Fundamental Computational Problems and Algorithms for SuperHyperGraphs, pp. 240-272, in Takaaki Fujita, Florentin Smarandache: *Advancing Uncertain Combinatorics through Graphization, Hyperization, and Uncertainization: Fuzzy, Neutrosophic, Soft, Rough, and Beyond. Second volume.* Gallup, NM, United States of America – Guayaquil (Ecuador): NSIA Publishing House, 2024, 518 p.

Fundamental Computational Problems and Algorithms for SuperHyperGraphs

Takaaki Fujita [1] *, *Florentin Smarandache*[2]

[1] Independent Researcher, Shinjuku, Shinjuku-ku, Tokyo, Japan.

[2] University of New Mexico, Gallup Campus, NM 87301, USA.
Corresponding Emails: t171d603@gunma-u.ac.jp

Abstract

Hypergraphs extend traditional graphs by allowing edges (known as hyperedges) to connect more than two vertices, rather than just pairs. This paper explores fundamental problems and algorithms in the context of SuperHypergraphs, an advanced extension of hypergraphs enabling modeling of hierarchical and complex relationships. Topics covered include constructing SuperHyperGraphs, recognizing SuperHyperTrees, and computing SuperHyperTree-width. We address a range of optimization problems, such as the SuperHy-pergraph Partition Problem, Reachability, Minimum Spanning SuperHypertree, and Single-Source Shortest Path. Furthermore, adaptations of classical problems like the Traveling Salesman Problem, Chinese Postman Problem, and Longest Simple Path Problem are presented in the SuperHypergraph framework.

Keywords: Superhypergraph, Hypergraph, Tree-width, Algorithm

MSC 2010 classifications: 05C65 - Hypergraphs, 68R10 - Graph theory in computer science

1 Introduction

1.1 Graphs and Hypergraphs

Graph theory serves as a foundational framework for analyzing networks, consisting of nodes (vertices) and their connections (edges). It provides valuable insights into the structure, connectivity, and properties of diverse networks [28].

Hypergraphs extend traditional graphs by allowing edges (known as hyperedges) to connect more than two vertices, rather than just pairs. These generalized structures have gained significant attention due to their broad applications in graph theory, computer science, and related fields [6, 21, 32, 49, 58, 116]. Further extending hypergraphs, SuperHypergraphs introduce even greater flexibility and are a topic of emerging interest in recent studies [105,106]. A SuperHypergraph generalizes hypergraphs, allowing vertices and hyperedges to represent sets or subsets, enabling modeling of hierarchical and complex relationships.

In the context of graphs, hypergraphs, and superhypergraphs, tree structures have been extensively studied. For instance, Hypertrees have been explored in hypergraphs [54], while SuperHypertrees have been investigated in superhypergraphs [51]. The concept of tree structures is widely adopted due to their simplicity and efficiency in applications. Beyond graph-related concepts, tree structures have been applied in various fields, including Tree Automata [17,23], TreeSoft sets [9,41,109], and Decision Trees [19,20], contributing to ongoing research and advancements in these areas.

1.2 Graph Width Parameters

Graph characteristics are often analyzed using various parameters, with significant research devoted to understanding these measures. Among these, graph width parameters such as tree-width [14–16, 90, 91, 96] are particularly prominent. These parameters evaluate how closely a graph approximates a tree structure, which is essential for many practical applications.

For hypergraphs, analogous parameters like Hypertree-width [3,53,55,75,121] and Hyperpath-width [2,78,83] have been developed. These metrics measure the resemblance of a hypergraph to a tree or a path, addressing the need to extend tree-based analyses to more complex structures.

1.3 Computational Complexity

An algorithm is a finite sequence of well-defined instructions intended to solve a specific problem or perform a computation [31]. The time and space complexities of the algorithm are often subjects of analysis. Computational complexity evaluates the resources, such as time and space, required by an algorithm to solve a problem as a function of input size, offering theoretical efficiency bounds [1, 10, 61, 85].

Algorithms in graph theory are referred to as graph algorithms [31]. For graphs and hypergraphs, numerous problems and algorithms have been studied, with research also exploring real-world applications (e.g. [46, 64, 82]).

1.4 Contributions of This Paper

Research on problems and algorithms related to SuperHypergraphs remains limited. Therefore, this paper investigates various fundamental problems in SuperHypergraphs and proposes algorithms to address them.

- *Exact Construction of SuperHyperGraph*: Develops an algorithm to construct a SuperHyperGraph from a given vertex and edge set.
- *Recognizing a SuperHyperTree*: Provides an algorithm to determine if a given structure is a valid SuperHyperTree.
- *Computation of SuperHyperTree-width*:
  - *Exact Algorithm*: Defines an exact algorithm to compute the SuperHyperTree-width of a superhypergraph.
  - *Approximation Algorithm*: Proposes an approximation algorithm for computationally efficient SuperHyperTree-width calculation.
- *Superhypergraph Partition Problem*: Studies methods to partition a SuperHyperGraph into disjoint subsets while minimizing inter-partition connections.
- *Reachability Problem in SuperHypergraphs*: Explores algorithms to verify if a path exists between two vertices in a SuperHyperGraph.
- *Minimum Spanning SuperHypertree Problem*: Presents a method to compute a SuperHypertree with the minimum total edge weight.
- *Single-Source Shortest Path Problem in a SuperHypergraph*: Develops an algorithm to find shortest paths from a single source vertex in a SuperHyperGraph.
- *Traveling Salesman Problem in a SuperHypergraph*: Examines the adaptation of TSP to SuperHyperGraphs, finding the minimum tour covering all vertices.
- *Chinese Postman Problem (CPP) in SuperHypergraphs*: Analyzes how to find an Eulerian circuit in SuperHyperGraphs under given conditions.
- *Longest Simple Path Problem in SuperHypergraphs*: Studies methods to identify the longest acyclic path in a SuperHyperGraph.
- *Maximum Spanning Tree Problem in SuperHypergraphs*: Proposes algorithms to compute a SuperHypertree with the maximum total edge weight.
- *Horn Satisfiability Problem in a SuperHypergraph*: Extends the Horn Satisfiability Problem to SuperHyperGraphs with adapted algorithms and proofs.

1.5 The Structure of the Paper

The structure of this paper is as follows.

2 Preliminaries and Definitions

In this section, we provide the preliminaries and definitions. Readers seeking foundational concepts and notations in graph theory are encouraged to consult standard texts, surveys, or lecture notes, such as [26–28,119]. This work also utilizes basic principles from set theory, for which references like [35, 62, 66, 67, 72] are recommended. For detailed discussions on specific operations and related topics addressed in this paper, readers may refer to the respective references for additional insights.

2.1 Graphs and Hypergraphs

Graph theory provides a fundamental framework for analyzing networks, which are composed of nodes (vertices) and their connections (edges). A hypergraph extends the traditional graph concept by allowing hyperedges, which can connect multiple vertices rather than just pairs, enabling the representation of more complex relationships between elements [11–13, 54–56]. The basic definitions of graphs and hypergraphs are presented below.

Definition 2.1 (Graph)**.** [28] A graph G is a mathematical structure consisting of a set of vertices $V(G)$ and a set of edges $E(G)$ that connect pairs of vertices, representing relationships or connections between them. Formally, a graph is defined as $G = (V, E)$, where V is the vertex set and E is the edge set.

Definition 2.2 (Subgraph)**.** [28] Let $G = (V, E)$ be a graph. A *subgraph* $H = (V_H, E_H)$ of G is a graph such that:

- $V_H \subseteq V$, i.e., the vertex set of H is a subset of the vertex set of G.
- $E_H \subseteq E$, i.e., the edge set of H is a subset of the edge set of G.
- Each edge in E_H connects vertices in V_H.

Definition 2.3 (Tree in a Graph)**.** Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. A subgraph $T = (V_T, E_T)$ of G is called a *tree* if it satisfies the following conditions:

1. *Acyclicity:* T does not contain any cycles. Formally, for any subset of edges $E' \subseteq E_T$, the graph (V_T, E') does not contain a closed path.
2. *Connectivity:* For every pair of vertices $u, v \in V_T$, there exists a unique path in T connecting u and v.
3. *Minimality:* T contains exactly $|V_T| - 1$ edges, where $|V_T|$ is the number of vertices in T.

Definition 2.4 (Path in a Graph)**.** Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. A *path* in G is a sequence of vertices $P = (v_1, v_2, \ldots, v_k)$ such that:

1. For every $i = 1, 2, \ldots, k-1$, $(v_i, v_{i+1}) \in E$, i.e., there is an edge connecting consecutive vertices in the sequence.
2. All vertices $v_1, v_2, \ldots, v_k$ are distinct, ensuring that the path does not revisit any vertex (a *simple path*).

The *length* of the path is the number of edges in the sequence, which is $k - 1$.

Definition 2.5 (Eulerian circuit)**.** An Eulerian circuit is a closed path in a graph that traverses every edge exactly once, starting and ending at the same vertex.

Definition 2.6 (Hypergraph)**.** [13] A *hypergraph* $H = (V, E)$ is a generalization of a graph, consisting of:

- A set V, called the *vertex set*, where each element $v \in V$ represents a vertex.
- A set E, called the *hyperedge set*, where each element $e \in E$ is a subset of V, representing a hyperedge. Thus, $e \subseteq V$.

Key properties of a hypergraph:

- The hyperedge set E is a subset of the power set of V, i.e., $E \subseteq \mathcal{P}(V)$, where $\mathcal{P}(V)$ denotes the collection of all subsets of V.
- Unlike standard graphs where edges connect exactly two vertices, in a hypergraph, a hyperedge can connect any number of vertices, including just one vertex or the entire vertex set.

Example 2.7. For example, given a vertex set $V = \{v_1, v_2, v_3, v_4\}$, a hypergraph can have hyperedges such as:

$$E = \{\{v_1, v_2\}, \{v_3\}, \{v_1, v_3, v_4\}\}.$$

Here, the hyperedge $\{v_1, v_3, v_4\}$ connects three vertices simultaneously, illustrating the generality of hyperedges.

Definition 2.8 (Hypertree)**.** [55] A *hypertree* is a hypergraph $H = (V, E)$ with the following properties:

1. *Tree-like Structure:* There exists a tree $T = (V_T, E_T)$, called the *host tree*, such that:
   - Each vertex $t \in V_T$ is associated with a *bag* $B_t \subseteq V$, where B_t is a subset of the vertices of H.
   - Each hyperedge $e \in E$ of the hypergraph is a subset of at least one bag B_t, i.e., $\exists t \in V_T$ such that $e \subseteq B_t$.
2. *Connectivity Condition:* For any vertex $v \in V$, the set of nodes $t \in V_T$ where $v \in B_t$ forms a connected subtree of T. This ensures that each vertex in H is consistently represented across the tree structure.
3. *Acyclicity Condition:* The host tree T must be acyclic, maintaining the tree-like structure of the decomposition.

Definition 2.9 (Hyperpath in a Hypergraph)**.** ([22, 70, 74]) Let $H = (V, E)$ be a hypergraph, where V is the set of vertices and $E \subseteq \mathcal{P}(V)$ is the set of hyperedges. A *hyperpath* P in H connecting two vertices $u, v \in V$ is a sequence of hyperedges:

$$P = (e_1, e_2, \ldots, e_k) \quad \text{with } e_i \in E \text{ for } i = 1, \ldots, k,$$

satisfying the following conditions:

- *Adjacency Condition:* For all $1 \leq i \leq k-1$, $e_i \cap e_{i+1} \neq \emptyset$, i.e., consecutive hyperedges share at least one common vertex.
- *Endpoint Condition:* $u \in e_1$ and $v \in e_k$.
- *Acyclic Condition:* The sequence does not form a cycle, meaning the set of vertices visited by P, denoted as $V(P) = \bigcup_{i=1}^{k} e_i$, does not contain repeated visits to the same hyperedge.

