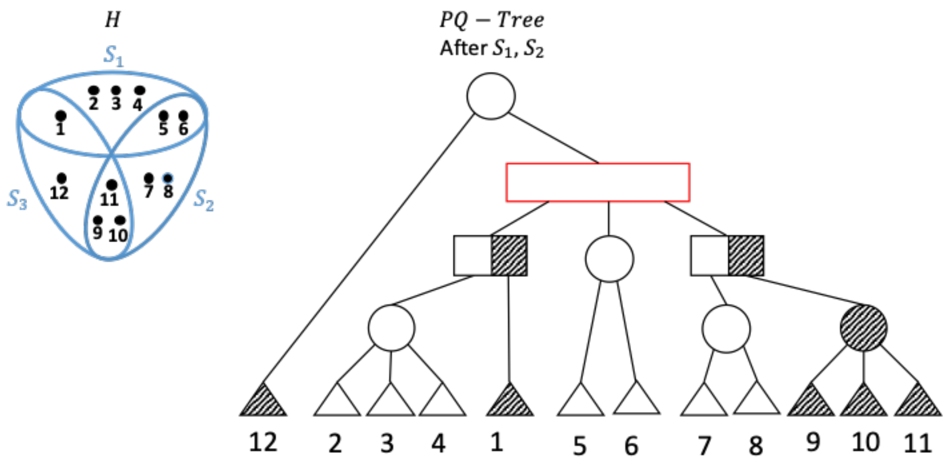

Figure 4.9: Example 4.1.6(a).

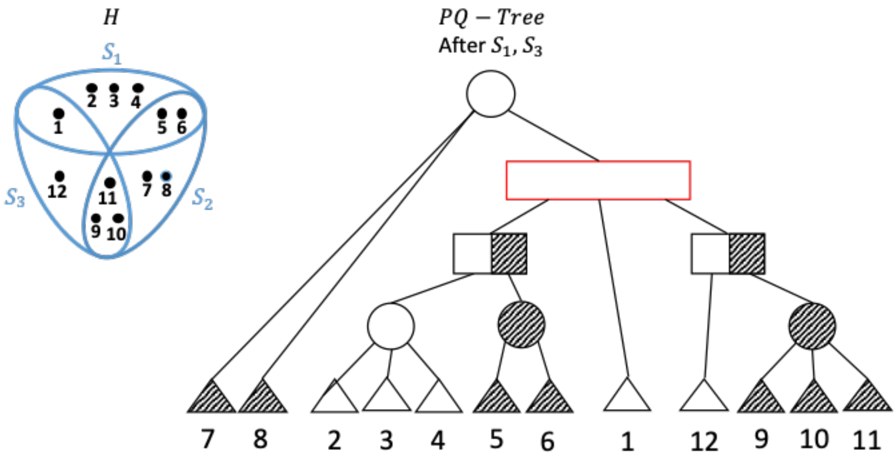

Figure 4.10: Example 4.1.6(b).

Theorem 4.1.7. *The time complexity of Extended Booth-Lueker Algorithm is $\mathcal{O}(n + m + \sum_{1\leq i\leq m} |S_i|)$, where $n = |V|, m = |\mathcal{S}|$.*

Proof. In [2] it is shown that the time complexity of Booth-Lueker Algorithm is $\mathcal{O}(n + m + \sum_{1\leq i\leq m} |S_i|)$. According to the description of Booth-Lueker Algorithm in [2], each node in the PQ-Tree has a list of its $full$ children and a list of its *partial* children. In Extended Booth-Lueker Algorithm, when there is no matching template, we transform some of the descendants of this node from $full$ or *partial* nodes to be *empty* nodes. Since the descendants of an *empty* node are also *empty*, it is sufficient to parse only the children that are $full$ or *partial*. This can be done recursively on the $full$ children and *partial* children lists and their descendants. Extended Booth-Lueker Algorithm makes a constant number of actions on each node during the removal, and each node is removed at most once while processing a cluster. Therefore, in the worst case, we multiply the number of nodes that are processed for each cluster by 2. Hence, the total time complexity of Extended Booth-Lueker Algorithm is $\mathcal{O}(n + m + \sum_{1\leq i\leq m} |S_i|)$. □

The following three tables show necessary conditions for matching the relevant templates, when running the algorithm on a specific cluster. Those conditions are determined by the number of *partial* nodes in the current subtree, and whether the current node is the root of the minimal subtree which contains all the vertices of the current cluster. If there is no matching template, we offer a possible simple removal, according to the conditions described. Extended Booth-Lueker Algorithm is based on these conditions and removals.

Description of the columns in the tables:
Partial Num - The number of *partial* nodes in the current subtree.
Is Root - Is the current node the root of the minimal subtree which contains all the vertices of the current cluster, or not.
Templates - The relevant Booth-Lueker templates to match, according to the number of *partial* nodes and whether the current node is the root or not.
Is Valid - Does the current subtree have a valid structure that may match one of the templates.
Conditions - The conditions for matching the current subtree to one of Booth-Lueker templates.
Removal - A possible simple removal.
The are cases in which it is not relevant to give a specific value to a cell in the table.

The table of the cases that require a removal when the current node is a L-Node, which is a leaf in the PQ-Tree:

Partial Num	**Is Root**	**Template**	**Is Valid**	**Conditions**	**Removal**
Not relevant	Not relevant	$L1$	Yes	All structures for a leaf are valid	Not required

Table 4.1: L-Node cases with possible removals.

The table of the cases which require a removal when the current node is a P-Node:

Partial Num	**Is Root**	**Template**	**Is Valid**	**Conditions**	**Removal**
0	Yes	$P0, P1, P2$	Yes	Since there is no significance to the order of the children, all structures are valid	Not required
0	No	$P0, P1, P3$	Yes	Since there is no significance to the order of the children, all structures are valid	Not required

Partial Num	**Is Root**	**Template**	**Is Valid**	**Conditions**	**Removal**
1	Yes	$P4$	Depends on *partial* structure	Considering the structure of the *partial* node, if there are children of the current node which are $full$, then the *empty* children of the *partial* node should be on one side of it. If all of the children of the current node are *empty* (except from the *partial* node), then the *empty* children of the *partial* node may appear on both sides of it.	Transform the *partial* node to be an *empty* node, and then try to match Templates $P0, P1, P2$.
1	No	$P5$	Depends on *partial* structure	Considering the structure of the *partial* node, if there are children of the current node which are $full$, then the *empty* children of the *partial* node should be on one side of it.	Transform the *partial* node to be an *empty* node, and then try to match Templates $P0, P1, P3$.
2	Yes	$P6$	Depends on *partial* structure	Considering the structure of the *partial* nodes, in both *partial* nodes, the *empty* children of the *partial* node should be on one side of it, so one can connect the two parts.	Transform the *partial* nodes which have an invalid structure to be *empty* nodes, and then try to match Template $P0, P1, P2, P4$.

Partial Num	**Is Root**	**Template**	**Is Valid**	**Conditions**	**Removal**
2	No	None	No	There are no matching templates for this case.	Transform one of the *partial* nodes to be an *empty* node, and then make removals from the other *partial* node if needed, by trying to match Template $P5$.
3 or more	Not relevant	None	No	There are no matching templates for this case.	Transform one of the *partial* nodes to be an *empty* node, and repeat this step while there are at least three *partial* nodes. Then, make removals from the remaining two *partial* nodes if needed, according to the cases of two *partial* nodes, and Template $P6$ if the current node is the root.

Table 4.2: P-Node cases with possible removals.

The table of the cases which require a removal when the current node is a Q-Node:

Partial Num	**Is Root**	**Template**	**Is Valid**	**Conditions**	**Removal**
0	Not relevant	$Q0, Q1$	Depends on children order	All the $full$ children of the current node should appear consecutively, and the *empty* children of the current node may appear on both sides of the $full$ sequence.	Transform the current node to be an *empty* node.

Partial Num	Is Root	Template	Is Valid	Conditions	Removal
1	Yes	$Q2$	Depends on children order	All the *full* children of the current node should appear consecutively, adjacent to the *full* side of the *partial* node. The *empty* children of the current node may appear on both sides of the sequence of *full* children. Also, considering the structure of the *partial* node, if the current node have *full* children, then the *empty* children of the *partial* node should be on one side of it.	If at least one of these conditions is not met, then we transform the current node to be an *empty* node, and then try to match Templates $Q0, Q1$.
1	No	$Q2$	Depends on children order	All the *full* children of the current node should appear consecutively, adjacent to the *full* side of the *partial* node. All the *empty* children of the current node should appear consecutively, adjacent to the *empty* side of the *partial* node. Also, considering the structure of the *partial* node, if the current node have *full* children, then the *empty* children of the *partial* node should be on one side of it.	If at least one of these conditions is not met, then we transform the current node to be an *empty* node, and then try to match Templates $Q0, Q1$.

