

Algorithms for Optimizing Acyclic Queries

Zheng Luo

UCLA

luo@cs.ucla.edu

Wim Van den Broeck

University of Bergen

wim.broeck@uib.no

Guy Van den Broeck

UCLA

guyvdb@cs.ucla.edu

Yisu Remy Wang

UCLA

remywang@cs.ucla.edu

Abstract

Most research on query optimization has centered on binary join algorithms like hash join and sort-merge join. However, recent years have seen growing interest in theoretically optimal algorithms, notably Yannakakis' algorithm. These algorithms rely on *join trees*, which differ from the operator trees for binary joins and require new optimization techniques. We propose three approaches to constructing join trees for acyclic queries. First, we give an algorithm to enumerate all join trees of an α -acyclic query *by edits* in linear time with amortized constant delay, which forms the basis of a cost-based optimizer for acyclic joins. Second, we show the Maximum Cardinality Search algorithm by Tarjan and Yannakakis constructs the unique *shallowest* join tree for any Berge-acyclic query, thus enabling parallel execution of large join queries. Finally, we prove that a simple algorithm by Hu et al. converts any connected left-deep linear plan of a γ -acyclic query into a join tree, allowing reuse of optimizers developed for binary joins.

1 Introduction

The query optimizer sits at the heart of a database system. It takes a query as input and generates a plan for efficient execution, allowing the user to program declaratively without worrying about performance. Among the many relational algebra operators, join has received significant attention in optimization research. Its *compositional* nature allows for combining information from multiple relations, constructing complex queries from simple ones, and producing an output asymptotically larger than the inputs. The primary challenge is the *join ordering problem* to find the best arrangement of many join operations. Most existing research has focused on binary join algorithms such as hash join and sort-merge join, but these can produce unnecessarily large intermediates. Recent work has revived interest in optimal join algorithms, notably Yannakakis' instance-optimal algorithm [45] for acyclic queries, which runs in linear time in the input and output size, $\mathcal{O}(|\text{IN}| + |\text{OUT}|)$. Its execution is guided by *join trees* whose nodes are relations, different from traditional binary join plans with relations at the leaves and join operators at the internal nodes. Although the algorithm is optimal regardless of the join tree, the choice of plan can affect practical performance. In this paper, we study the optimization problem in the context of Yannakakis-style algorithms.

A query optimizer typically has two parts: a *plan generator* and a *cost model* to assess each plan. This paper focuses on plan generation and presents three approaches:

- Given the line graph L (Definition 6) of an α -acyclic query H , Algorithm 2 enumerates all join trees *by edits*¹ with amortized constant delay, i.e., in $\mathcal{O}(|L| + k)$ where k is the number of trees generated (Theorem 27). If the query is γ -acyclic, Algorithm 4 further reduces the total time complexity to $\mathcal{O}(|H| + k)$ (Theorem 28).
- Given a Berge-acyclic query, we prove that the classic Maximum Cardinality Search algorithm by Tarjan and Yannakakis [41] yields a unique *shallowest* join tree (Theorem 35), enabling parallel execution of very large queries.
- Given a connected left-deep linear join plan for any γ -acyclic query, we prove that a simple algorithm by Hu et al. [22] always converts the plan into a valid join tree (Corollary 39), allowing reuse of existing optimizers.

¹To avoid redundant work, enumeration *by edits* [24] outputs the difference between consecutive elements.

The rest of the paper is organized as follows: Section 2 discusses related work; Section 3 introduces relevant concepts and notations; Section 4 presents the join tree enumeration algorithms; Section 5 introduces the unique shallowest join tree, namely the canonical join tree and its construction; Section 6 discusses the conversion from binary plans to join trees; Section 7 concludes the paper by pointing to avenues for future work. Additional technical details, proofs and a notation table (Table 2) can be found in the appendix.

2 Related Work

Join order optimization is well studied, with algorithms based on dynamic programming (DP) from the bottom up [30, 36, 32], cost-based rewriting from the top down [13, 17], greedy heuristics [6, 16, 40], and randomized search [38]. Since the plan space is exponential, most methods prune it: some restrict to left-deep plans [23, 25], while others avoid Cartesian products [30, 31]. Our algorithms restrict the query plans to those running in linear time for acyclic queries. In particular, while avoiding Cartesian products requires each subplan to form a *spanning tree* of the corresponding subquery’s join graph,² our algorithms find *maximum spanning trees* of the *weighted* join graph. Several algorithms for ordering binary joins are based on dynamic programming and tabulate shared structures among different plans [30, 31]. This is desirable because subplans are grouped into equivalence classes, and the optimal plan can be constructed in a bottom-up manner. Future work may explore constructing compact representations of join trees, suitable for dynamic programming.