The *length* of the hyperpath is the number of hyperedges in P, denoted $|P| = k$.

2.2 Tree-width and Hypertree-width

Tree-width quantifies how closely a graph resembles a tree by representing it using a tree-like structure with minimal width [92–96]. Hypertree-width generalizes this concept to hypergraphs, measuring how effectively a hypergraph can be decomposed into a tree-like structure [2, 3, 52–55, 75, 121]. The formal definitions of Tree-width and Hypertree-width are presented below.

Definition 2.10. [96] A tree-decomposition of an undirected graph G is a pair (T, W), where T is a tree, and $W = (W_t \mid t \in V(T))$ is a family of subsets that associates with every node t of T a subset W_t of vertices of G such that:

(T1) $\bigcup_{t \in V(T)} W_t = V(G)$,

(T2) For each edge $(u, v) \in E(G)$, there exists some node t of T such that $\{u, v\} \subseteq W_t$, and

(T3) For all nodes r, s, t in T, if s is on the unique path from r to t then $W_r \cap W_t \subseteq W_s$.

The width of a tree-decomposition (T, W) is the maximum of $|W_t| - 1$ over all nodes t of T. The tree-width of G is the minimum width over all tree-decompositions of G.

Definition 2.11. [3] A *generalized hypertree decomposition* of a hypergraph $H = (V(H), E(H))$ is a triple (T, B, C), where:

- (T, B) is a *tree decomposition* of H, where:
  - T is a tree with vertex set $V(T)$,
  - $B = \{B_t \mid t \in V(T)\}$ is a family of subsets of $V(H)$, called *bags*, satisfying the tree decomposition properties.
- $C = \{C_t \mid t \in V(T)\}$ is a family of subsets of $E(H)$ (hyperedges of H), called *guards*.

The decomposition must satisfy the following conditions for each $t \in V(T)$:

1. $B_t \subseteq \bigcup C_t$, where $\bigcup C_t$ is defined as:

$$\bigcup C_t = \{v \in V(H) \mid \exists e \in C_t : v \in e\}.$$

   In other words, every vertex in B_t must belong to at least one hyperedge in C_t.

The *width* of the generalized hypertree decomposition (T, B, C) is defined as:

$$\text{width}(T, B, C) = \max\{|C_t| \mid t \in V(T)\},$$

where $|C_t|$ denotes the number of hyperedges in C_t.

The *generalized hypertree width* of H, denoted $\text{ghw}(H)$, is the minimum width among all possible generalized hypertree decompositions of H.

A *hypertree decomposition* of H is a special case of a generalized hypertree decomposition (T, B, C) that satisfies the following additional condition for all $t \in V(T)$:

$$(\bigcup C_t) \cap \bigcup_{u \in V(T_t)} B_u \subseteq B_t,$$

where T_t is the subtree of T rooted at t, and $\bigcup_{u \in V(T_t)} B_u$ denotes the union of the bags associated with all nodes in T_t.

The *width* of a hypertree decomposition is defined in the same way as for a generalized hypertree decomposition. The *hypertree width* of H, denoted $\text{hw}(H)$, is the minimum width among all possible hypertree decompositions of H.

2.3 SuperHyperGraph and Superhypertree

A Superhypertree is a tree in a Superhypergraph [39, 51]. A Superhypergraph is known as a generalization of concepts such as graphs and hypergraphs (cf. [39, 42, 59, 60, 89, 105–108, 110, 110, 111]). The definitions, including related concepts, are provided below.

Definition 2.12 (SuperHyperGraph)**.** [105, 106] Let V be a finite set of vertices. A *superhypergraph* is an ordered pair $H = (V, E)$, where:

- $V \subseteq P(V)$ (the power set of V), meaning that each element of V can be either a single vertex or a subset of vertices (called a *supervertex*).
- $E \subseteq P(V)$ represents the set of edges, called *superedges*, where each $e \in E$ can connect multiple supervertices.

In this framework, a superhypergraph can accommodate complex relationships among groups of vertices, including single edges, hyperedges, superedges, and multi-edges. Superhypergraphs provide a flexible structure to represent high-order and hierarchical relationships.

Proposition 2.13. *[42] Every superhypergraph can be transformed into a hypergraph.*

Proof. Refer to [42] for details. □

Definition 2.14 (SuperHyperTree)**.** [39, 51] A *SuperHyperTree (SHT)* is a SuperHyperGraph $\text{SHT} = (V, E)$ that satisfies the following conditions:

1. *Host Tree Condition*: There exists a tree $T = (V_T, E_T)$, called the *host tree*, such that:
   - The vertex set of T is $V_T = V$.
   - Each superedge $e \in E$ corresponds to a connected subtree of T.
2. *Acyclicity Condition*: The host tree T must be acyclic, ensuring that SHT inherits this property.
3. *Connectedness Condition*: For any $v, w \in V$, there exists a sequence of superedges $e_1, e_2, \ldots, e_k \in E$ such that $v \in e_1$, $w \in e_k$, and $e_i \cap e_{i+1} \neq \emptyset$ for $1 \leq i < k$.

Definition 2.15 (SuperHyperpath in a SuperHypergraph)**.** Let SHG $= (V, E)$ be a superhypergraph, where $V \subseteq \mathcal{P}(V_0)$ is a set of supervertices (each being a subset of some base set V_0) and $E \subseteq \mathcal{P}(V)$ is a set of superhyperedges. A *superhyperpath* P in SHG connecting two supervertices $u, v \in V$ is a sequence of superhyperedges:

$$P = (e_1, e_2, \ldots, e_k) \quad \text{with } e_i \in E \text{ for } i = 1, \ldots, k,$$

satisfying the following conditions:

- *Adjacency Condition:* For all $1 \leq i \leq k-1$, $\bigcup e_i \cap \bigcup e_{i+1} \neq \emptyset$, i.e., the union of vertices in consecutive superhyperedges share at least one vertex in the base set V_0.
- *Endpoint Condition:* $u \in e_1$ and $v \in e_k$.
- *Acyclic Condition:* The sequence does not form a cycle, meaning the set of supervertices visited by P, denoted as $V(P) = \bigcup_{i=1}^{k} e_i$, does not revisit the same supervertex in the same sequence.

The *length* of the superhyperpath is the number of superhyperedges in P, denoted $|P| = k$.

Proposition 2.16. *A* SuperHyperpath *P in a superhypergraph SHG $= (V, E)$ is a* SuperHyperTree (SHT) *if and only if P satisfies the Host Tree Condition, Acyclicity Condition, and Connectedness Condition of a SuperHyperTree.*

Proof. We will prove the proposition in two parts:

- *(1) If P is a SuperHyperpath, then P satisfies the conditions of a SuperHyperTree.*
- *(2) If P satisfies the conditions of a SuperHyperTree, then P is a SuperHyperpath.*

(1) P as a SuperHyperpath implies P is a SuperHyperTree Let $P = (e_1, e_2, \ldots, e_k)$ be a SuperHyperpath.

1. *Host Tree Condition:* Since P is a SuperHyperpath, its superedges $e_1, e_2, \ldots, e_k$ are connected sequentially such that $\bigcup e_i \cap \bigcup e_{i+1} \neq \emptyset$ for $1 \leq i \leq k-1$. We can construct a tree $T = (V_T, E_T)$, where:
$$V_T = \{e_1, e_2, \ldots, e_k\}, \quad E_T = \{(e_i, e_{i+1}) \mid \bigcup e_i \cap \bigcup e_{i+1} \neq \emptyset\}.$$
Thus, P satisfies the Host Tree Condition.
2. *Acyclicity Condition:* By definition, P is a path and does not revisit any supervertex in the sequence of superedges. Therefore, the constructed tree T is acyclic.
3. *Connectedness Condition:* In P, every pair of supervertices $u, v \in V$ in $V(P) = \bigcup_{i=1}^{k} e_i$ is connected through a sequence of superedges $e_1, e_2, \ldots, e_k$. Hence, P satisfies the Connectedness Condition.

Since all three conditions of a SuperHyperTree are satisfied, P is a SuperHyperTree.

(2) P satisfies SuperHyperTree conditions implies P is a SuperHyperpath Let P satisfy the Host Tree Condition, Acyclicity Condition, and Connectedness Condition of a SuperHyperTree.

1. By the Host Tree Condition, there exists a tree $T = (V_T, E_T)$, where each superedge $e \in E$ corresponds to a subtree of T. In particular, the tree structure ensures that consecutive superedges e_i and e_{i+1} share at least one vertex in the base set V_0. Thus, P satisfies the Adjacency Condition of a SuperHyperpath.
2. By the Acyclicity Condition, P forms a simple path with no repeated supervertices in the sequence of superedges. Therefore, P satisfies the Acyclic Condition of a SuperHyperpath.

3. By the Connectedness Condition, P ensures that for any pair of vertices $u, v \in V(P)$, there exists a sequence of superedges $e_1, e_2, \ldots, e_k$ connecting u and v. Hence, P satisfies the Endpoint Condition of a SuperHyperpath.

Since P satisfies all the conditions of a SuperHyperpath, P is a SuperHyperpath.

From parts (1) and (2), we conclude that a SuperHyperpath is a SuperHyperTree if and only if it satisfies the conditions of a SuperHyperTree. □

Proposition 2.17. *A SuperHyperTree generalizes a Hypertree.*

Proof. Let $H = (V, E)$ be a hypergraph that satisfies the conditions of a Hypertree:

- There exists a host tree $T = (V_T, E_T)$, where each vertex $t \in V_T$ is associated with a bag $B_t \subseteq V$, and each hyperedge $e \in E$ is contained within at least one bag B_t, i.e., $e \subseteq B_t$ for some $t \in V_T$.
- The connectivity condition ensures that for any $v \in V$, the set of tree nodes $t \in V_T$ where $v \in B_t$ forms a connected subtree of T.
- The host tree T is acyclic.

Consider a SuperHyperTree $\text{SHT} = (V_{\text{SHT}}, E_{\text{SHT}})$, where $V_{\text{SHT}} \subseteq \mathcal{P}(V)$ (the power set of the vertices of H), and $E_{\text{SHT}} \subseteq \mathcal{P}(V_{\text{SHT}})$. By definition, a SuperHyperTree satisfies:

- There exists a host tree T_{SHT} such that each superedge $e \in E_{\text{SHT}}$ corresponds to a connected subtree of T_{SHT}.
- The host tree T_{SHT} is acyclic.
- The connectivity condition ensures that any two vertices $v, w \in V_{\text{SHT}}$ can be connected through a sequence of superedges.

To show that a Hypertree H is a special case of a SuperHyperTree SHT:

1. Let each vertex $v \in V$ of the Hypertree H correspond to a singleton set $\{v\} \in V_{\text{SHT}}$, making $V_{\text{SHT}} = \{\{v\} \mid v \in V\}$.
2. Let each hyperedge $e \in E$ of H correspond to a superedge $e_{\text{SHT}} \in E_{\text{SHT}}$, where $e_{\text{SHT}} = \{\{v\} \mid v \in e\}$.