Partial Num	Is Root	Template	Is Valid	Conditions	Removal
2	Yes	$Q3$	Depends on children order	All the $full$ children of the current node should appear consecutively, between both *partial* nodes. The *empty* children of the current node should appear on the outer sides of the *partial* nodes. Also, considering the structure of the *partial* nodes, in both *partial* nodes, the *empty* children of the *partial* node should be on one side of it, so one can connect the two parts.	If at least one of these conditions is not met, then we transform one or both of the *partial* nodes to be *empty* nodes, and then try to match Templates $Q0, Q1, Q2$.
2	No	None	No	There are no matching templates for this case.	Transform one of the *partial* nodes to be an *empty* node, and then make removals from the other *partial* node and the children of the current node if needed, by trying to match Template $Q2$.

Partial Num	**Is Root**	**Template**	**Is Valid**	**Conditions**	**Removal**
3 or more	Not relevant	None	No	There are no matching templates for this case.	Transform one of the *partial* nodes to be an *empty* node, and repeat this step while there are at least three *partial* nodes. Then, make removals from the remaining two *partial* nodes if needed, according to the cases of two *partial* nodes, and Template $Q3$ if the current node is the root.

Table 4.3: Q-Node cases with possible removals.

Appendix .1 shows a basic implementation of Extended Booth-Lueker Algorithm. This implementation is based on the PQ-Tree module in Tyralgo library, which is an Open Source Library for Algorithmic Problem Solving, released under the MIT License.

4.2 Booth-Lueker Algorithm for a Known End of Path

In this section we present Algorithm KnownEndBL, that finds a feasible solution with a known cluster at an end of it, if such a solution exists, in linear time. This algorithm is also based on Booth-Lueker Algorithm presented in [2]. Algorithm KnownEndBL initializes a PQ-Tree in which the vertices of the known cluster at an end are descendants of one P-Node, and the rest of the vertices are descendants of another P-Node. We then use Booth-Lueker Algorithm to process the clusters of the hypergraph, except for the cluster that is a known end of the solution path.

Algorithm 13 KnownEndBL: Adjusted Booth-Lueker Algorithm which finds a feasible solution with a known cluster at an end of it, if such a solution exists.

function KnownEndBL()

Input:
A hypergraph $H =< V, \mathcal{S} >$.
A cluster $S' \in \mathcal{S}$.
Output:
A PQ-Tree with S' at an end of the corresponding feasible solution if the algorithm ended successfully, or an indication that there is no feasible solution with S' at an end of it.

begin
Initialize a PQ-Tree, denoted by T, with the following structure:
1. The root of the tree is a P-Node, denoted by *root*, with two children.
2. The first children of *root* is a P-Node, denoted by $P1$.
3. The children of $P1$ are the vertices in S'.
4. The second children of *root* is a P-Node, denoted by $P2$.
5. The children of $P2$ are the vertices in $V \setminus S'$.

Call Booth-Lueker Algorithm on $(H[\mathcal{S} \setminus \{S'\}], T)$.
return The result of Booth-Lueker Algorithm.
end function

Figure 4.11: Algorithm KnownEndBL

Example 4.2.1. *Figure 4.12 shows an example for KnownEndBL Algorithm, on the given hypergraph. The first tree is the PQ-Tree which is constructed at the beginning of the algorithm, with cluster S_1 as a known end of the solution path. The second tree is the PQ-Tree constructed by Booth-Lueker Algorithm, when called on the first PQ-Tree and clusters $\{S_2, S_3\}$. At the end of the algorithm, $Path = (1, 2, 3, 4, 5, 6, 7)$ is a feasible solution for H of FCTSP, and cluster S_1 is at an end of this solution.*

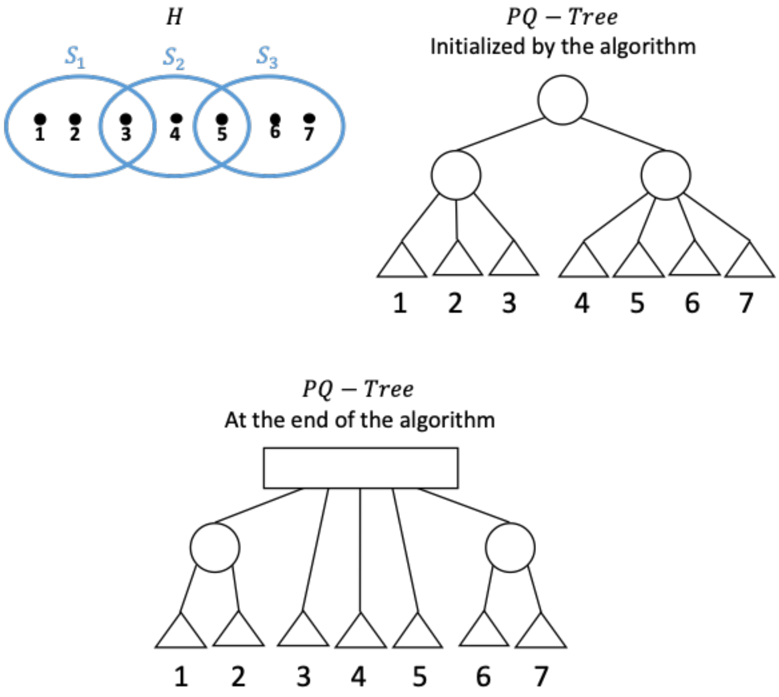

Figure 4.12: Example 4.2.1.

Theorem 4.2.2. *Algorithm KnownEndBL finds a feasible solution of FCTSP with S' at an end of it, if such a solution exists.*

Proof. If we call Booth-Lueker Algorithm on a set of two disjoint clusters S_1, S_2, then we get a PQ-Tree in which each of those clusters is under a different P-Node, denote these P-Nodes by P_1, P_2. Booth-Lueker Algorithm assures that at the end of the algorithm, S_1 and S_2 have a consecutive subpath in the solution defined by the PQ-Tree. Algorithm KnownEndBL creates a structure which is equivalent to this, by separating between S', which is a known end of the solution path, and the rest of the vertices. That is, S' is equivalent to S_1, and the P-Node which contains the rest of the vertices is equivalent to S_2. Therefore, if the algorithm ended successfully, S' has a consecutive subpath in the solution. Also, all the vertices under P_2 have a consecutive subpath in the solution, and the groups of vertices under P_1, P_2 are disjoint. Hence, the subpath defined by P_1 cannot appear inside the subpath defined by P_2, and if there is a feasible solution, then S' is at an end of this solution. □

Theorem 4.2.3. *Algorithm KnownEndBL runs in $\mathcal{O}(n + m + \sum_{1 \le i \le m} |S_i|)$ time complexity, where $n = |V|, m = |\mathcal{S}|$.*

Proof. In [2] it is shown that the time complexity of Booth-Lueker Algorithm is $\mathcal{O}(n + m + \sum_{1 \leq i \leq m} |S_i|)$. Algorithm KnownEndBL is the same as Booth-Lueker Algorithm, with an extra step for creating the initial tree. Hence, the time complexity of the algorithm is $\mathcal{O}(n + m + \sum_{1 \leq i \leq m} |S_i|)$. $\square$

Appendix .2 shows a basic implementation of Booth-Lueker Algorithm for a Known End of Path. This implementation is based on the PQ-Tree module in Tyralgo library, which is an Open Source Library for Algorithmic Problem Solving, released under the MIT License.

Section 5

Summary and Future Research

In this work we focus on the Feasibility Clustered Travelling Salesman Problem (*FCTSP*) with non-disjoint clusters.

We characterize cases where there is no feasible solution for the hypergraph, according to the structure of its intersection graph. In those cases, we present several algorithms which find a minimum cardinality feasible removal list for the hypergraph. After finding a feasible removal list, we can use other algorithms to find the feasible solution for the new hypergraph. Let $H =< V, \mathcal{S} >$ be a hypergraph and L a minimum cardinality feasible removal list for H. We can use the algorithm of Booth and Lueker to find a feasible solution for $H \setminus L$. We also present theorems for cases where there is a feasible solution for a hypergraph with known ends of the solution path. Those theorems are useful while desigining algorithms that find a feasible removal list for hypergraphs whose intersection graphs contain a cut edge or a cut node, and may also be useful for other known structures of the intersection graph.