Several recent papers have proposed practical implementations of Yannakakis’ algorithm for acyclic queries [2, 22, 43, 48]. For example, Zhao et al. [48] find that different query plans perform similarly, thanks to the optimality of Yannakakis’ algorithm. They adopt a simple heuristic to construct the join tree by picking the largest input relation as the root, and then greedily attaching the remaining relations into the tree. Inspired by this algorithm, we prove in Section 5 that for Berge-acyclic queries there is a unique *shallowest* join tree for any given root where the depth of each node is minimized. Furthermore, this tree can be constructed in linear time by Tarjan and Yannakakis’ Maximum Cardinality Search algorithm [41]. Shallow trees are desirable for parallel execution, where the depth of the tree determines the number of sequential steps. Other practical implementations of Yannakakis’ algorithm leverage existing optimizers for binary joins and convert a binary plan into a join tree [2, 22]. In particular, Hu et al. [22] find that every left-deep linear plan encountered in practice can be converted into a join tree by a simple algorithm. This is not surprising, as we will prove in Section 6 that every connected left-deep linear plan of a γ -acyclic query must traverse some join tree from root to leaves.

On the theoretical side, attention has been focused on finding *(hyper-)tree decompositions* to improve the asymptotic complexity of query processing [39, 21, 18]. The general goal is to find a decomposition with small *width* which can be used to guide the execution of join algorithms. Most algorithms find a single decomposition with minimum width to achieve the optimal asymptotic complexity [39, 21, 18]. Nevertheless, different decompositions with the same width may still lead to different performance in practice, and cost-based optimization remains crucial. For this, Carmeli et al. [7] propose an algorithm to enumerate tree decompositions with polynomial delay. In this paper, we focus on acyclic queries and their join trees, which are precisely the decompositions with width 1. Our enumeration algorithm can generate all join trees by edits with amortized constant delay.

3 Preliminaries

We focus on full conjunctive queries [1] in this paper and identify each query with its hypergraph, where each vertex represents a variable and each hyperedge represents a relation.

Definition 1 (Hypergraph). A hypergraph $H = (X, R, \chi)$ consists of a set of vertices X , a set of hyperedges R and an incidence function $\chi : R \rightarrow \mathcal{P}(X)$.

We only consider hypergraphs without isolated vertices, empty hyperedges or duplicated hyperedges over the same set of vertices. We assume each hyperedge r contains a bounded number ($\mathcal{O}(1)$) of vertices

²The join graph of a query has a vertex for each relation and an edge for each pair of relations that join with each other. We later define this as the line graph of the query hypergraph in Definition 6.

exclusive to r . When there is no ambiguity, we will use r interchangeably with $\chi(r)$, $x \in r$ with $x \in \chi(r)$, and $H(X, R)$ with $H(X, R, \chi)$. We will also apply common set operations directly to hyperedges, e.g., $r_1 \cap r_2$ for $\chi(r_1) \cap \chi(r_2)$. In fact, the reader can often ignore χ and identify a hyperedge with its set of vertices; the purpose of χ is to relate hyperedges over different vertex sets across hypergraphs in the correctness proofs of our algorithms. We write $X(H)$ and $R(H)$ to refer to the vertices and edges of H respectively. The size of a hypergraph is the total size of all hyperedges $|H| = \sum_{r \in R} |r|$. We let $H|_x$ denote the *neighborhood* of $x \in X$, consisting of all hyperedges containing x .

Definition 2 (Multigraph). A multigraph $G = (R, E, \rho)$ consists of a set of vertices R , a set of edges E , and the incidence function $\rho : E \rightarrow \mathcal{P}(R)$ such that $1 \leq |\rho(e)| \leq 2$ for any $e \in E$. An edge e is a self-loop if $|\rho(e)| = 1$. Edges $e_1, e_2 \in E$ are parallel if $\rho(e_1) = \rho(e_2)$.

We denote the set of vertices in G with R , because we will soon define the *line graph* L of a hypergraph where each vertex in L represents a hyperedge. We write $R(G)$ and $E(G)$ to refer to the vertices and edges of G respectively.

Definition 3 (Simple Graph). A simple graph is a multigraph with no parallel edges or self-loops, i.e., the incidence function is injective and always returns two distinct vertices.

Because each edge e in a simple graph can be identified with its two endpoints $\rho(e) = \{r_1, r_2\}$, we will omit ρ and write $\{r_1, r_2\}$ for e when there is no ambiguity.