With this mapping:

- Each hyperedge $e \in E$ is contained within a bag B_t of the Hypertree H, and thus the corresponding superedge e_{SHT} forms a subtree in T_{SHT}.
- The connectivity condition and acyclicity of the Hypertree H directly translate to the SuperHyperTree SHT.

Therefore, a Hypertree H is a specific case of a SuperHyperTree SHT, where the supervertices of SHT are restricted to singleton sets. □

Proposition 2.18. *A SuperHyperPath generalizes a HyperPath.*

Proof. Let $H = (V, E)$ be a hypergraph and $P = (e_1, e_2, \ldots, e_k)$ be a hyperpath in H, satisfying:

- For all $1 \leq i \leq k-1$, $e_i \cap e_{i+1} \neq \emptyset$, ensuring that consecutive hyperedges share at least one vertex.
- $u \in e_1$ and $v \in e_k$, ensuring that P connects u to v.
- P does not form a cycle, ensuring acyclicity.

Consider a SuperHyperPath P_{SHT} in a SuperHyperGraph SHG = $(V_{\text{SHT}}, E_{\text{SHT}})$, where:

- $V_{\text{SHT}} \subseteq \mathcal{P}(V)$, and $E_{\text{SHT}} \subseteq \mathcal{P}(V_{\text{SHT}})$.
- $P_{\text{SHT}} = (e_1, e_2, \ldots, e_k)$ satisfies:
  - $\bigcup e_i \cap \bigcup e_{i+1} \neq \emptyset$, ensuring that consecutive superedges share at least one vertex.
  - $u \in e_1$ and $v \in e_k$, where $u, v \in V_{\text{SHT}}$, ensuring that P_{SHT} connects u to v.
  - P_{SHT} does not form a cycle.

To show that a HyperPath P is a special case of a SuperHyperPath P_{SHT}:

1. Let each vertex $v \in V$ correspond to a singleton set $\{v\} \in V_{\text{SHT}}$.
2. Let each hyperedge $e_i \in P$ correspond to a superedge $e_i^{\text{SHT}} \in E_{\text{SHT}}$, where $e_i^{\text{SHT}} = \{\{v\} \mid v \in e_i\}$.

This mapping preserves:

- The adjacency condition, as $e_i \cap e_{i+1} \neq \emptyset$ in P implies $\bigcup e_i^{\text{SHT}} \cap \bigcup e_{i+1}^{\text{SHT}} \neq \emptyset$ in P_{SHT}.
- The endpoint condition, as $u \in e_1$ and $v \in e_k$ in P correspond to $u \in e_1^{\text{SHT}}$ and $v \in e_k^{\text{SHT}}$ in P_{SHT}.
- The acyclicity condition.

Thus, a HyperPath P is a specific case of a SuperHyperPath P_{SHT}, where the supervertices are singleton sets. □

2.4 SuperHyperTree-width

SuperHyperTree-width is an abstraction of Hypertree-width, extending the concept of Tree-width. The definition is presented as follows [39].

Definition 2.19 (SuperHyperTree Decomposition and SuperHyperTree-width)**.** [39] Let SHT = (V, E) be a SuperHyperGraph. A *SuperHyperTree decomposition* of SHT is a tuple $(T, \mathcal{B}, C)$, where:

- $T = (V_T, E_T)$ is a tree.
- $\mathcal{B} = \{B_t \mid t \in V_T\}$, a collection of subsets of V (called *bags*), satisfying:
  1. For every superedge $e \in E$, there exists a node $t \in V_T$ such that $e \subseteq B_t$.
  2. For every vertex $v \in V$, the set $\{t \in V_T \mid v \in B_t\}$ induces a connected subtree of T.
- $C = \{C_t \mid t \in V_T\}$, a collection of subsets of E (called *guards*), such that:
  1. For every node $t \in V_T$, $B_t \subseteq \bigcup C_t$, where $\bigcup C_t = \{v \in V \mid \exists e \in C_t \text{ such that } v \in e\}$.
  2. For every node $t \in V_T$, $(\bigcup C_t) \cap \bigcup_{u \in T_t} B_u \subseteq B_t$, where T_t is the subtree of T rooted at t.

The *width* of a SuperHyperTree decomposition $(T, \mathcal{B}, C)$ is defined as:

$$\text{width}(T, \mathcal{B}, C) = \max_{t \in V_T} |C_t|.$$

The *SuperHyperTree-width* of SHT, denoted SHT-width(SHT), is the minimum width over all possible SuperHyperTree decompositions:

$$\text{SHT-width}(\text{SHT}) = \min_{(T, \mathcal{B}, C)} \text{width}(T, \mathcal{B}, C).$$

2.5 Basic Definition of Algorithm

The basic definitions related to the algorithms described in the Results section are provided here. Readers may refer to the Lecture Notes or the Introduction for additional details as needed [24, 31, 99].

Definition 2.20. (cf. [85, 99]) The *Total Time Complexity* of an algorithm is defined as the sum of the time required to execute each step of the algorithm, expressed as a function of the input size. If an algorithm involves multiple steps or operations, the total time complexity is determined by the maximum time required for the most time-consuming operation.

Formally, let $T(n, m)$ be the time complexity as a function of input sizes n and m. The total time complexity is:

$$T(n, m) = \max(T_{\text{operation1}}(n, m), T_{\text{operation2}}(n, m), \ldots, T_{\text{operationk}}(n, m)),$$

where n is the size of the set of propositions and m is the size or complexity of the context.

Definition 2.21. (cf. [85, 99]) The *Space Complexity* of an algorithm is the total amount of memory required to execute the algorithm, expressed as a function of the input size. This includes:

- The *input space*, which depends on the size of the input n, m,
- The *auxiliary space*, which includes temporary variables, data structures, or storage used during computation.

Formally, let $S(n, m)$ be the space complexity as a function of input sizes n and m. The total space complexity is:

$$S(n, m) = S_{\text{input}}(n, m) + S_{\text{auxiliary}}(n, m).$$

Definition 2.22. (cf. [85, 99]) *Big-O notation* is a mathematical concept used to describe the upper bound of the time or space complexity of an algorithm. Let $f(n)$ and $g(n)$ be functions that map non-negative integers to non-negative real numbers. We say:

$$f(n) \in O(g(n))$$

if there exist constants $c > 0$ and $n_0 \geq 0$ such that:

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

Definition 2.23 (NP-hard)**.** (cf. [63, 120]) A decision problem is classified as *NP-hard* if every problem in the class NP can be reduced to it in polynomial time. Formally, a problem P is NP-hard if there exists a polynomial-time reduction from any problem $Q \in$ NP to P, such that solving P allows the solution of Q. Importantly, NP-hard problems may not necessarily belong to the class NP, as they are not required to have a solution verifiable in polynomial time.

3 Results in this Paper

The results of this paper are presented below.

3.1 Exact Construct SuperHyperGraph

We consider about the algorithm of Constructing SuperHyperGraph. The problem considered in this subsection is described below.

Problem 3.1. Given a finite set of vertices V and a collection of subsets of V, construct a valid SuperHyperGraph $H = (V, E)$ where E is the set of superedges.

The algorithm and related theorems for the above problem are presented below.

Algorithm 1: Construct SuperHyperGraph

Input: A set $V = \{v_1, v_2, \ldots, v_n\}$ and a collection $C \subseteq \mathcal{P}(V)$.
Output: A SuperHyperGraph $H = (V, E)$.

```
Initialize E ← ∅;
foreach subset S ∈ C do
    if S ≠ ∅ then
        Add S to E;
return H = (V, E);
```

Theorem 3.2. *The algorithm constructs a valid SuperHyperGraph $H = (V, E)$ such that V and E satisfy the definition of a SuperHyperGraph.*

Proof. The algorithm iterates over the input collection C and includes each non-empty subset $S \in C$ into the set of superedges E. By definition:

- V remains unchanged and consists of the original set of vertices.
- $E \subseteq \mathcal{P}(V) \setminus \{\emptyset\}$, as only non-empty subsets are added.

Thus, the output $H = (V, E)$ satisfies the definition of a SuperHyperGraph. □

Theorem 3.3. *The time complexity of the algorithm is $O(|C|)$, where $|C|$ is the size of the input collection of subsets.*

Proof. The algorithm processes each subset $S \in C$ exactly once. For each subset, the inclusion check and addition to E take constant time. Therefore, the overall time complexity is $O(|C|)$. □

Theorem 3.4. *The space complexity of the algorithm is $O(|C| + |V|)$.*

Proof. The algorithm requires space to store:

- The vertex set V, requiring $O(|V|)$ space.
- The collection of superedges E, which is derived from C and requires $O(|C|)$ space.

Thus, the total space complexity is $O(|C| + |V|)$. □

3.2 Algorithm: Recognizing a SuperHyperTree

In this subsection, we examine the algorithm for recognizing a SuperHyperTree. For example, in the context of hypergraphs, recognizing algorithms for Hypertrees have been developed [114]. Here, we extend this concept to the SuperHyperTree framework. The problem considered in this subsection is described below.

Problem 3.5. Given a SuperHyperGraph $H = (V, E)$, determine whether H is a SuperHyperTree.

The algorithm and related theorems for the above problem are presented below.

Algorithm 2: Recognize SuperHyperTree

Input: A SuperHyperGraph $H = (V, E)$.
Output: `True` if H is a SuperHyperTree; `False` otherwise.

```
Construct a graph T = (V, E_T), where E_T consists of edges connecting all vertices in each e ∈ E;
Check if T is acyclic;
if T is not acyclic then
    return False;
foreach e ∈ E do
    Verify that e forms a connected subtree of T;
    if e does not form a connected subtree then
        return False;
Check that H satisfies the connectedness condition;
if H does not satisfy the connectedness condition then
    return False;
return True;
```

Theorem 3.6. *The algorithm correctly determines whether H is a SuperHyperTree.*

Proof. The algorithm verifies:

- The acyclicity of the host tree T.
- That each superedge $e \in E$ forms a connected subtree of T.
- That H satisfies the connectedness condition.

Since these are the defining properties of a SuperHyperTree, the algorithm is correct. □

Theorem 3.7. *The time complexity of the algorithm is $O(|V| + |E| \cdot |V|)$.*

Proof. Constructing the graph T requires $O(|V| + |E| \cdot |V|)$ time, as each superedge may connect multiple vertices. Checking acyclicity and verifying subtree conditions also require $O(|V| + |E| \cdot |V|)$ time. Thus, the overall time complexity is $O(|V| + |E| \cdot |V|)$. □

Theorem 3.8. *The space complexity of the algorithm is $O(|V| + |E|)$.*

Proof. The algorithm requires space to store the graph T and the original SuperHyperGraph H. Thus, the space complexity is $O(|V| + |E|)$. □

3.3 Computation of SuperHyperTree-width

This subsection examines the Algorithm for the Computation of SuperHyperTree-width. The problem considered in this subsection is described below.

Problem 3.9. The problem is to compute the exact SuperHyperTree-width of a given SuperHyperGraph $H = (V, E)$. This involves finding a SuperHyperTree decomposition $(\mathcal{T}, \mathcal{X})$ such that the width of the decomposition is minimized. The objective is to determine the smallest possible width, denoted as $w_{\min}$, satisfying the host tree, acyclicity, and connectedness conditions for all superhyperedges $e \in E$ within H.

3.3.1 Exact Algorithm for Computation of SuperHyperTree-width

The Exact Algorithm for the Computation of SuperHyperTree-width is introduced as follows.