Another significant result of this research is the designing of an algorithm that finds a feasible removal list for any hypergraph in linear time. In addition, we present an algorithm that finds a feasible solution with a known end of the solution path, if such a solution exists, in linear time. Both algorithms are based on the algorithm of Booth and Lueker.

Summary of our main results: Given a hypergraph $H =< V, \mathcal{S} >$, where $n = |V|, m = |\mathcal{S}|$:

1. **Simple Path:** The hypergraph always has a feasible solution. Algorithm FindPath finds a feasible solution in $\mathcal{O}(m^2+nm)$ time complexity.

2. **Chordless Cycle:** If the size of the cycle is $m \geq 4$ then there is no feasible solution. Algorithm DelCycle finds a minimum cardinality feasible removal list in $\mathcal{O}(m^2 + n)$ time complexity.

3. **Tree:** If the tree is not a simple path, and there are no contained clusters, then there is no feasible solution.

4. **Star:** If the star contains $k \geq 3$ leaves, then there is no feasible solution. Algorithm DelStar finds a minimum cardinality feasible removal list in $\mathcal{O}(mn)$ time complexity.

5. **Star with Paths:** If the star contains $k \geq 3$ paths, then there is no feasible solution. Algorithm DelStarWithPaths finds a minimum cardinality feasible removal list in $\mathcal{O}(nm^2)$ time complexity.

6. **Caterpillar Tree:** If the tree is not a simple path, then there is no feasible solution. Algorithm DelCaterpillar is a dynamic programming algorithm which finds a feasible removal list in $\mathcal{O}(nm^3)$ time complexity.

7. **Bipartite Graph:** If the bipartite graph is isomorphic to $K_{x,y}$, $x \geq 2, y \geq 2$ or to $K_{1,y}$, $y \geq 3$, then there is no feasible solution. Algorithm DelBipartite finds a feasible removal list in $\mathcal{O}(nm^3)$ time complexity.

8. **Clique:** We characterize when there exists a feasible solution for a clique of size $m = 3$. Algorithm DelClique3 finds a minimum cardinality feasible removal list for a clique of size $m = 3$ in $\mathcal{O}(n)$ time complexity.

9. **Cut Edge:** There exists a feasible solution if and only if each connected component has a feasible solution with the appropriate node of the cut edge at its end. Algorithm DelCutEdge finds a minimum cardinality feasible removal list.

10. **Cut Node:** If there are three connected components which are not contained in the cut node, then there is no feasible solution. Algorithm DelCutNode finds a minimum cardinality feasible removal list.

11. **General Graph:** In the general case, for any given hypergraph, Extended Booth-Lueker Algorithm finds a feasible removal list in linear time, $\mathcal{O}(n+m+\sum_{1\leq i\leq m}|S_i|)$. This algorithm is based on the algorithm of Booth and Lueker.

12. **Known Endpoint:** Algorithm KnownEndBL finds a feasible solution with a known endpoint, if such a solution exists, in linear time,

$\mathcal{O}(n + m + \sum_{1 \leq i \leq m} |S_i|)$. This algorithm is based on the algorithm of Booth and Lueker.

There are several different directions for further research in this field. This work focus on vertices removal from clusters in the hypergraph, in order to achieve feasibility. We would like to explore possible algorithms that insert vertices to clusters in the hypergraph, in order to achieve feasibility.

We would also like to explore different target functions. For example, we can design algorithms that use a target function which minimizes the number of vertices that are changed, or a target function which minimizes the number of clusters that are changed. Furthermore, in the algorithms we presented, a vertex may be removed from the graph, if it is removed from all the clusters that contained it. We would like to design algorithms that make sure that no vertex is removed from the graph.

Another direction for research is to explore algorithms which find a minimum cardinality feasible removal list for intersection graphs of other known graph structures. Other algorithms may also rely on additional theorems and conditions for known ends of the solution path.

In addition, we can also deepen the study of algorithms that are based on the algorithm of Booth and Lueker, and explore a proper order for processing clusters, and advanced algorithms for removals, in order to find a minimum cardinality feasible removal list.

References

[1] S. Anily, J. Bramel, and A. Hertz. A 5/3-approximation algorithm for the clustered traveling salesman tour and path problems. *Operations Research Letters*, 24:29–35, 1999.

[2] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[3] T. Christof, M. Jünger, J. Kececioglu, P. Mutzel, and G. Reinelt. A branch-and-cut approach to physical mapping of chromosomes by unique end-probes. *Journal of Computational Biology*, page 4:433–47, 1997.

[4] N. ChristofIdes. Worst-case analysis of a new heuristic for the travelling salesman problem. *Management Sciences Research Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University*, 1976.

[5] P. Duchet. Propriété de helly et problémes de représentation. *Colloqu. Internat. CNRS, Problemes Conbinatoires et Theorie du Graphs, Orsay, France*, 260:117–118, 1976.

[6] C. Flament. Hypergraphes arborés. *Discrete Math.*, 21(3):223–227, 1978.

[7] Greg N. Frederickson, Matthew S. Hecht, and Chul E. Kim. Approximation algorithms for some routing problems. *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 1976.

[8] N. Guttmann-Beck, R. Hassin, S. Khuller, and B. Raghavachari. Approximation algorithms with bounded performance guarantees for the clustered traveling salesman problem. *Algorithmica*, 28:422–437, 2000.

[9] N. Guttmann-Beck, E. Knaan, and M. Stern. Approximation algorithms for not necessarily disjoint clustered tsp. *Journal of Graph Algorithms and Applications*, 22,4:555–575, 2018.

[10] N. Guttmann-Beck, Z. Sorek, and M. Stern. Clustered spanning tree - conditions for feasibility. *Discrete Mathematics and Theoretical Computer Science*, 2019.

[11] J.A. Hoogeveen. Analysis of christofides' heuristic: Some paths are more difficult than cycles. *Operations Research Letters*, 10:291–295, 1991.

[12] N. Lindzey and R. M. McConnell. Linear-time algorithms for finding tucker submatrices and lekkerkerker-boland subgraphs. *Colorado State University*, 2013.

[13] F.C.J. Lokin. Procedures for travelling salesman problems with additional constraints. *European Journal of Operational Research 3*, pages 135–141, 1978.

[14] T. A. McKee and F. R. McMorris. Topics in intersection graph theory. *SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA*, 1999.

[15] P. J. Slater. A characterization of soft hypergraphs. *Canad. Math. Bull.*, 21(3):335–337, 1978.

[16] R. Swaminathan and D. K. Wagner. On the consecutive-retrieval problem. *SIAM J. Comput.*, 23:398–414, 1994.

[17] A. Tucker. A structure theorem for the consecutive 1's property. *Journal of Combinatorial Theory*, 12(B):153–162, 1972.

Appendices

.1 Extended Booth-Lueker Algorithm Implementation

This implementation is based on the Tyralgo library, and includes a more explicit implementation for the templates of Booth and Lueker Algorithm. The algorithm is used as an Extended Booth-Lueker Algorithm, which finds a feasible removal list if the hypergraph does not have a feasible solution of *FCTSP*.

Note that the algorithm is only a partial implementation of Extended Booth-Lueker Algorithm, used to demonstrate the principles of this algorithm.