Algorithm 3: Computation of SuperHyperTree-width

Input: SuperHyperGraph SHT = (V, E)
Output: SuperHyperTree-width $w_{\min}$ and the corresponding decomposition $(\mathcal{T}, \mathcal{X})$

```
Initialize a tree decomposition algorithm for SHT;
Set an initial upper bound for the tree-width: w_min ← ∞;
foreach possible tree decomposition (T, X) of SHT do
    Compute the width of the decomposition;
    if width of decomposition < w_min then
        Update w_min;
        Store the corresponding decomposition (T, X);
return w_min, (T, X);
```

Theorem 3.10. *The above algorithm correctly computes the SuperHyperTree-width of the given SuperHyperGraph SHT.*

Proof. Completeness: The algorithm considers all possible tree decompositions over the vertex set V and evaluates each decomposition to ensure that it satisfies the necessary conditions for a valid SuperHyperTree-decomposition. The algorithm guarantees that it explores every possible decomposition that could potentially minimize the SuperHyperTree-width.

Soundness: By systematically testing all feasible decompositions and comparing their widths, the algorithm ensures that the computed SuperHyperTree-width is indeed the smallest possible value, thus providing an accurate result. □

Theorem 3.11 (Time Complexity)**.** *The time complexity of the approximation algorithm for computing the SuperHyperTree-width is $O(2^{|V|^2})$, where $|V|$ is the number of vertices in the SuperHyperGraph.*

Proof. Let $H = (V, E)$ be the given SuperHyperGraph with vertex set V and edge set E. The core of the algorithm involves enumerating and processing various tree decompositions of the graph, which correspond to subsets of V.

The total number of possible tree decompositions over V is exponential in the number of vertices. Specifically, we need to consider all possible ways to partition V into subsets, which is of the order of $O(2^{|V|^2})$. This arises from the fact that each pair of vertices can either be in the same subset or in different subsets, leading to an exponential number of possible combinations.

Since each decomposition requires checking the structural constraints of the SuperHyperGraph (i.e., the superedges), the time complexity for evaluating each decomposition is polynomial in $|V|$. However, the dominant factor is the number of decompositions, which grows exponentially as $2^{|V|^2}$.

Thus, the overall time complexity of the algorithm is $O(2^{|V|^2})$. □

Theorem 3.12 (Space Complexity)**.** *The space complexity of the approximation algorithm for computing the SuperHyperTree-width is $O(2^{|V|^2})$, where $|V|$ is the number of vertices in the SuperHyperGraph.*

Proof. The space complexity of the algorithm is determined by the storage requirements for enumerating and processing the possible tree decompositions of the graph.

Each tree decomposition is represented as a set of subsets of the vertex set V. The number of subsets of V is $O(2^{|V|})$, and each subset requires storing a list of vertices. Thus, the space required for each decomposition is proportional to $O(2^{|V|})$.

Moreover, the algorithm needs to store all possible decompositions, which can be of the order $O(2^{|V|^2})$, as explained in the time complexity analysis. Therefore, the total space required is dominated by the number of decompositions, leading to a space complexity of $O(2^{|V|^2})$.

Thus, the overall space complexity is $O(2^{|V|^2})$. □

3.3.2 Approximation Algorithm for SuperHyperTree-width

An approximation algorithm provides near-optimal solutions to computationally hard problems within a provable error bound, ensuring efficiency and feasibility. Given a SuperHyperGraph $H = (V, E)$, we present an approximation algorithm for computing its SuperHyperTree-width. The algorithm is designed to run in polynomial time and approximates the SuperHyperTree-width within a factor of $O(\log |V|)$.

Algorithm 4: Approximation Algorithm for SuperHyperTree-width

Input: SuperHyperGraph $H = (V, E)$
Output: Approximation of the SuperHyperTree-width

```
Initialize an empty tree decomposition T;
Construct an initial decomposition using a greedy approach:;
foreach superedge e_i ∈ E do
    Select the smallest superedge e_i based on the number of vertices connected;
    Add e_i to the decomposition T;
Perform a greedy refinement:;
foreach superedge e_i in T do
    Try to connect e_i to existing parts of the decomposition with minimal additional width;
    if improvement is found then
        Adjust the decomposition;
Terminate the algorithm when no further improvements are possible;
return the SuperHyperTree-width of the final decomposition;
```

Theorem 3.13 (Correctness of the Approximation Algorithm)**.** *The algorithm correctly computes an approximation of the SuperHyperTree-width of the SuperHyperGraph H.*

Proof. The correctness of the algorithm follows from the fact that:

1. The greedy tree decomposition approach used in the algorithm always provides a decomposition whose treewidth is within a constant factor of the optimal treewidth.

2. The hypertree-width of a SuperHyperGraph is bounded above by the treewidth of any valid tree decomposition, and the algorithm uses such a decomposition to compute an approximation.

3. Therefore, the approximation computed by the algorithm will always be a valid approximation to the true SuperHyperTree-width. □

Theorem 3.14 (Time Complexity)**.** *Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of hyperedges in the SuperHyperGraph. The time complexity of the approximation algorithm is $O(n^2)$.*

Proof. The time complexity is dominated by the steps where we construct the tree decomposition and compute the hypertree-width approximation.

1. Constructing a tree decomposition with treewidth approximation τ can be done in $O(n^2)$ time using greedy algorithms or dynamic programming techniques for tree decomposition.

2. Computing the hypertree-width from the tree decomposition requires linear time in terms of the size of the decomposition, which is at most $O(n^2)$.

Therefore, the overall time complexity is $O(n^2)$. □

Theorem 3.15 (Space Complexity)**.** *Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of hyperedges in the SuperHyperGraph. The space complexity of the approximation algorithm is $O(n^2)$.*

Proof. The space required to store the SuperHyperGraph $H = (V, E)$ is $O(m)$, where m is the number of hyperedges. Additionally, the space required to store the tree decomposition and related data structures is $O(n^2)$, since the decomposition involves a set of nodes and edges that depend on the number of vertices and hyperedges in the graph. Thus, the overall space complexity is $O(n^2)$. □

Theorem 3.16 (Approximation Bound)**.** *Let SuperHyperTree-width(H) denote the true SuperHyperTree-width of H, and τ the approximation computed by the algorithm. Then, the algorithm computes an approximation such that:*

$$\textit{SuperHyperTree-width}(H) \leq \tau \leq \beta \cdot \textit{SuperHyperTree-width}(H),$$

where β is a constant factor depending on the quality of the greedy heuristic used in the tree decomposition algorithm (e.g., $\beta = 2$ for simple greedy heuristics).

Proof. The approximation bound follows from the properties of the greedy tree decomposition algorithm. In particular, if the tree decomposition has treewidth τ, the hypertree-width of the SuperHyperGraph is at most τ. Since the greedy heuristic for tree decomposition guarantees that the treewidth is within a constant factor β of the optimal treewidth, the approximation τ computed by the algorithm will be within a factor of β of the true SuperHyperTree-width. □

3.4 Superhypergraph Partition Problem

We consider the well-known Graph Partition Problem. The Graph Partition Problem aims to divide the vertices of a graph into disjoint subsets while minimizing edge cuts or maximizing intra-group connectivity [33,71,100]. The Hypergraph Partition Problem generalizes this by partitioning the vertices of a hypergraph into subsets while minimizing hyperedge cuts, where hyperedges connect multiple vertices [34,69,98,113]. These problems are further extended in this study to the context of superhypergraphs. The problem considered in this subsection is described below.

Problem 3.17 (Superhypergraph Partition Problem)**.** Given a superhypergraph $H = (V, E)$ and an integer $k \geq 2$, partition V into k disjoint subsets $V_1, V_2, \ldots, V_k$ such that:

- Balance Constraint: For each i, $|V_i| \leq (1 + \epsilon) \left\lceil \frac{|V|}{k} \right\rceil$ for some small $\epsilon > 0$.
- Cut Minimization: The number of hyperedges that are *cut* is minimized. A hyperedge is said to be *cut* if it contains vertices from more than one partition.

The algorithm for solving this problem is presented below.

Remark 3.18. The algorithm consists of the following steps:

1. *Coarsening Phase*: Reduce the size of the superhypergraph by collapsing vertices and hyperedges to create a hierarchy of smaller superhypergraphs.
    - *Matching*: Pair vertices based on some similarity metric (e.g., the number of shared hyperedges).
    - *Aggregation*: Merge matched vertices to form supervertices.
    - *Hyperedge Reduction*: Adjust hyperedges to reflect the new supervertices.
2. *Initial Partitioning*: Use a simple partitioning algorithm (e.g., greedy assignment) on the coarsest superhypergraph H_l.

3. *Uncoarsening Phase*: Project the partition back onto the original superhypergraph, refining the partition at each level to improve the cut size and balance. In the algorithm, At each level i, project the partition P_{i+1} onto H_i and refine it using a local optimization method (e.g., Kernighan–Lin algorithm adapted for superhypergraphs).

Algorithm 5: Superhypergraph Partitioning Algorithm

Input: Superhypergraph $H = (V, E)$, number of partitions $k \geq 2$
Output: Partition $V_1, V_2, \ldots, V_k$ of V

```
H_0 ← H;
l ← 0;
while Size of H_l is greater than threshold do
    H_{l+1} ← Coarsen(H_l);
    l ← l + 1;
P_l ← InitialPartition(H_l, k);
for i ← l − 1 down to 0 do
    P_i ← Refine(H_i, P_{i+1});
return P_0;
```

Theorem 3.19. *The algorithm produces a valid partition of the vertex set V into k disjoint subsets $V_1, V_2, \ldots, V_k$ that satisfy the balance constraint.*

Proof. The algorithm maintains the balance constraint at each level:

- During initial partitioning, the algorithm assigns vertices to partitions such that the balance constraint is satisfied.
- During uncoarsening and refinement, vertices are moved between partitions only if the balance constraint is not violated.

Since the coarsening and uncoarsening processes are designed to preserve the structure of the original superhypergraph, the final partition P_0 is valid and satisfies the balance constraint. □

Theorem 3.20. *The time complexity of the algorithm is $O(|E| \log |V|)$.*

Proof. The algorithm consists of multiple phases:

- *Coarsening Phase*:
  - Each level reduces the number of vertices by a constant factor.
  - The number of levels is $O(\log |V|)$.
  - At each level, matching and aggregation can be done in $O(|E|)$ time.
- *Initial Partitioning*:
  - The coarsest superhypergraph has significantly fewer vertices.
  - Partitioning can be done in $O(1)$ time relative to the original graph size.
- *Uncoarsening and Refinement*:
  - At each level, refinement operations (e.g., swapping vertices between partitions) can be performed in $O(|E|)$ time.
  - There are $O(\log |V|)$ levels.

Therefore, the total time complexity is $O(|E| \log |V|)$. □

Theorem 3.21. *The space complexity of the algorithm is $O(|V| + |E|)$.*

Proof. At each level, the algorithm stores:

- The superhypergraph H_i, which has at most $|V|$ vertices and $|E|$ hyperedges.
- The partition P_i, which is a mapping from vertices to partition indices.

Since the size of H_i decreases with each level, the total space required is dominated by the original superhypergraph H, requiring $O(|V| + |E|)$ space. □

Theorem 3.22. *The SuperHypergraph Partition Problem is NP-hard.*

Proof. We reduce the Graph Partition Problem (GPP), which is known to be NP-hard, to the SuperHypergraph Partition Problem (SHGP). Let $G = (V_G, E_G)$ be an instance of GPP. Construct a superhypergraph SHG = (V, E) as follows:

- For each vertex $v \in V_G$, create a supervertex $\{v\} \in V$.
- For each edge $(u, v) \in E_G$, create a superhyperedge $\{\{u\}, \{v\}\} \in E$.