Also note that the algorithm includes two options for making changes to the clusters, using methods of vertices removal from clusters and vertices insertion to clusters in the hypergraph, in order to achieve a feasible solution for the new set of clusters.

```
"""\
c.durr, a.durr - 2017-2019


    Solve the consecutive all ones column problem using PQ-trees

    In short, a PQ-tree represents sets of total orders over a ground set. The
    leafs are the values of the ground set. Inner nodes are of type P or Q. P
    means all permutations of the children are allowed. Q means only the left
    to right or the right to left order of the children is allowed.

    The method restrict(S), changes the tree such that it represents only
    total orders which would leave the elements of the set S consecutive.  The
    complexity of restrict is linear in the tree size.

    References:

    [W] https://en.wikipedia.org/wiki/PQ_tree

    [L10] Richard Ladner, slides.
        https://courses.cs.washington.edu/courses/cse421/10au/lectures/PQ.pdf

    [H00] Mohammad Taghi Hajiaghayi, notes.
```

```
        http://www-math.mit.edu/~hajiagha/pp11.ps


    Disclaimer: this implementation does not have the optimal time complexity.
    And also there are more recent and easier algorithms for this problem.

"""

# ==========================================================================
# PQ-Tree Implementation with clusters fix for a Feasible CTSP
# ==========================================================================
# This implementation is based on the tryalgo library, and includes a more explicit implementation for the templates of Booth and Lueker Algorithm.
# The algorithm is used as an Extended Booth and Lueker Algorithm, which finds a feasible removal list if the hypergraph does not have a feasible solution.
# We suggest two fix options in order to gain feasibility:
# 1. FIX_BOTLOC_DELETE = Fixing clusters during the Booth and Lueker Algorithm, by removing vertices from clusters, in order to match one of the templates.
# 2. FIX_BOTLOC_INSERT = Fixing clusters during the Booth and Lueker Algorithm, by inserting vertices to clusters, in order to match one of the templates.
# The selection of a fix method is made by setting variable FIX_TYPE according to the options described above.
# The sections of code which are used to fix the cluster during the Booth and Lueker Algorithm are marked by a comment in the relevant templates.
# The algorithm also contains a DEBUG_MODE, which when activated, is used to print the results of the different steps to the console.
# ==========================================================================
# Disclaimer: The purpose of the algorithm is to demonstrate the possible options for fixing a hypergraph, using a simple implementation.
# Note that the algorithm is only a partial implementation of Extended Booth and Lueker Algorithm, used to demonstrate the principles of the algorithm.
# The code contains implementation for transforming a node to be empty or full, when there is no matching template.
# The algorithm was checked on a limited number of hypergraphs, and therefore may contain bugs in the fixing implementation.
# Also, this implementation is not a complete implementation of the original Booth and Lueker Algorithm, and does not have the optimal time complexity.
# ==========================================================================
```

```python
# This implementation is based on the pq-tree module in the tryalgo library, released under the MIT License.
# The extensions of the algorithm which are described above were implemented by: Hadas Sayag
# Date: 08/06/2020
# ==================================================================

import os
from collections import deque
from copy import deepcopy

P_NODE = 0
Q_NODE = 1
L_NODE = 2

LABEL_EMPTY = 0
LABEL_FULL = 1
LABEL_PARTIAL = 2

FIX_BOTLOC_DELETE = 0
FIX_BOTLOC_INSERT = 1

FIX_TYPE = FIX_BOTLOC_DELETE
DEBUG_MODE = True

class IsNotC1P(Exception):
    """The given instance does not have the all consecutive ones property"""
    pass

class PQ_Node:
    def __init__(self, type, value=None):
        self.parent = None
        self.type = type
        self.label = LABEL_EMPTY
        self.leaf_value = value
        self.full_leafs = 0
        self.processed_sons = 0
        self.sons = []
        self.partial_children = []

    def border(self, L):
        """Append to L the border of the subtree.
        """
        if self.type == L_NODE:
            L.append(self.leaf_value)
        else:
            for x in self.sons:
                x.border(L)
```

```

    def __str__(self):
        if self.type == L_NODE:
            return str(self.leaf_value)
        for x in self.sons:
            assert x.parent == self
        if self.type == P_NODE:
            return "(" + ",".join(map(str, self.sons)) + ")"
        else:
            return "[" + ",".join(map(str, self.sons)) + "]"

class PQ_Tree:
    def __init__(self, allSets):
        self.root = PQ_Node(P_NODE)
        self.all_leafs = []
        for v in allSets:
            node = PQ_Node(L_NODE, value=v)
            node.parent = self.root
            self.root.sons.append(node)
            self.all_leafs.append(node)

    def __str__(self):
        """returns a string representation,
        () for P nodes and [] for Q nodes
        """
        return str(self.root)

    def border(self):
        """returns the list of the leafs in order
        """
        L = []
        self.root.border(L)
        return L

    def reduce(self, set):
        queue = deque(self.all_leafs)

        if DEBUG_MODE: print()
        if DEBUG_MODE: print("set - ", set)
        if DEBUG_MODE: print("tree - ", self.root)
        set_len = len(set)

        while len(queue) > 0:
            x = queue.popleft()

            # x is not root(tree,set)
            if x.full_leafs < set_len:
                isMatch = self.template_L1(x, set)
                if not isMatch:
```

```
                    isMatch = self.template_P1(x, set)
                if not isMatch:
                    isMatch = self.template_P3(x, set)
                if not isMatch:
                    isMatch = self.template_P5(x, set)
                if not isMatch:
                    isMatch = self.template_Fix(x, set, 1, self.template_P5)
                if not isMatch:
                    isMatch = self.template_Q1(x, set)
                if not isMatch:
                    isMatch = self.template_Q2(x, set)
                if not isMatch:
                    isMatch = self.template_Fix(x, set, 1, self.template_Q2)
                if not isMatch:
                    self.cleanTree()
                    raise IsNotC1P

                y = x.parent
                y.full_leafs += x.full_leafs
                y.processed_sons += 1
                if y.processed_sons == len(y.sons):
                    queue.append(y)
            # x is root(tree,set)
            else:
                isMatch = self.template_L1(x, set)
                if not isMatch:
                    isMatch = self.template_P1(x, set)
                if not isMatch:
                    isMatch = self.template_P2(x, set)
                if not isMatch:
                    isMatch = self.template_P4(x, set)
                if not isMatch:
                    isMatch = self.template_P6(x, set)
                if not isMatch:
                    isMatch = self.template_Fix(x, set, 2, self.template_P6)
                if not isMatch:
                    isMatch = self.template_Q1(x, set)
                if not isMatch:
                    isMatch = self.template_Q2(x, set)
                if not isMatch:
                    isMatch = self.template_Q3(x, set)
                if not isMatch:
                    isMatch = self.template_Fix(x, set, 2, self.template_Q3)
                if not isMatch:
                    self.cleanTree()
```

```
                    raise IsNotC1P

        if self.root != None:
            self.cleanTree()

    def cleanTree(self):
        self.root.full_leafs = 0
        self.root.processed_sons = 0
        self.root.label = LABEL_EMPTY
        self.root.partial_children = []
        self.cleanSons(self.root.sons)

    def cleanSons(self, sons):
        newSons = []
        for son in sons:
            son.full_leafs = 0
            son.processed_sons = 0
            son.label = LABEL_EMPTY
            son.partial_children = []
            for grandson in son.sons:
                newSons.append(grandson)
        if len(newSons) > 0:
            self.cleanSons(newSons)

    def template_L1(self, x, set):
        if x.type != L_NODE:
            return False

        if x.leaf_value in set:
            x.label = LABEL_FULL
            x.full_leafs = 1

        if DEBUG_MODE: print("L1 - ", x)
        return True

    def template_P1(self, x, set):
        if x.type != P_NODE:
            return False
        if len(x.partial_children) != 0:
            return False

        if DEBUG_MODE: print("Start P1 - ", x)

        # Check for templates P0 and P1
        isAllEmpty = True
        isAllFull = True
        for son in x.sons:
            if son.label != LABEL_EMPTY:
                isAllEmpty = False
```

```
            if son.label != LABEL_FULL:
                isAllFull = False

        # Update the tree if templates P0 or P1 matches the node
        if isAllFull:
            x.label = LABEL_FULL
        if isAllEmpty or isAllFull:
            if DEBUG_MODE: print("End P1 - ", x)
        return isAllEmpty or isAllFull

    def template_P2(self, x, set):
        if x.type != P_NODE:
            return False
        if len(x.partial_children) != 0:
            return False

        if DEBUG_MODE: print("Start P2 - ", x)

        # Update the tree according to template P2
        full_node = PQ_Node(P_NODE)
        full_node.label = LABEL_FULL

        for son in x.sons:
            if son.label == LABEL_FULL:
                son.parent = full_node
                full_node.sons.append(son)

        for son in full_node.sons:
            x.sons.remove(son)

        if len(full_node.sons) == 1:
            full_node = full_node.sons[0]

        full_node.parent = x
        x.sons.append(full_node)
        if DEBUG_MODE: print("End P2 - ", x)
        return True

    def template_P3(self, x, set):
        if x.type != P_NODE:
            return False
        if len(x.partial_children) != 0:
            return False

        if DEBUG_MODE: print("Start P3 - ", x)

        # Update the tree according to template P3
        full_node = PQ_Node(P_NODE)
        full_node.label = LABEL_FULL
```

```
        empty_node = PQ_Node(P_NODE)
        empty_node.label = LABEL_EMPTY

        for son in x.sons:
            if son.label == LABEL_FULL:
                son.parent = full_node
                full_node.sons.append(son)
            elif son.label == LABEL_EMPTY:
                son.parent = empty_node
                empty_node.sons.append(son)

        for son in full_node.sons:
            x.sons.remove(son)
        for son in empty_node.sons:
            x.sons.remove(son)

        if len(full_node.sons) == 1:
            full_node = full_node.sons[0]
        if len(empty_node.sons) == 1:
            empty_node = empty_node.sons[0]

        full_node.parent = x
        empty_node.parent = x
        x.type = Q_NODE
        x.label = LABEL_PARTIAL
        x.sons.append(empty_node)
        x.sons.append(full_node)
        x.parent.partial_children.append(x)
        if DEBUG_MODE: print("End P3 - ", x)
        return True

    def template_P4(self, x, set):
        if x.type != P_NODE:
            return False
        if len(x.partial_children) != 1:
            return False

        if DEBUG_MODE: print("Start P4 - ", x)

        # Check if the node needs a fix
        part_node = x.partial_children[0]
        isFullSons = False
        for son in x.sons:
            if son.label == LABEL_FULL:
                isFullSons = True
                break
        if isFullSons:
            # Make sure that the partial node is in legal order.
            # Since there are full children, the empty children
```

```
of the partial node should be on one side of it
            if (part_node.sons[0].label == LABEL_EMPTY) and (part_node.sons[len(part_node.sons) - 1].label == LABEL_EMPTY):
                # ====================
                # FIX!
                # ====================
                if FIX_TYPE == FIX_BOTLOC_DELETE:
                    self.makeNodeEmpty(part_node, set)
                    x.partial_children.remove(part_node)
                    isMatch = self.template_P1(x, set)
                    if not isMatch:
                        isMatch = self.template_P2(x, set)
                    return isMatch
                elif FIX_TYPE == FIX_BOTLOC_INSERT:
                    self.makeNodeFull(part_node, set)
                    x.partial_children.remove(part_node)
                    isMatch = self.template_P1(x, set)
                    if not isMatch:
                        isMatch = self.template_P2(x, set)
                    return isMatch
                else:
                    return False
                # ====================

        # Make sure that the partial node is in correct order
        part_node = x.partial_children[0]
        if part_node.sons[0].label == LABEL_FULL:
            part_node.sons.reverse()

        # Update the tree according to template P4
        full_node = PQ_Node(P_NODE)
        full_node.label = LABEL_FULL
        for son in x.sons:
            if son.label == LABEL_FULL:
                son.parent = full_node
                full_node.sons.append(son)
        if len(full_node.sons) > 0:
            for son in full_node.sons:
                x.sons.remove(son)
            if len(full_node.sons) == 1:
                full_node = full_node.sons[0]
            full_node.parent = part_node
            part_node.sons.append(full_node)

        if DEBUG_MODE: print("End P4 - ", x)
        return True

    def template_P5(self, x, set):
```

```
        if x.type != P_NODE:
            return False
        if len(x.partial_children) != 1:
            return False

        if DEBUG_MODE: print("Start P5 - ", x)

        # Check if the node needs a fix
        part_node = x.partial_children[0]
        isFullSons = False
        for son in x.sons:
            if son.label == LABEL_FULL:
                isFullSons = True
                break
        if isFullSons:
            # Make sure that the partial node is in legal order.
            # Since there are full children, the empty children of the partial node should be on one side of it
            if (part_node.sons[0].label == LABEL_EMPTY) and (part_node.sons[len(part_node.sons) - 1].label == LABEL_EMPTY):
                # ====================
                # FIX!
                # ====================
                if FIX_TYPE == FIX_BOTLOC_DELETE:
                    self.makeNodeEmpty(part_node, set)
                    x.partial_children.remove(part_node)
                    isMatch = self.template_P1(x, set)
                    if not isMatch:
                        isMatch = self.template_P3(x, set)
                    return isMatch
                elif FIX_TYPE == FIX_BOTLOC_INSERT:
                    self.makeNodeFull(part_node, set)
                    x.partial_children.remove(part_node)
                    isMatch = self.template_P1(x, set)
                    if not isMatch:
                        isMatch = self.template_P3(x, set)
                    return isMatch
                else:
                    return False
                # ====================

        full_node = PQ_Node(P_NODE)
        full_node.label = LABEL_FULL
        empty_node = PQ_Node(P_NODE)
        empty_node.label = LABEL_EMPTY

        # Make sure that the partial node is in correct order
        if part_node.sons[0].label == LABEL_FULL:
```

```
            part_node.sons.reverse()

        # Update the tree according to template P5
        for son in x.sons:
            if son.label == LABEL_FULL:
                son.parent = full_node
                full_node.sons.append(son)
            elif son.label == LABEL_EMPTY:
                son.parent = empty_node
                empty_node.sons.append(son)

        if len(empty_node.sons) > 0:
            for son in empty_node.sons:
                x.sons.remove(son)
            if len(empty_node.sons) == 1:
                empty_node = empty_node.sons[0]
            empty_node.parent = x
            x.sons.insert(0,empty_node)

        for son in part_node.sons:
            son.parent = x
            x.sons.append(son)

        if len(full_node.sons) > 0:
            for son in full_node.sons:
                x.sons.remove(son)
            if len(full_node.sons) == 1:
                full_node = full_node.sons[0]
            full_node.parent = x
            x.sons.append(full_node)

        x.type = Q_NODE
        x.label = LABEL_PARTIAL
        x.sons.remove(part_node)
        x.partial_children.remove(part_node)
        x.parent.partial_children.append(x)
        if DEBUG_MODE: print("End P5 - ", x)
        return True

    def template_P6(self, x, set):
        if x.type != P_NODE:
            return False
        if len(x.partial_children) != 2:
            return False

        if DEBUG_MODE: print("Start P6 - ", x)

        # Make sure that the partial nodes are in legal order.
        # In both partial nodes, the empty children should be on
```

```
one side of it
        first_part_node = x.partial_children[0]
        second_part_node = x.partial_children[1]
        if (((first_part_node.sons[0].label == LABEL_EMPTY) and
              (first_part_node.sons[len(first_part_node.sons) -
1].label ==
               LABEL_EMPTY)) or
             ((second_part_node.sons[0].label == LABEL_EMPTY) and
              (second_part_node.sons[len(second_part_node.sons) -
1].label ==
               LABEL_EMPTY))):
            # ====================
            # FIX!
            # ====================
            if FIX_TYPE == FIX_BOTLOC_DELETE:
                self.makeNodeEmpty(first_part_node, set)
                self.makeNodeEmpty(second_part_node, set)
                x.partial_children.remove(first_part_node)
                x.partial_children.remove(second_part_node)
                isMatch = self.template_P1(x, set)
                if not isMatch:
                    isMatch = self.template_P2(x, set)
                return isMatch
            elif FIX_TYPE == FIX_BOTLOC_INSERT:
                self.makeNodeFull(first_part_node, set)
                self.makeNodeFull(second_part_node, set)
                x.partial_children.remove(first_part_node)
                x.partial_children.remove(second_part_node)
                isMatch = self.template_P1(x, set)
                if not isMatch:
                    isMatch = self.template_P2(x, set)
                return isMatch
            else:
                return False
            # ====================

        # Make sure that the partial nodes are in correct order
        if first_part_node.sons[0].label == LABEL_FULL:
            first_part_node.sons.reverse()
        if second_part_node.sons[0].label == LABEL_EMPTY:
            second_part_node.sons.reverse()

        # Update the tree according to template P6
        full_node = PQ_Node(P_NODE)
        full_node.label = LABEL_FULL

        for son in x.sons:
            if son.label == LABEL_FULL:
                son.parent = full_node
```

```
                full_node.sons.append(son)

        for son in full_node.sons:
            x.sons.remove(son)

        if len(full_node.sons) == 1:
            full_node = full_node.sons[0]

        full_node.parent = first_part_node
        first_part_node.sons.append(full_node)

        for i in range(len(second_part_node),0,-1):
            second_part_node[i].parent = first_part_node
            first_part_node.append(second_part_node[i])

        x.sons.remove(second_part_node)
        if DEBUG_MODE: print("End P6 - ", x)
        return True

    def template_Q1(self, x, set):
        if x.type != Q_NODE:
            return False
        if len(x.partial_children) != 0:
            return False

        if DEBUG_MODE: print("Start Q1 - ", x)

        # Check for templates Q0 and Q1
        isAllEmpty = True
        isAllFull = True
        isLegalPartial = True
        for son in x.sons:
            if son.label != LABEL_EMPTY:
                isAllEmpty = False
            if son.label != LABEL_FULL:
                isAllFull = False

        # Update the tree if templates Q0 or Q1 matches the node
        if isAllFull:
            x.label = LABEL_FULL
            isLegalPartial = False
        elif isAllEmpty:
            isLegalPartial = False
        # Check if there is a legal order of the full sequence
        else:
            isLegalPartial = self.checkLegalQNodeFullSequence(x)
            if isLegalPartial:
                x.label = LABEL_PARTIAL
                x.parent.partial_children.append(x)
```

```
            else:
                # ====================
                # FIX!
                # ====================
                if FIX_TYPE == FIX_BOTLOC_DELETE:
                    self.makeNodeEmpty(x, set)
                    x.label = LABEL_EMPTY
                    return True
                elif FIX_TYPE == FIX_BOTLOC_INSERT:
                    self.makeNodeFull(x, set)
                    x.label = LABEL_FULL
                    return True
                else:
                    return False
                # ====================

        if DEBUG_MODE: print("End Q1 - ", x)
        return isAllEmpty or isAllFull or isLegalPartial

    def template_Q2(self, x, set):
        if x.type != Q_NODE:
            return False
        if len(x.partial_children) != 1:
            return False

        if DEBUG_MODE: print("Start Q2 - ", x)

        # Check if there is a legal order of the full sequence
        isNeedFix = False
        isLegalPartial = self.checkLegalQNodeFullSequence(x)
        if not isLegalPartial:
            isNeedFix = True

        [first_full_index, last_full_index, first_partial_index, second_partial_index] = self.checkFullPartialLocations(x)
        part_node = x.partial_children[0]

        # The partial node must be next to the full sequence (if there is no full sequence, any location of the partial is legal)
        if first_full_index != None:
            # Make sure that the partial node is in legal order
            if (part_node.sons[0].label == LABEL_EMPTY) and (part_node.sons[len(part_node.sons) - 1].label == LABEL_EMPTY):
                isNeedFix = True
            # If the partial is before the full sequence, the order of the partial should be empty and then full
```

```
            if first_partial_index == first_full_index - 1:
                if part_node.sons[0].label == LABEL_FULL:
                    part_node.sons.reverse()
            # If the partial is after the full sequence, the order of the partial should be full and then empty
            elif first_partial_index == last_full_index + 1:
                if part_node.sons[0].label == LABEL_EMPTY:
                    part_node.sons.reverse()
            # The partial place is not legal
            else:
                isNeedFix = True

        # ==========================
        # FIX!
        # ==========================
        if isNeedFix:
            if FIX_TYPE == FIX_BOTLOC_DELETE:
                self.makeNodeEmpty(part_node, set)
                x.partial_children.remove(part_node)
                return self.template_Q1(x, set)
            elif FIX_TYPE == FIX_BOTLOC_INSERT:
                self.makeNodeFull(part_node, set)
                x.partial_children.remove(part_node)
                return self.template_Q1(x, set)
            else:
                return False
        # ==========================

        # Update the tree according to template Q2
        x.partial_children.remove(part_node)
        x.sons.remove(part_node)
        for son in part_node.sons:
            son.parent = x
            x.sons.insert(first_partial_index, son)
            first_partial_index += 1

        if DEBUG_MODE: print("End Q2 - ", x)
        return True

    def template_Q3(self, x, set):
        if x.type != Q_NODE:
            return False
        if len(x.partial_children) != 2:
            return False

        if DEBUG_MODE: print("Start Q3 - ", x)

        # Check if there is a legal order of the full sequence
        isNeedFix = False
```

```python
        isLegalPartial = self.checkLegalQNodeFullSequence(x)
        if not isLegalPartial:
            isNeedFix = True

        [first_full_index, last_full_index, first_partial_index,
            second_partial_index] = self.checkFullPartialLocations(x)
        first_part_node = x.partial_children[0]
        second_part_node = x.partial_children[1]

        # Make sure that the partial nodes is in legal order
        if (first_part_node.sons[0].label == LABEL_EMPTY) and (first_part_node.sons[len(first_part_node.sons) - 1].label ==
            LABEL_EMPTY):
            isNeedFix = True
        if (second_part_node.sons[0].label == LABEL_EMPTY) and (second_part_node.sons[len(second_part_node.sons) - 1].label ==
            LABEL_EMPTY):
            isNeedFix = True

        # The partial nodes must be on both sides of the full sequence, or next to each other if there is no full
        if first_full_index != None:
            if (first_partial_index != first_full_index - 1) or (second_partial_index != last_full_index + 1):
                isNeedFix = True
        else:
            if second_partial_index != first_partial_index + 1:
                isNeedFix = True

        # ==================
        # FIX!
        # ==================
        if isNeedFix:
            if FIX_TYPE == FIX_BOTLOC_DELETE:
                self.makeNodeEmpty(first_part_node, set)
                x.partial_children.remove(first_part_node)
                self.makeNodeEmpty(second_part_node, set)
                x.partial_children.remove(second_part_node)
                return self.template_Q1(x, set)
            elif FIX_TYPE == FIX_BOTLOC_INSERT:
                self.makeNodeFull(first_part_node, set)
                x.partial_children.remove(first_part_node)
                self.makeNodeFull(second_part_node, set)
                x.partial_children.remove(second_part_node)
                return self.template_Q1(x, set)
            else:
                return False
```

```python
        # ====================

        # If the partial is first or before the full sequence, the order of the partial should be empty and then full.
        # If the partial is second or after the full sequence, the order of the partial should be full and then empty
        if first_part_node.sons[0].label == LABEL_FULL:
            first_part_node.sons.reverse()
        if second_part_node.sons[0].label == LABEL_EMPTY:
            second_part_node.sons.reverse()

        # Update the tree according to template Q3
        x.partial_children.remove(first_part_node)
        x.partial_children.remove(second_part_node)
        x.sons.remove(first_part_node)
        x.sons.remove(second_part_node)

        for son in first_part_node.sons:
            son.parent = x
            x.sons.insert(first_partial_index, son)
            first_partial_index += 1
        second_partial_index += len(first_part_node.sons)
        for son in second_part_node.sons:
            son.parent = x
            x.sons.insert(second_partial_index, son)
            second_partial_index += 1

        if DEBUG_MODE: print("End Q3 - ", x)
        return True

    def template_Fix(self, x, set, start_range, templateToRun):

        # ====================
        # FIX!
        # ====================
        if FIX_TYPE == FIX_BOTLOC_DELETE:
            for i in range(start_range,len(x.partial_children)):
                self.makeNodeEmpty(x.partial_children[i], set)
                x.partial_children.remove(x.partial_children[i])
            return templateToRun(x, set)
        elif FIX_TYPE == FIX_BOTLOC_INSERT:
            for i in range(start_range,len(x.partial_children)):
                self.makeNodeFull(x.partial_children[i], set)
                x.partial_children.remove(x.partial_children[i])
            return templateToRun(x, set)
        else:
            return False
        # ====================
```

```python
    # Check if the sequence of the Q_Node is legal.
    # All the full should appear consecutively
    def checkLegalQNodeFullSequence(self, x):
        isLegalPartial = True
        isFullStarted = False
        isFullFinished = False
        for son in x.sons:
            if son.label == LABEL_FULL:
                isFullStarted = True
                if isFullFinished:
                    isLegalPartial = False
                    break
            else:
                if isFullStarted:
                    isFullFinished = True
        return isLegalPartial

    def makeNodeEmpty(self, node, set):
        if DEBUG_MODE: print(
                "Empty Fix: set - ", set, ", leaf - ", node.leaf_value)
        node.label = LABEL_EMPTY
        if node.type == L_NODE:
            if node.leaf_value in set:
                set.remove(node.leaf_value)
        for son in node.sons:
            self.makeNodeEmpty(son, set)

    def makeNodeFull(self, node, set):
        if DEBUG_MODE: print(
                "Full Fix: set - ", set, ", leaf - ", node.leaf_value)
        node.label = LABEL_FULL
        if node.type == L_NODE:
            if not (node.leaf_value in set):
                set.add(node.leaf_value)
        for son in node.sons:
            self.makeNodeFull(son, set)

    # Check for start and end of the full sequence in x, and the location of the partial node
    def checkFullPartialLocations(self, x):
        first_full_index = None
        last_full_index = None
        first_partial_index = None
        second_partial_index = None
        for i in range(len(x.sons)):
            if x.sons[i].label == LABEL_FULL:
                if first_full_index == None:
```

```python
                    first_full_index = i
            elif x.sons[i].label == LABEL_EMPTY:
                if first_full_index != None:
                    last_full_index = i - 1
            elif x.sons[i].label == LABEL_PARTIAL:
                if first_partial_index == None:
                    first_partial_index = i
                else:
                    second_partial_index = i
                if first_full_index != None:
                    last_full_index = i - 1
        if last_full_index == None:
            if first_full_index != None:
                last_full_index = len(x.sons) - 1

        return [first_full_index, last_full_index, first_partial_index,
                second_partial_index]

def consecutive_ones_property(sets, universe=None):
    """ Check the consecutive ones property.

    :param list sets: is a list of subsets of the ground set.
    :param groundset: is the set of all elements,
                by default it is the union of the given sets
    :returns: returns a list of the ordered ground set where
              every given set is consecutive,
              or None if there is no solution.
    :complexity: O(len(groundset) * len(sets))
    :disclaimer: an optimal implementation would have complexity
                 O(len(groundset) + len(sets) + sum(map(len,sets))),
                 and there are more recent easier algorithms for this problem.
    """
    if universe is None:
        universe = set()
        for S in sets:
            universe |= set(S)
    tree = PQ_Tree(universe)

    tempSets = deepcopy(sets)
    for i in range(len(tempSets)):
        try:
            tree.reduce(tempSets[i])
        except IsNotC1P:
            print("Error - No Feasible Solution!")

    print()
```

```python
    print("tree = ", tree, "\t() = P node, [] = Q node")
    print("Original clusters = ", sets)
    print("New clusters = ", tempSets)
    return tree.border()

# ==============================================================
# Test Implementation - My Test Cases
# ==============================================================

os.system('cls')

def runTest(testSet):
    print()
    print("=" * 70)
    print("Clusters = ", testSet)
    print("=" * 70)
    treeBorder = consecutive_ones_property(testSet)
    print("Path = ", treeBorder)

# Test 1.1
testSet = [{1,2,3}, {2,3,4}]
runTest(testSet)

# Test 1.2
#testSet = [{1,2,3,6}, {3,4,5,6}, {2,3,4}]
#runTest(testSet)

# Test 1.3
#testSet = [{1,2,3,6}, {3,4,5,6}, {3,4}]
#runTest(testSet)

# Test 2
#testSet = [{1,2,3}, {3,4,5}, {2,3,4}]
#runTest(testSet)

# Test 3
#testSet = [{1,2,3,6}, {3,4,5,6}, {2,3,4}]
#runTest(testSet)

# Test 4
#testSet = [{1,2,3,4,5}, {4,5,6,7}, {2,3,6}]
#runTest(testSet)

# Test 5
#testSet = [{1,2,3,4,5,6}, {5,6,7,8,9,10,11}, {1,9,10,11,12}]
#testSet = [{1,2,3,4,5,6}, {1,9,10,11,12}, {5,6,7,8,9,10,11}]
#testSet = [{1,9,10,11,12}, {5,6,7,8,9,10,11}, {1,2,3,4,5,6}]
#runTest(testSet)
```

```
# Test 6
#testSet = [{1,2}, {1,3}, {1,4}, {1,5}]
#runTest(testSet)
```

Listing 1: Extended Booth-Lueker Algorithm Implementation

.2 Booth-Lueker Algorithm for a Known End of Path Implementation

This implementation is based on the Tyralgo library, and extends it to find a feasible solution of *FCTSP* with a known cluster at an end of it, if such a solution exists. The algorithm initializes a PQ-Tree with two P-Nodes. One P-Node contains the vertices of a known endpoint, and the other P-Node contains the rest of the vertices in the hypergraph. The reduce step is unchanged, using the implementation in the Tryalgo library, but it is activated on the updated PQ-Tree and only on the clusters which are not the known endpoint.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""\
c.durr, a.durr - 2017-2019


    Solve the consecutive all ones column problem using PQ-trees

    In short, a PQ-tree represents sets of total orders over a
    ground set. The
    leafs are the values of the ground set. Inner nodes are of
    type P or Q. P
    means all permutations of the children are allowed. Q means
    only the left
    to right or the right to left order of the children is
    allowed.

    The method restrict(S), changes the tree such that it
    represents only
    total orders which would leave the elements of the set S
    consecutive.  The
    complexity of restrict is linear in the tree size.

    References:

    [W] https://en.wikipedia.org/wiki/PQ_tree
```

```

    [L10] Richard Ladner, slides.
        https://courses.cs.washington.edu/courses/cse421/10au/lectures/PQ.pdf

    [H00] Mohammad Taghi Hajiaghayi, notes.
        http://www-math.mit.edu/~hajiagha/pp11.ps


    Disclaimer: this implementation does not have the optimal time complexity.
    And also there are more recent and easier algorithms for this problem.

"""

# ============================================================================
# PQ-Tree Implementation for a Known Endpoint
# ============================================================================
# This implementation is based on the tryalgo library, and extends it to find a solution with a known cluster at an end of the solution, if such a solution exists.
# The algorithm initializes a PQ-Tree with two P-Nodes. One P-Node contains the vertices of a known endpoint, and the other P-Node contains the rest of the vertices in the hypergraph.
# The reduce step is unchanged, using the implementation in the tryalgo library, but it is activated on the updated PQ-Tree and only on the clusters which are not the known endpoint.
# ============================================================================
# This implementation is based on the pq-tree module in the tryalgo library, released under the MIT License.
# The extensions of the algorithm which are described above were implemented by: Hadas Sayag
# Date: 08/06/2020
# ============================================================================

# pylint: disable=bad-whitespace, missing-docstring, len-as-condition
# pylint: disable=too-many-nested-blocks, no-else-raise, too-many-branches

import os
from collections import deque
import itertools

# pylint: disable=unnecessary-pass
class IsNotC1P(Exception):
    """The given instance does not have the all consecutive ones property"""
```

```
    pass

P_shape = 0
Q_shape = 1
L_shape = 2

EMPTY = 0
FULL = 1
PARTIAL = 2

# automaton is used for pattern recognition when reducing a Q
    node
# -1 represents the result of a forbidden transition
#               E  F  P
automaton = [[1,  5,  4],  # 0 initial state
             [1,  2,  2],  # 1
             [3,  2,  3],  # 2
             [3, -1, -1],  # 3
             [6,  2,  3],  # 4
             [6,  5,  6],  # 5
             [6, -1, -1]]  # 6


class PQ_node:

    def __init__(self, shape, value=None):
        self.shape = shape
        self.sons = []
        self.parent = None
        self.value = value
        self.full_leafs = 0
        self.processed_sons = 0
        self.mark = EMPTY

    def add(self, node):
        """Add one node as descendant
        """
        self.sons.append(node)
        node.parent = self

    def add_all(self, L):
        for x in L:
            self.add(x)

    def add_group(self, L):
        """Add elements of L as descendants of the node.
        If there are several elements in L, group them in a P-
    node first
        """
```

```
        if len(L) == 1:
            self.add(L[0])
        elif len(L) >= 2:
            x = PQ_node(P_shape)
            x.add_all(L)
            self.add(x)
        # nothing to be done for an empty list L

    def border(self, L):
        """Append to L the border of the subtree.
        """
        if self.shape == L_shape:
            L.append(self.value)
        else:
            for x in self.sons:
                x.border(L)

# pylint: disable=no-else-return
    def __str__(self):
        if self.shape == L_shape:
            return str(self.value)
        for x in self.sons:
            assert x.parent == self
        if self.shape == P_shape:
            return "(" + ",".join(map(str, self.sons)) + ")"
        else:
            return "[" + ",".join(map(str, self.sons)) + "]"

# pylint: disable=too-many-statements
class PQ_tree:

    def __init__(self, endpoint_cluster, universe):
        self.tree = PQ_node(P_shape)
        self.leafs = []

        # Initialize a PQ-Tree with an appropriate structure to find a feasible solution with a known endpoint
        endpoint_pnode = PQ_node(P_shape)
        self.tree.add(endpoint_pnode)
        for i in endpoint_cluster:
            x = PQ_node(L_shape, value=i)
            endpoint_pnode.add(x)
            self.leafs.append(x)

        universe_pnode = PQ_node(P_shape)
        self.tree.add(universe_pnode)
        for i in universe:
            x = PQ_node(L_shape, value=i)
            universe_pnode.add(x)
```