Any solution to SHGP corresponds to a partition of V_G that minimizes edge cuts in G. Thus, SHGP is at least as hard as GPP, proving its NP-hardness. □

3.5 Reachability Problem in Superhypergraph

The Reachability Problem determines whether there exists a path between two vertices in a graph or hypergraph using its edges [7, 8]. The problem considered in this subsection is described below.

Definition 3.23. Given a Superhypergraph $H = (V, E)$, where V is a finite set of vertices and E is a set of hyperedges, the Reachability Problem asks whether there exists a path from a vertex $v_1 \in V$ to a vertex $v_2 \in V$ using hyperedges in E.

The algorithm for solving the Reachability problem follows a depth-first search (DFS) [115] strategy adapted for hypergraphs. In a traditional DFS, we explore vertices by following edges. For a Superhypergraph, a hyperedge can connect any number of vertices, and we need to modify the DFS to consider all vertices that a hyperedge connects.

Algorithm 6: Reachability in Superhypergraph

Input: Superhypergraph $H = (V, E)$, vertices $v_1, v_2 \in V$
Output: `True` if there is a path from v_1 to v_2, `False` otherwise

```
Initialize a set visited ← ∅;
Call DFS(v1, v2, H, visited);
if DFS returns True then
    return True;
else
    return False;
```

Theorem 3.24. *The algorithm explores all reachable vertices from v_1 by traversing through all hyperedges in H. Since each hyperedge connects a set of vertices, DFS ensures that if there is a path from v_1 to v_2, it will be discovered.*

Proof. If there is a path from v_1 to v_2, DFS will eventually visit v_2 by following the hyperedges. If v_2 is not reachable, the algorithm will not visit v_2, and the function will return `False`. Thus, the algorithm is correct. □

Theorem 3.25. *The time complexity is* $O(|V| + |E|)$.

Proof. In the worst case, the algorithm needs to explore all vertices and all hyperedges in the graph. Since there are $|V|$ vertices and $|E|$ hyperedges, the time complexity is $O(|V| + |E|)$. □

Theorem 3.26. *The Space Complexity is* $O(|V| + |E|)$.

Proof. The space complexity is dominated by the storage of the visited set and the recursion stack in the DFS. Therefore, the space complexity is $O(|V|)$. □

3.6 Minimum Spanning SuperHypertree Problem

The Minimum Spanning Tree Problem identifies a tree connecting all graph vertices with the minimum total edge weight [4, 57, 124]. The Minimum Spanning Hypertree Problem extends this concept to hypergraphs, seeking a tree-like structure minimizing hyperedge weights [118]. In this subsection, we examine the Minimum Spanning SuperHypertree Problem. The problem considered is described below.

Definition 3.27 (Weighted SuperHyperGraph)**.** Let V be a finite set of vertices. A *weighted superhypergraph* is an ordered pair $H = (V, E)$, where:

- $V \subseteq \mathcal{P}(V)$, the power set of V, meaning that each element of V can be either a single vertex or a subset of vertices (called a *supervertex*).
- E is a set of *superhyperedges*, where each $e \in E$ is a non-empty subset of V (i.e., $e \subseteq V$ and $e \neq \emptyset$).
- Each superhyperedge $e \in E$ has an associated positive weight $w(e) \in \mathbb{R}^+$.

Definition 3.28 (Minimum Spanning SuperHypertree Problem)**.** Given a weighted superhypergraph $H = (V, E)$, the *Minimum Spanning SuperHypertree (MSST)* is a subgraph $T = (V, E_T)$ satisfying:

1. $E_T \subseteq E$.
2. T is a *superhypertree*, meaning it satisfies the conditions of a SuperHyperTree as defined below.
3. The total weight $w(T) = \sum_{e \in E_T} w(e)$ is minimized among all possible superhypertrees of H.

We propose an algorithm inspired by Kruskal's algorithm [18] for finding a Minimum Spanning Tree (MST) in graphs. The algorithm operates as follows:

Algorithm 7: Minimum Spanning SuperHypertree Algorithm

Input: A weighted superhypergraph $H = (V, E)$
Output: A Minimum Spanning SuperHypertree $T = (V, E_T)$

```
Initialize the edge set of the superhypertree: E_T ← ∅;
Sort the superhyperedges E in non-decreasing order of weight;
Initialize a disjoint-set data structure 𝒟 for the vertices in V;
foreach superhyperedge e ∈ E (in sorted order) do
    Compute the union of sets containing vertices in e:
    C_e ← ⋃_{v∈e} Find(v);
    if |C_e| > 1 then
        Add e to E_T;
        Update the disjoint-set structure by performing:
        Union(Find(v)) for all v ∈ e;
Construct the Minimum Spanning SuperHypertree: T ← (V, E_T);
return T;
```

Theorem 3.29. *The algorithm correctly finds a Minimum Spanning SuperHypertree of the weighted superhypergraph H.*

Proof. We need to show that:

1. The algorithm produces a superhypertree.
2. The superhypertree is spanning and minimal in total weight.

1. Produces a SuperHypertree:

Acyclicity: The algorithm only adds superhyperedges that connect disjoint components, ensuring no cycles are formed.

Connectivity: By uniting components whenever a superhyperedge is added, eventually all vertices become connected.

2. Minimal Total Weight:

The algorithm always chooses the smallest available superhyperedge that does not create a cycle, similar to Kruskal's algorithm.

Suppose there exists another superhypertree T' with a smaller total weight. Then, there must be at least one superhyperedge e in T' not in T with weight less than or equal to the heaviest superhyperedge in T.

Replacing superhyperedges in T with those in T' cannot lead to a total weight less than T without violating acyclicity or connectivity.Therefore, T is minimal. □

Theorem 3.30. *The time complexity of the algorithm is $O(|E| \log |E| + |E| \cdot \alpha(|V|))$, where α is the inverse Ackermann function.*

Proof. The algorithm can be divided into the following steps:

- *Sorting the Superhyperedges:*
  - Sorting $|E|$ superhyperedges by weight takes $O(|E| \log |E|)$ time.
- *Processing Each Superhyperedge:*
  - For each superhyperedge e, the following operations are performed:
    - *Find Operations:* For each vertex $v \in e$, a Find(v) operation is performed. The total number of find operations is $O\left(\sum_{e \in E} |e|\right)$, which is proportional to the size of all superhyperedges combined.
    - *Union Operation:* If e is added to the spanning superhypertree, a union operation is performed. The number of union operations is at most $|E_T| \leq |V| - 1$, where $|E_T|$ is the number of edges in the final superhypertree.
  - Each Union-Find operation (find or union) takes $O(\alpha(|V|))$ time.

Assuming the total size of all superhyperedges is $\sum_{e \in E} |e| = O(|E| \cdot m)$, where m is the maximum size of a superhyperedge:

- The time complexity for Union-Find operations becomes $O(|E| \cdot m \cdot \alpha(|V|))$.

However, in practical cases:

- The Union-Find operations are dominated by $O(|E| \cdot \alpha(|V|))$, as $\alpha(|V|)$ grows extremely slowly, and m (the size of superhyperedges) is typically much smaller than $|V|$.

Thus, the overall time complexity of the algorithm is:

$$O(|E| \log |E| + |E| \cdot \alpha(|V|)).$$

□

Theorem 3.31. *The space complexity of the algorithm is* $O(|V| + |E|)$.

Proof. The space requirements for the algorithm are as follows:

- *Storage for Vertices* V: $O(|V|)$ space is required to store the vertex set.
- *Storage for Superhyperedges* E: $O(|E|)$ space is required to store the list of superhyperedges.
- *Union-Find Data Structure:* $O(|V|)$ space is required to store parent and rank arrays for the Union-Find operations.
- *Edge List of the Spanning SuperHypertree* E_T: $O(|E|)$ space is required in the worst case, where all superhyperedges are included.

Combining these, the total space complexity is:

$$O(|V| + |E|).$$

□

Example 3.32. To illustrate the algorithm, consider the following weighted superhypergraph $H = (V, E)$:

- $V = \{v_1, v_2, v_3, v_4\}$.
- $E = \{e_1, e_2, e_3\}$, with:
  - $e_1 = \{v_1, v_2\}$, $w(e_1) = 1$.
  - $e_2 = \{v_2, v_3, v_4\}$, $w(e_2) = 2$.
  - $e_3 = \{v_1, v_4\}$, $w(e_3) = 3$.

Step-by-Step Execution of the Algorithm:

1. *Sort* E:
   - Sort the superhyperedges by weight: e_1, e_2, e_3.
2. *Initialize Union-Find Data Structure*:
   - Each vertex $v \in V$ is its own set.
3. *Process* $e_1 = \{v_1, v_2\}$:
   - Find$(v_1) \neq$ Find(v_2).
   - Add e_1 to E_T.
   - Union v_1 and v_2.
4. *Process* $e_2 = \{v_2, v_3, v_4\}$:
   - Find$(v_2) \neq$ Find(v_3) and Find$(v_2) \neq$ Find(v_4).

- Add e_2 to E_T.
- Union v_2, v_3, and v_4.

5. *Process* $e_3 = \{v_1, v_4\}$:
   - All vertices are now in the same connected component.
   - Adding e_3 would create a cycle, so e_3 is skipped.

The resulting Minimum Spanning SuperHypertree is:

$$T = (V, \{e_1, e_2\}),$$

with a total weight:

$$w(T) = 1 + 2 = 3.$$

This example demonstrates the step-by-step execution of the algorithm for finding the Minimum Spanning SuperHypertree in a weighted superhypergraph. The analysis confirms the correctness and efficiency of the algorithm, with a time complexity of $O(|E| \log |E| + |E| \cdot \alpha(|V|))$ and a space complexity of $O(|V| + |E|)$.

3.7 Single-Source Shortest Path Problem in a SuperHypergraph

The Single-Source Shortest Path Problem [77, 84, 88] in a SuperHypergraph involves finding the shortest paths from a source vertex to all other vertices, minimizing the total weight of traversed superhyperedges. The problem considered is described below.

Problem 3.33. Given a weighted superhypergraph SHG = (V, E) and a source supervertex $s \in V$, the *Single-Source Shortest Path Problem* seeks to find the shortest superhyperpaths from s to all other supervertices $v \in V$, minimizing the total weight.

The algorithm and related theorems for the above problem are presented below.

Algorithm 8: Single-Source Shortest Path in SuperHypergraph

Input: A weighted superhypergraph SHG = (V, E), source supervertex $s \in V$
Output: Shortest path distances $d(v)$ from s to each $v \in V$

```
Initialize d(v) ← ∞ for all v ∈ V; set d(s) ← 0
Initialize priority queue Q ← {(s, d(s))}
while Q is not empty do
    Extract supervertex u with minimal d(u) from Q
    foreach superhyperedge e ∈ E such that u ∈ e do
        foreach supervertex v ∈ e do
            if d(v) > d(u) + w(e) then
                d(v) ← d(u) + w(e)
                Insert or update v in Q with priority d(v)
return d(v) for all v ∈ V
```

Theorem 3.34. *Algorithm 8 correctly computes the shortest path distances from the source supervertex s to every other supervertex $v \in V$ in SHG.*

Proof. We prove the correctness of the algorithm by induction on the number of supervertices whose shortest path distances from s have been finalized.

Base Case: Initially, only s has a finalized distance $d(s) = 0$, which is correct.

Inductive Step: Assume that the distances $d(u)$ are correct for all supervertices whose shortest paths have been found. When we extract the supervertex u with the minimal tentative distance $d(u)$ from the priority queue,

we know that $d(u)$ is the shortest possible distance from s to u. This is because all weights $w(e)$ are positive, and any alternative path to u via other supervertices would have a total weight at least $d(u)$ or greater.

For each superhyperedge e containing u, we consider all supervertices $v \in e$. If the path from s to v via e and u offers a shorter distance than the current $d(v)$, we update $d(v)$ accordingly. Since we consider all such superhyperedges and supervertices, we ensure that the shortest distances are propagated throughout the superhypergraph.

By continuously selecting the supervertex with the minimal tentative distance and updating distances of adjacent supervertices, we guarantee that once a supervertex u is extracted from Q, $d(u)$ is indeed the shortest distance from s to u.

Therefore, by induction, the algorithm correctly computes the shortest path distances from s to all supervertices in V. □

Theorem 3.35. *The time complexity of the algorithm is $O(|E| \cdot m \cdot \log |V|)$, where m is the maximum number of supervertices in a superhyperedge.*

Proof. The algorithm processes each superhyperedge $e \in E$ for each supervertex u extracted from the priority queue Q. For each superhyperedge e containing u, we examine all supervertices $v \in e$, resulting in up to m operations per superhyperedge.

The number of times we extract a supervertex from Q is $O(|V|)$. For each extraction, we may perform $O(m \cdot \deg(u))$ operations, where $\deg(u)$ is the number of superhyperedges containing u.

Assuming $\deg(u) \leq |E|$, the total number of operations is $O(|V| \cdot |E| \cdot m)$. Each priority queue operation (insert or extract) takes $O(\log |V|)$ time.

Therefore, the overall time complexity is $O(|V| \cdot |E| \cdot m \cdot \log |V|)$. However, since $|V| \leq |E| \cdot m$, we can simplify the time complexity to $O(|E| \cdot m \cdot \log |V|)$. □

Theorem 3.36. *The space complexity of the algorithm is $O(|V| + |E| \cdot m)$.*

Proof. The space requirements are as follows:

- Distance array $d(v)$ for each $v \in V$: $O(|V|)$.
- Priority queue Q: at most $O(|V|)$ elements.
- Storage of the superhypergraph structure: $O(|E| \cdot m)$, since each superhyperedge can contain up to m supervertices.

Thus, the total space complexity is $O(|V| + |E| \cdot m)$. □

3.8 Traveling Salesman Problem in a SuperHypergraph

The Traveling Salesman Problem [29, 73, 87] in a SuperHypergraph seeks a minimum-weight tour visiting all supervertices exactly once, considering superhyperedges' weights, and returning to the starting supervertex while ensuring connectivity constraints. The problem considered is described below.

Problem 3.37. Given a weighted superhypergraph SHG $= (V, E)$, the *Traveling Salesman Problem (TSP)* seeks a minimal-weight closed superhyperpath that visits each supervertex $v \in V$ at least once.

Theorem 3.38. *The Traveling Salesman Problem in a superhypergraph is NP-hard.*

Proof. We prove this by reduction from the classical TSP in graphs, which is known to be NP-hard. Given an instance of the TSP in a graph $G = (V_G, E_G)$, we construct a corresponding superhypergraph SHG $= (V, E)$ as follows:

- For each vertex $v \in V_G$, create a supervertex $v \in V$ in the superhypergraph.
- For each edge $(u, v) \in E_G$, create a superhyperedge $e = \{u, v\} \in E$ with the same weight as the edge in G.

In this construction, the superhypergraph essentially mirrors the original graph. Therefore, any solution to the TSP in SHG corresponds to a solution in G, and vice versa.

Since the TSP in graphs is NP-hard, and we can polynomially reduce any instance of the TSP in graphs to an instance in superhypergraphs, it follows that the TSP in superhypergraphs is also NP-hard. □

3.9 Chinese Postman Problem (CPP) in superhypergraph

The Chinese Postman Problem (CPP) [30, 79, 117] in a superhypergraph seeks a minimum-weight closed walk traversing all superhyperedges at least once, ensuring efficiency in traversal. The problem considered is described below.

Problem 3.39. Given a weighted superhypergraph SHG = (V, E), the *Chinese Postman Problem (CPP)* seeks a minimal-weight closed superhyperpath that traverses every superhyperedge at least once.

The algorithm and related theorems for the above problem are presented below.

Algorithm 9: Chinese Postman Problem in SuperHypergraph

Input: A weighted superhypergraph SHG = (V, E)
Output: A minimal-weight closed superhyperpath traversing every superhyperedge at least once

Initialization:
  Construct an incidence multigraph $G = (V', E')$, where:
  - $V' = V$, the vertices of SHG,
  - E' is formed by replacing each superhyperedge $e \in E$ with all edges between pairs of supervertices in e.

Degree Calculation:
  Compute the degree $\deg(v)$ of each vertex $v \in V'$ in G.
  Identify the set $O \subseteq V'$ of vertices with odd degree in G.

Shortest Path Computation:
  Compute the shortest paths between all pairs of vertices in O using the Floyd-Warshall algorithm.

Perfect Matching:
  Find a minimum-weight perfect matching M on O based on the shortest path distances.

Graph Augmentation:
  Augment G by adding the edges from the matching M.

Eulerian Circuit and Transformation:
  Find an Eulerian circuit C in the augmented graph G.
  Transform the Eulerian circuit C back into a superhyperpath in SHG.

return The minimal-weight closed superhyperpath corresponding to the Eulerian circuit C.

Theorem 3.40. *Algorithm 9 finds a minimal-weight closed superhyperpath that traverses every superhyperedge at least once in SHG.*

Proof. The correctness of the algorithm relies on the properties of Eulerian circuits and the transformation between the superhypergraph and the incidence multigraph.

Construction of G: By replacing each superhyperedge e with edges between every pair of supervertices in e, we create a multigraph G that captures the connectivity of SHG.

Eulerian Circuit Existence: In G, we compute the degrees of all vertices. By adding edges (paths) between pairs of vertices with odd degrees to make all degrees even (through the minimum weight perfect matching M), we ensure that the augmented graph G is Eulerian.

Minimum Weight Matching: The matching M is chosen to minimize the additional weight added to G, which corresponds to minimizing the total traversal cost in SHG.

Eulerian Circuit and Superhyperpath Correspondence: An Eulerian circuit in G traverses every edge at least once. Since edges in G correspond to superhyperedges or shortest paths between supervertices in SHG, the circuit can be mapped back to a closed superhyperpath in SHG that traverses every superhyperedge at least once.

Optimality: The algorithm constructs a circuit with minimal total weight because it only adds the minimum necessary edges (with minimal total weight) to make the graph Eulerian. Therefore, the resulting superhyperpath is of minimal weight. □

Theorem 3.41. *The algorithm runs in polynomial time, specifically $O(|V|^3)$.*

Proof. The time complexity of each step is as follows:

- *Constructing G:* Replacing each superhyperedge e with edges between every pair of supervertices in e can be done in $O(|E| \cdot m^2)$, where m is the maximum size of a superhyperedge. Since $m \leq |V|$, this step is $O(|E| \cdot |V|^2)$.
- *Computing Degrees:* Calculated in $O(|V|)$.
- *Identifying Odd-Degree Vertices O:* $O(|V|)$.
- *Computing Shortest Paths:* Using the Floyd-Warshall algorithm(cf. [5, 65]), this takes $O(|V|^3)$.
- *Minimum Weight Perfect Matching(cf. [25]):* Can be found in $O(|V|^3)$ using algorithms such as the Hungarian method.
- *Finding Eulerian Circuit:* Linear in the number of edges, $O(|E'|)$.
- *Transforming Circuit Back to SHG:* $O(|E'|)$.

The dominant terms are the shortest path computation and the matching, both $O(|V|^3)$. Therefore, the overall time complexity is $O(|V|^3)$. □

3.10 Longest Simple Path Problem in SuperHypergraphs

The Longest Simple Path Problem in SuperHypergraphs involves finding a maximum-length path that visits each supervertex at most once, satisfying adjacency constraints defined by the superhyperedges [86]. The problem considered in this subsection is described below.

Problem 3.42. Given a superhypergraph SHG $= (V, E)$ and two supervertices $u, v \in V$, the *Longest Simple Path Problem* seeks the longest superhyperpath connecting u and v, where the path is defined in terms of the number of superhyperedges.

Theorem 3.43. *The Longest Simple Path Problem in SuperHypergraphs is NP-hard.*

Proof. We reduce the classical Longest Path Problem (LPP) in graphs, known to be NP-hard, to the Longest Simple Path Problem in superhypergraphs. Let $G = (V_G, E_G)$ be an instance of LPP. Construct a superhypergraph SHG $= (V, E)$ as follows:

- For each vertex $v \in V_G$, create a supervertex $\{v\} \in V$.
- For each edge $(u, v) \in E_G$, create a superhyperedge $\{\{u\}, \{v\}\} \in E$.

A simple path in G corresponds to a superhyperpath in SHG. Therefore, solving the Longest Simple Path Problem in SHG provides a solution to LPP in G. Since LPP is NP-hard, the Longest Simple Path Problem in superhypergraphs is also NP-hard. □

3.11 Maximum Spanning Tree Problem in SuperHypergraphs

The Maximum Spanning Tree Problem (cf. [47, 76, 81]) in SuperHypergraphs seeks a superhypertree that connects all supervertices while maximizing the total weight of selected superhyperedges, maintaining tree-like structural constraints. The problem considered in this subsection is described below.

Problem 3.44. Given a weighted superhypergraph SHG = (V, E), the *Maximum Spanning Tree Problem* seeks a superhypertree $T = (V, E_T)$ such that:

1. T satisfies the conditions of a superhypertree.
2. The total weight of T, defined as $w(T) = \sum_{e \in E_T} w(e)$, is maximized.

The algorithm and related theorems for the above problem are presented below.

Algorithm 10: Maximum Spanning SuperHypertree Algorithm

Input: A weighted superhypergraph SHG = (V, E)
Output: A Maximum Spanning SuperHypertree $T = (V, E_T)$

```
Initialize E_T ← ∅;
Sort the superhyperedges E in non-increasing order of weight;
Initialize a disjoint-set data structure for vertices in V;
foreach superhyperedge e ∈ E (in sorted order) do
    Compute the union of sets containing vertices in e;
    if adding e does not create a cycle then
        Add e to E_T;
        Update the disjoint-set structure for all vertices in e;
return T = (V, E_T);
```

Theorem 3.45. *The above algorithm correctly computes the Maximum Spanning SuperHypertree for a weighted superhypergraph SHG.*

Proof. The algorithm iteratively adds the heaviest superhyperedge e to E_T while maintaining the superhypertree properties:

- *Acyclicity:* The disjoint-set structure ensures that no cycles are formed.
- *Connectedness:* By construction, the algorithm only terminates when E_T connects all supervertices in V.
- *Optimality:* At each step, the algorithm selects the heaviest available superhyperedge that maintains the superhypertree properties, ensuring the total weight $w(T)$ is maximized.

Thus, the algorithm is correct. □

Theorem 3.46. *The time complexity of the algorithm is $O(|E| \cdot \log |E| + |E| \cdot m \cdot \alpha(|V|))$, where m is the maximum size of a superhyperedge and α is the inverse Ackermann function. The space complexity is $O(|V| + |E|)$.*

Proof. Time Complexity:

- Sorting the superhyperedges takes $O(|E| \cdot \log |E|)$.
- Processing each superhyperedge involves:
  - m Find operations, each taking $O(\alpha(|V|))$.
  - At most one Union operation, taking $O(\alpha(|V|))$.