```
            self.leafs.append(x)

    def __str__(self):
        """returns a string representation,
        () for P nodes and [] for Q nodes
        """
        return str(self.tree)

    def border(self):
        """returns the list of the leafs in order
        """
        L = []
        self.tree.border(L)
        return L

    def reduce(self, S):
        queue = deque(self.leafs)
        cleanup = []                         # we don't need to cleanup leafs
        is_key_node = False
        # while there are nodes to be processed
        while queue and not is_key_node:
            x = queue.popleft()
            is_key_node = (x.full_leafs == len(S))
            x.mark = PARTIAL                 # default mark
            if x.shape == P_shape:
                # group descendants according to marks
                E = []
                F = []
                P = []
                for y in x.sons:
                    if y.mark == EMPTY:
                        E.append(y)
                    elif y.mark == FULL:
                        F.append(y)
                    else:
                        P.append(y)
                if len(P) == 0:        # start long case analysis
                    if len(E) == 0:
                        x.mark = FULL
                    else:
                        if len(F) == 0:
                            # template P1
                            x.mark = EMPTY
                        else:
                            if is_key_node:
                                # template P2
                                x.sons = E
                                x.add_group(F)
```

```
                            else:                       # is not root
                                # template P3
                                x.shape = Q_shape
                                x.sons = []
                                x.add_group(E)
                                x.add_group(F)
                    elif len(P) == 1:
                        assert P[0].shape == Q_shape
                        if is_key_node:
                            # template P4
                            x.sons = E + [P[0]]
                            P[0].add_group(F)
                        else:                           # is not root
                            # template P5
                            x.shape = Q_shape
                            x.sons = []
                            x.add_group(E)
                            x.add_all(P[0].sons)
                            x.add_group(F)
                    elif len(P) == 2:
                        if is_key_node:
                            # template P6
                            x.sons = E
                            z = P[0]
                            z.add_group(F)
                            z.add_all(reversed(P[1].sons))
                            # POSSIBLE BUG IN TRY_ALGO - z was not added as a son to the root of the tree.
                            # This line was added to solve this bug
                            x.add(z)
                        else:
                            raise IsNotC1P
                    else:                               # more than 2 partial descendants
                        raise IsNotC1P
                elif x.shape == Q_shape:
                    state = 0
                    L = []
                    for y in x.sons:
                        previous = state
                        state = automaton[state][y.mark]
                        if state == -1:
                            raise IsNotC1P
                        elif (state in (3, 6)) and y.mark == PARTIAL:
                            assert y.shape == Q_shape
                            L += reversed(y.sons)
                        elif state == 6 and previous == 4:
                            L = L[::-1]
```

```python
                        L.append(y)
                    elif y.mark == PARTIAL:
                        L += y.sons
                    else:
                        L.append(y)
                if state == 3 and not is_key_node:
                    raise IsNotC1P
                elif state == 5:
                    x.mark = FULL
                x.sons = []
                if state == 6:
                    x.add_all(reversed(L))
                else:
                    x.add_all(L)
            else:                              # x is a leaf
                if x.value in S:
                    x.mark = FULL
                    x.full_leafs = 1
                else:
                    x.mark = EMPTY
                    x.full_leafs = 0
            # propagate node processing
            if not is_key_node:
                z = x.parent
                # cumulate bottom up full leaf numbers
                z.full_leafs += x.full_leafs
                if z.processed_sons == 0:
                    # first time considered
                    cleanup.append(z)
                z.processed_sons += 1
                if z.processed_sons == len(z.sons):
                    # otherwise prune tree at z
                    queue.append(z)
        for x in cleanup:
            x.full_leafs = 0
            x.processed_sons = 0
            x.mark = EMPTY

def consecutive_ones_property(endpoint_cluster, other_sets, universe=None):
    """ Check the consecutive ones property.

    :param list sets: is a list of subsets of the ground set.
    :param groundset: is the set of all elements,
                by default it is the union of the given sets
    :returns: returns a list of the ordered ground set where
              every given set is consecutive,
              or None if there is no solution.
    :complexity: O(len(groundset) * len(sets))
```

```
    :disclaimer: an optimal implementation would have complexity
                  O(len(groundset) + len(sets) + sum(map(len,sets))),
                  and there are more recent easier algorithms for this problem.
    """
    if universe is None:
        universe = set()
        for S in other_sets:
            for leaf in S:
                if leaf not in endpoint_cluster:
                    universe.add(leaf)
    tree = PQ_tree(endpoint_cluster, universe)

    print("Original Tree = ", tree, "\t() = P node, [] = Q node")

    isFeasibleSolution = True

    # Process all the clusters except for the known endpoint
    for i in range(len(other_sets)):
        try:
            tree.reduce(other_sets[i])
        # If there is an exception there is no feasible solution
        except:
            isFeasibleSolution = False
            break

    if (isFeasibleSolution):
        print("Tree after Reduce = ", tree, "\t() = P node, [] = Q node")
        return tree.border()
    else:
        print("There is no feasible solution!")
        return None

# ==============================================
# Test Implementation - My Test Cases
# ==============================================

def runTest(endpointCluster, testSetOtherClusters):
    print()
    print("=" * 70)
    print("Endpoint = ", endpointCluster)
    print("Other Clusters = ", testSetOtherClusters)
    print("=" * 70)
    treeBorder = consecutive_ones_property(
        endpointCluster, testSetOtherClusters)
    print("Path = ", treeBorder)

```

```
os.system('cls')

# Test 1.1
testSet = [{1,2,3}, {3,4,5}, {5,6,7}]
testSetOtherClusters = [{3,4,5}, {5,6,7}]
endpointCluster = {1,2,3}
runTest(endpointCluster, testSetOtherClusters)

# Test 2.1
#testSet = [{1,2,3,4}, {3,4,5,6,7,8}, {7,8,9,10,11,12},
    {11,12,13,14}]
#testSetOtherClusters = [{3,4,5,6,7,8}, {7,8,9,10,11,12},
    {11,12,13,14}]
#endpointCluster = {1,2,3,4}
#runTest(endpointCluster, testSetOtherClusters)

# Test 2.2
#testSet = [{1,2,3,4}, {3,4,5,6,7,8}, {7,8,9,10,11,12},
    {11,12,13,14}]
#testSetOtherClusters = [{1,2,3,4}, {7,8,9,10,11,12},
    {11,12,13,14}]
#endpointCluster = {3,4,5,6,7,8}
#runTest(endpointCluster, testSetOtherClusters)

# Test 3.1
#testSet = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3}, {3,4,5,6,7,8,9},
    {9,10,11}]
#testSetOtherClusters = [{1,2,3,4}, {1,2,3}, {3,4,5,6,7,8,9},
    {9,10,11}]
#endpointCluster = {1,2,3,4,5,6}
#runTest(endpointCluster, testSetOtherClusters)

# Test 3.2
#testSet = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3}, {3,4,5,6,7,8,9},
    {9,10,11}]
#testSetOtherClusters = [{1,2,3,4,5,6}, {1,2,3}, {3,4,5,6,7,8,9},
     {9,10,11}]
#endpointCluster = {1,2,3,4}
#runTest(endpointCluster, testSetOtherClusters)

# Test 3.3
#testSet = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3}, {3,4,5,6,7,8,9},
    {9,10,11}]
#testSetOtherClusters = [{1,2,3,4,5,6}, {1,2,3,4},
    {3,4,5,6,7,8,9}, {9,10,11}]
#endpointCluster = {1,2,3}
#runTest(endpointCluster, testSetOtherClusters)

# Test 3.4
```

```
#testSet = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3}, {3,4,5,6,7,8,9},
    {9,10,11}]
#testSetOtherClusters = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3},
    {9,10,11}]
#endpointCluster = {3,4,5,6,7,8,9}
#runTest(endpointCluster, testSetOtherClusters)

# Test 3.5
#testSet = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3}, {3,4,5,6,7,8,9},
    {9,10,11}]
#testSetOtherClusters = [{1,2,3,4,5,6}, {1,2,3,4}, {1,2,3},
    {3,4,5,6,7,8,9}]
#endpointCluster = {9,10,11}
#runTest(endpointCluster, testSetOtherClusters)
```

Listing 2: Booth-Lueker Algorithm for a Known End of Path Implementation