- Thus, processing all superhyperedges takes $O(|E| \cdot m \cdot \alpha(|V|))$.

The total time complexity is $O(|E| \cdot \log |E| + |E| \cdot m \cdot \alpha(|V|))$.

Space Complexity:

- Storing the disjoint-set structure requires $O(|V|)$.
- Storing the superhyperedges requires $O(|E|)$.

Thus, the space complexity is $O(|V| + |E|)$. □

3.12 Horn satisfiability problem in a Superhypergraph

The Horn Satisfiability Problem determines whether a Boolean formula in conjunctive normal form, with at most one positive literal per clause, is satisfiable(cf. [48]). In this subsection, we explore the Horn Satisfiability Problem in a SuperHyperGraph. The related definitions and an overview of the problem are provided below.

Definition 3.47 (Boolean Variable)**.** (cf. [68]) A *Boolean variable* is a variable that can take one of two possible values: `True` (1) or `False` (0). Formally, a Boolean variable x is defined as:

$$x \in \{0, 1\},$$

where 1 represents `True` and 0 represents `False`.

Definition 3.48 (Satisfiable)**.** A Boolean formula is said to be *satisfiable* if there exists an assignment of `True` (1) or `False` (0) to its Boolean variables such that the entire formula evaluates to `True`. Formally, for a formula $F(x_1, x_2, \ldots, x_n)$ composed of Boolean variables $x_1, x_2, \ldots, x_n$, F is satisfiable if there exists an assignment $A : \{x_1, x_2, \ldots, x_n\} \to \{0, 1\}$ such that:

$$F(A(x_1), A(x_2), \ldots, A(x_n)) = 1.$$

Definition 3.49 (Horn Clause)**.** (cf. [48]) A *Horn clause* is a disjunction of literals with at most one positive literal. Formally, a Horn clause C is of the form:

$$\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_k \vee x_{k+1},$$

where $x_1, x_2, \ldots, x_k, x_{k+1}$ are Boolean variables. The clause is satisfied if at least one literal evaluates to `True`.

Problem 3.50 (Horn Satisfiability Problem in a Superhypergraph)**.** (cf. [48, 80]) Let $H = (V, E)$ be a superhypergraph, where V is a set of supervertices and E is a set of superhyperedges. Each superhyperedge $e \in E$ represents a Horn clause. The *Horn Satisfiability Problem in a Superhypergraph* is to determine whether there exists a satisfying assignment $f : V \to \{\texttt{True}, \texttt{False}\}$ such that every clause (superhyperedge) is satisfied.

We adapt the Unit Propagation algorithm, which is commonly used for Horn formulas, to solve the satisfiability problem in Superhypergraphs.

Algorithm 11: Horn Satisfiability in a Superhypergraph

Input: A superhypergraph $H = (V, E)$, where each $e \in E$ represents a Horn clause.
Output: `True` if the Horn clauses are satisfiable, `False` otherwise.

```
Initialize all vertices v ∈ V as Unknown;
repeat
    foreach hyperedge e ∈ E do
        if all but one literal in e are assigned a value and the remaining literal is unassigned then
            Assign the remaining literal to satisfy the clause;
until no changes occur;
if all clauses are satisfied then
    return True;
else
    return False;
```

Theorem 3.51. *The algorithm correctly determines whether the given Horn clauses represented by the super-hypergraph are satisfiable.*

Proof. The algorithm performs unit propagation, which ensures that:

1. If a Horn clause has all literals but one assigned and the remaining literal is unassigned, the remaining literal is assigned a value to satisfy the clause.
2. This process repeats until no further assignments can be made.

If all clauses are satisfied after this process, the formula is satisfiable, and the algorithm returns `True`. If a state is reached where no assignment can satisfy a clause, the formula is unsatisfiable, and the algorithm returns `False`. □

Theorem 3.52 (Time Complexity)**.** *The worst-case time complexity of the algorithm is* $O(|E|)$.

Proof. In each iteration, the algorithm processes all hyperedges $e \in E$. Since each hyperedge is processed once, the overall complexity is $O(|E|)$. □

Theorem 3.53 (Space Complexity)**.** *The space complexity of the algorithm is* $O(|V|)$.

Proof. The algorithm requires space to store the assignment of truth values for all vertices $v \in V$. Therefore, the space complexity is $O(|V|)$. □

4 Future Tasks of This Research

This section outlines the future tasks related to this research.

Beyond the problems discussed in this paper, numerous classic problems in graph theory and computer science are well-known for standard graphs [50]. Expanding these problems to the framework of superhypergraphs presents an exciting avenue for future exploration.

Additionally, we hope that further investigations will focus on extending superhypergraph problems and algorithms to fuzzy environments [97, 122, 123], neutrosophic environments [37, 101, 102, 112], hypersoft environments [36, 43, 103], and plithogenic environments [38, 40, 44, 45, 104]. These extensions could provide deeper insights and broader applications of superhypergraph theory.

Funding

This research received no external funding.

Acknowledgments

We humbly extend our heartfelt gratitude to everyone who has provided invaluable support, enabling the successful completion of this paper. We also express our sincere appreciation to all readers who have taken the time to engage with this work. Furthermore, we extend our deepest respect and gratitude to the authors of the references cited in this paper. Thank you for your significant contributions.

Data Availability

This paper does not involve any data analysis.

Ethical Approval

This article does not involve any research with human participants or animals.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Disclaimer

This study primarily focuses on theoretical aspects, and its application to practical scenarios has not yet been validated. Future research may involve empirical testing and refinement of the proposed methods. The authors have made every effort to ensure that all references cited in this paper are accurate and appropriately attributed. However, unintentional errors or omissions may occur. The authors bear no legal responsibility for inaccuracies in external sources, and readers are encouraged to verify the information provided in the references independently. Furthermore, the interpretations and opinions expressed in this paper are solely those of the authors and do not necessarily reflect the views of any affiliated institutions.

References

[1] Scott Aaronson and Alex Arkhipov. The computational complexity of linear optics. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 333–342, 2011.

[2] Isolde Adler, Tomáš Gavenčiak, and Tereza Klimošová. Hypertree-depth and minors in hypergraphs. *Theoretical Computer Science*, 463:84–95, 2012.

[3] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, 28(8):2167–2181, 2007.

[4] Hamed Ahmadi and José Ramon Martí. Minimum-loss network reconfiguration: A minimum spanning tree problem. *Sustainable Energy, Grids and Networks*, 1:1–9, 2015.

[5] Asghar Aini and Amir Salehipour. Speeding up the floyd-warshall algorithm for the cycled shortest path problem. *Appl. Math. Lett.*, 25:1–5, 2012.

[6] Md. Tanvir Alam, Chowdhury Farhan Ahmed, Md. Samiullah, and Carson Kai-Sang Leung. Mining frequent patterns from hypergraph databases. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2021.

[7] Eric Allender. Reachability problems: An update. In *Computation and Logic in the Real World: Third Conference on Computability in Europe, CiE 2007, Siena, Italy, June 18-23, 2007. Proceedings 3*, pages 25–27. Springer, 2007.

[8] Eric Allender, Samir Datta, and Sambuddha Roy. The directed planar reachability problem. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science: 25th International Conference, Hyderabad, India, December 15-18, 2005. Proceedings 25*, pages 238–249. Springer, 2005.

[9] Ali Alqazzaz and Karam M Sallam. Evaluation of sustainable waste valorization using treesoft set with neutrosophic sets. *Neutrosophic Sets and Systems*, 65(1):9, 2024.

[10] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[11] M Amin Bahmanian and Mateja Šajna. Hypergraphs: connection and separation. *arXiv preprint arXiv:1504.04274*, 2015.

[12] Mohammad A Bahmanian and Mateja Sajna. Connection and separation in hypergraphs. *Theory and Applications of Graphs*, 2(2):5, 2015.

[13] Claude Berge. *Hypergraphs: combinatorics of finite sets*, volume 45. Elsevier, 1984.

[14] Hans L Bodlaender. A tourist guide through treewidth. *Acta cybernetica*, 11(1-2):1–21, 1993.

[15] Hans L Bodlaender, Michael R Fellows, and Michael T Hallett. Beyond np-completeness for problems of bounded width (extended abstract) hardness for the w hierarchy. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 449–458, 1994.

[16] Hans L Bodlaender and Arie MCA Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[17] Walter S Brainerd. The minimalization of tree automata. *Information and Control*, 13(5):484–491, 1968.

[18] Nicolas Broutin, Luc Devroye, and Erin McLeish. Note on the structure of kruskal's algorithm. *Algorithmica*, 56:141–159, 2010.

[19] Robin L. P. Chang and Theodosios Pavlidis. Fuzzy decision tree algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 7:28–35, 1977.

[20] Yi-lai Chen, Tao Wang, Ben-sheng Wang, and Zhou-jun Li. A survey of fuzzy decision tree classifier. *Fuzzy Information and Engineering*, 1:149–159, 2009.

[21] Eli Chien, Chao Pan, Jianhao Peng, and Olgica Milenkovic. You are allset: A multiset function framework for hypergraph neural networks. *ArXiv*, abs/2106.13264, 2021.

[22] Yusuf Civan and Demet Taylan. Coloring hypercomplete and hyperpath graphs. *Turkish Journal of Mathematics*, 38(1):1–15, 2014.

[23] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2008.

[24] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[25] Ulrich Derigs. A generalized hungarian method for solving minimum weight perfect matching problems with algebraic objective. *Discret. Appl. Math.*, 1:167–180, 1979.

[26] Reinhard Diestel. Graduate texts in mathematics: Graph theory.

[27] Reinhard Diestel. Graph theory 3rd ed. *Graduate texts in mathematics*, 173(33):12, 2005.

[28] Reinhard Diestel. *Graph theory*. Springer (print edition); Reinhard Diestel (eBooks), 2024.

[29] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans. Evol. Comput.*, 1:53–66, 1997.

[30] Horst A. Eiselt, Michel Gendreau, and Gilbert Laporte. Arc routing problems, part i: The chinese postman problem. *Oper. Res.*, 43:231–242, 1995.

[31] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

[32] Yifan Feng, Haoxuan You, Zizhao Zhang, R. Ji, and Yue Gao. Hypergraph neural networks. In *AAAI Conference on Artificial Intelligence*, 2018.

[33] Carlos Eduardo Ferreira, Alexander Martin, Cid C. de Souza, Robert Weismantel, and Laurence A. Wolsey. The node capacitated graph partitioning problem: A computational study. *Mathematical Programming*, 81:229–256, 1998.

[34] Eldar Fischer, Arie Matsliah, and Asaf Shapira. Approximate hypergraph partitioning and applications. *SIAM Journal on Computing*, 39(7):3155–3185, 2010.

[35] Ronald C. Freiwald. An introduction to set theory and topology. 2014.

[36] Takaaki Fujita. Note for hypersoft filter and fuzzy hypersoft filter. *Multicriteria Algorithms With Applications*, 5:32–51, 2024.

[37] Takaaki Fujita. Note for neutrosophic incidence and threshold graph. *SciNexuses*, 1:97–125, 2024.

[38] Takaaki Fujita. A review of the hierarchy of plithogenic, neutrosophic, and fuzzy graphs: Survey and applications. *ResearchGate(Preprint)*, 2024.

[39] Takaaki Fujita. Short note of supertree-width and n-superhypertree-width. *Neutrosophic Sets and Systems*, 77:54–78, 2024.

[40] Takaaki Fujita. *Advancing Uncertain Combinatorics through Graphization, Hyperization, and Uncertainization: Fuzzy, Neutrosophic, Soft, Rough, and Beyond*. Biblio Publishing, 2025.

[41] Takaaki Fujita. A comprehensive discussion on fuzzy hypersoft expert, superhypersoft, and indetermsoft graphs. *Neutrosophic Sets and Systems*, 77:241–263, 2025.

[42] Takaaki Fujita and Florentin Smarandache. A concise study of some superhypergraph classes. *Neutrosophic Sets and Systems*, 77:548–593, 2024.

[43] Takaaki Fujita and Florentin Smarandache. A short note for hypersoft rough graphs. *HyperSoft Set Methods in Engineering*, 3:1–25, 2024.

[44] Takaaki Fujita and Florentin Smarandache. Study for general plithogenic soft expert graphs. *Plithogenic Logic and Computation*, 2:107–121, 2024.

[45] Takaaki Fujita and Florentin Smarandache. Uncertain automata and uncertain graph grammar. *Neutrosophic Sets and Systems*, 74:128–191, 2024.

[46] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[47] Giulia Galbiati, Angelo Morzenti, and Francesco Maffioli. On the approximability of some maximum spanning tree problems. *Theoretical Computer Science*, 181(1):107–118, 1997.

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

[49] Yue Gao, Yifan Feng, Shuyi Ji, and Rongrong Ji. Hgnn+: General hypergraph neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45:3181–3199, 2022.

[50] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[51] Masoud Ghods, Zahra Rostami, and Florentin Smarandache. Introduction to neutrosophic restricted superhypergraphs and neutrosophic restricted superhypertrees and several of their properties. *Neutrosophic Sets and Systems*, 50:480–487, 2022.

[52] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 57–74, 2016.

[53] Georg Gottlob, Gianluigi Greco, Francesco Scarcello, et al. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1:20, 2014.

[54] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32, 1999.

[55] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: A survey. In *Mathematical Foundations of Computer Science 2001: 26th International Symposium, MFCS 2001 Mariánské Lázne, Czech Republic, August 27–31, 2001 Proceedings 26*, pages 37–57. Springer, 2001.

[56] Georg Gottlob and Reinhard Pichler. Hypergraphs in model checking: Acyclicity and hypertree-width versus clique-width. *SIAM Journal on Computing*, 33(2):351–378, 2004.

[57] Ronald L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7:43–57, 1985.

[58] Dae Geun Ha, Tae Wook Ha, Junghyuk Seo, and Myoung-Ho Kim. Index-based searching for isomorphic subgraphs in hypergraph databases. *Journal of KIISE*, 2019.

[59] Mohammad Hamidi, Florentin Smarandache, and Elham Davneshvar. Spectrum of superhypergraphs via flows. *Journal of Mathematics*, 2022(1):9158912, 2022.

[60] Mohammad Hamidi and Mohadeseh Taghinezhad. *Application of Superhypergraphs-Based Domination Number in Real World*. Infinite Study, 2023.

[61] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[62] Felix Hausdorff. *Set theory*, volume 119. American Mathematical Soc., 2021.

[63] Dorit S Hochba. Approximation algorithms for np-hard problems. *ACM Sigact News*, 28(2):40–52, 1997.

[64] Seok-Hee Hong. Algorithms for 1-planar graphs. In *Beyond Planar Graphs: Communications of NII Shonan Meetings*, pages 69–87. Springer, 2020.

[65] Stefan Hougardy. The floyd-warshall algorithm on graphs with negative cycles. *Inf. Process. Lett.*, 110:279–281, 2010.

[66] Karel Hrbacek and Thomas Jech. Introduction to set theory, revised and expanded. 2017.

[67] Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.

[68] Jeff Kahn, Gil Kalai, and Nathan Linial. *The influence of variables on Boolean functions*. Institute for Mathematical Studies in the Social Sciences, 1989.

[69] George Karypis. Multilevel hypergraph partitioning. In *Multilevel Optimization in VLSICAD*, pages 125–154. Springer, 2003.

[70] Spencer Krieger and John Kececioglu. Shortest hyperpaths in directed hypergraphs for reaction pathway inference. *Journal of Computational Biology*, 30(11):1198–1225, 2023.

[71] G. Laszewski. Intelligent structural operators for the k-way graph partitioning problem. In *International Conference on Genetic Algorithms*, 1991.

[72] Azriel Levy. *Basic set theory*. Courier Corporation, 2012.

[73] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21:498–516, 1973.

[74] David López and Angélica Lozano. Shortest hyperpaths in a multimodal hypergraph with real-time information on some transit lines. *Transportation Research Part A: Policy and Practice*, 137:541–559, 2020.

[75] Dániel Marx. Approximating fractional hypertree width. *ACM Transactions on Algorithms (TALG)*, 6(2):1–17, 2010.

[76] Sugama Maskar. Maximum spanning tree graph model: National examination data analysis of junior high school in lampung province. In *Proceeding International Conference on Science and Engineering*, volume 3, pages 375–378, 2020.

[77] Ulrich Meyer and Peter Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *Embedded Systems and Applications*, 1998.

[78] Zoltán Miklós. *Understanding Tractable Decompositions for Constraint Satisfaction.* PhD thesis, University of Oxford, 2008.

[79] Edward Minieka. The chinese postman problem for mixed networks. *Management Science*, 25:643–648, 1979.

[80] Michel Minoux. The unique horn-satisfiability problem and quadratic boolean equations. *Annals of Mathematics and Artificial Intelligence*, 6:253–266, 1992.

[81] Clyde Monma, Mike Paterson, Subhash Suri, and Frances Yao. Computing euclidean maximum spanning trees. In *Proceedings of the fourth annual symposium on Computational geometry*, pages 241–251, 1988.

[82] Takao Nishizeki and Norishige Chiba. *Planar graphs: Theory and algorithms*. Elsevier, 1988.

[83] Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, pages 285–298, 2012.

[84] James B. Orlin, Kamesh Madduri, K. Subramani, and Matthew D. Williamson. A faster algorithm for the single source shortest path problem with few distinct positive lengths. *J. Discrete Algorithms*, 8:189–198, 2010.

[85] Christos H Papadimitriou. Computational complexity. In *Encyclopedia of computer science*, pages 260–265. 2003.

[86] Quang Dung Pham and Yves Deville. Solving the longest simple path problem with constraint-based techniques. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*, pages 292–306. Springer, 2012.

[87] Petrica C. Pop, Ovidiu Cosma, Cosmin Sabo, and Corina Pop Sitar. A comprehensive survey on the generalized traveling salesman problem. *Eur. J. Oper. Res.*, 314:819–835, 2023.

[88] Gomathi Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21:267–305, 1996.

[89] Lisbeth J Reales-Chacón, María E Lucena de Ustáriz, Francisco J Ustáriz-Fajardo, Andrea C Peñafiel Luna, Gabriela J Bonilla-Ayala, Pablo Djabayan-Djibeyan, José L Erazo-Parra, and Mónica A Valdiviezo-Maygua. Study of the efficacy of neural mobilizations to improve sensory and functional responses of lower extremities in older adults with diabetic peripheral neuropathy using plithogenic n-superhypergraphs. *Neutrosophic Sets and Systems*, 74:1–12, 2024.

[90] Neil Robertson and Paul D. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[91] Neil Robertson and Paul D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[92] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[93] Neil Robertson and Paul D Seymour. Graph minors. v. excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 41(1):92–114, 1986.

[94] Neil Robertson and Paul D Seymour. Graph minors. iv. tree-width and well-quasi-ordering. *Journal of Combinatorial Theory, Series B*, 48(2):227–254, 1990.

[95] Neil Robertson and Paul D Seymour. Graph minors. viii. a kuratowski theorem for general surfaces. *Journal of Combinatorial Theory, Series B*, 48(2):255–288, 1990.

[96] Neil Robertson and Paul D. Seymour. Graph minors. x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.

[97] Azriel Rosenfeld. Fuzzy graphs. In *Fuzzy sets and their applications to cognitive and decision processes*, pages 77–95. Elsevier, 1975.

[98] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics*, 27:1–39, 2023.

[99] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

[100] Arunabha Sen, Haiyong Deng, and Sumanta Guha. On a graph partition problem with application to vlsi layout. *Inf. Process. Lett.*, 43:87–94, 1992.

[101] Florentin Smarandache. A unifying field in logics: Neutrosophic logic. In *Philosophy*, pages 1–141. American Research Press, 1999.

[102] Florentin Smarandache. Neutrosophic set-a generalization of the intuitionistic fuzzy set. *International journal of pure and applied mathematics*, 24(3):287, 2005.

[103] Florentin Smarandache. Extension of soft set to hypersoft set, and then to plithogenic hypersoft set. *Neutrosophic sets and systems*, 22(1):168–170, 2018.

[104] Florentin Smarandache. *Plithogenic set, an extension of crisp, fuzzy, intuitionistic fuzzy, and neutrosophic sets-revisited*. Infinite study, 2018.

[105] Florentin Smarandache. n-superhypergraph and plithogenic n-superhypergraph. *Nidus Idearum*, 7:107–113, 2019.

[106] Florentin Smarandache. *Extension of HyperGraph to n-SuperHyperGraph and to Plithogenic n-SuperHyperGraph, and Extension of HyperAlgebra to n-ary (Classical-/Neutro-/Anti-) HyperAlgebra*. Infinite Study, 2020.

[107] Florentin Smarandache. *Introduction to the n-SuperHyperGraph-the most general form of graph today*. Infinite Study, 2022.

[108] Florentin Smarandache. Decision making based on valued fuzzy superhypergraphs. 2023.

[109] Florentin Smarandache. *New types of soft sets "hypersoft set, indetermsoft set, indetermhypersoft set, and treesoft set": an improved version*. Infinite Study, 2023.

[110] Florentin Smarandache. *SuperHyperFunction, SuperHyperStructure, Neutrosophic SuperHyperFunction and Neutrosophic SuperHyperStructure: Current understanding and future directions*. Infinite Study, 2023.

[111] Florentin Smarandache. Foundation of superhyperstructure & neutrosophic superhyperstructure. *Neutrosophic Sets and Systems*, 63(1):21, 2024.

[112] Florentin Smarandache. Short introduction to standard and nonstandard neutrosophic set and logic. *Neutrosophic Sets and Systems*, 77:395–404, 2025.

[113] Justin Sybrandt, Ruslan Shaydulin, and Ilya Safro. Hypergraph partitioning with embeddings. *IEEE Transactions on Knowledge and Data Engineering*, 34(6):2771–2782, 2020.

[114] Robert E Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579, 1984.

[115] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[116] Lev Telyatnikov, Maria Sofia Bucarelli, Guillermo Bernardez, Olga Zaghen, Simone Scardapane, and Pietro Lió. Hypergraph neural networks through the lens of message passing: A common perspective to homophily and architecture design. *ArXiv*, abs/2310.07684, 2023.

[117] Harold W. Thimbleby. The directed chinese postman problem. *Software: Practice and Experience*, 33, 2003.

[118] Ioan Tomescu and Marius Zimand. Minimum spanning hypertrees. *Discret. Appl. Math.*, 54:67–76, 1994.

[119] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[120] Gerhard J Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization-Eureka, You Shrink! Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001 Revised Papers*, pages 185–207. Springer, 2003.

[121] Nikola Yolov. Minor-matching hypertree width. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 219–233. SIAM, 2018.

[122] Lotfi A Zadeh. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.