Definition 4 (Cycle, Clique and Diamond). In a simple graph $G = (R, E)$, a sequence of $n \geq 3$ distinct vertices (v_0, \dots, v_{n-1}) where $v_i \in R$ is

- a cycle if there is an edge between v_i and $v_{(i+1) \bmod n}$ for all $0 \leq i \leq n-1$;
- an n -clique K_n if the induced graph $G|_{\{v_0, \dots, v_n\}}$ has an edge between each pair of vertices;
- a diamond if $n = 4$ and $G|_{\{v_0, \dots, v_3\}}$ has one fewer edge than a K_4 .

A weighted graph is a (multi-)graph where each edge is assigned a weight:

Definition 5 (Weighted Graph). A weighted graph (G, ω) consists of a multigraph $G = (R, E, \rho)$ and weight function $\omega : E \rightarrow \mathbb{N}$ assigning a natural number to each edge in G .

The line graph of a hypergraph is the “intersection graph” of its hyperedges:

Definition 6 (Line Graph). For a hypergraph H , the line graph $L(H) = (G, \chi)$ consists of a simple graph $G = (R, E)$ and an edge labeling function $\chi : E \rightarrow \mathcal{P}(X)$. The vertices of G are the hyperedges R of H , and there is an edge $e = \{r_1, r_2\} \in E$ when $r_1 \cap r_2 \neq \emptyset$. The edge labeling function maps each edge e to the intersection of the hyperedges represented by the endpoints of e : $\chi(\{r_1, r_2\}) = r_1 \cap r_2$.

In addition, we define a weight function mapping each edge to the size of its label:

Definition 7 (Line Graph Edge Weight). For each edge $e \in E(L(H))$, the weight function $\omega : E(L(H)) \rightarrow \mathbb{N}$ is defined as $\omega(e) = |\chi(e)|$.

When there is no ambiguity from the context, we simply write $L(H)$ as L , and we may treat the line graph (G, χ) as a weighted graph (G, ω) . The size of a line graph is the sum of all edge weights $|L| = \sum_{e \in E(L)} \omega(e)$. For any subgraph G of $L(H)$, we let $G|_x$ denote the subgraph of G induced by $H|_x$.

Example 8. Figure 1a shows an example hypergraph of size 14. The line graph $L_6 = L(H_6)$ is shown in Figure 1b. For instance, hyperedges Y and U share two vertices a and d , so they are connected by an edge in the line graph. The edge weight is given by $\omega(\{Y, U\}) = |\{a, d\}| = 2$. Each unannotated edge in L_6 connects two hyperedges sharing only the vertex a . Otherwise, the common vertices shared by a pair of hyperedges are annotated next to the corresponding edge. The size of the line graph is $|L_6| = 19$.

Definition 9 (Join Tree). A join tree T of hypergraph H is a spanning tree of $L(H)$ such that $T|_x$ is a connected subtree for each $x \in X(H)$. If a certain vertex is specified as the root, T becomes a rooted join tree.

Algorithm 1: MCS from $r \in R(H)$

Input: α -acyclic hypergraph H and root
 r

Output: Join tree T_r rooted at r

```

1  $p(r) \leftarrow \text{null}; E(T_r) \leftarrow \emptyset$ 
2 while  $R \neq \emptyset$  do
3   for  $x \in r \cap X$  do
4     for  $r' \in R : x \in r'$  do
5        $p(r') \leftarrow r$ 
6    $X \leftarrow X \setminus r$ 
7    $R \leftarrow R \setminus \{r\}$ 
8    $E(T_r) \leftarrow E(T_r) \cup \{\{r, p(r)\}\}$ 
9    $r \leftarrow \arg \max_{r \in R} |r \setminus X|$ 
10 return  $T_r$ 

```

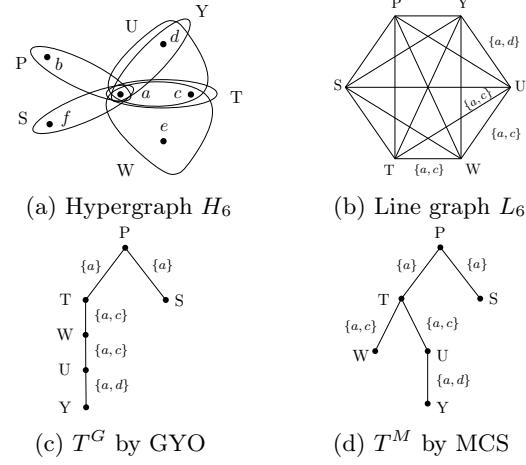


Figure 1: A hypergraph H_6 , its line graph L_6 , and two join trees T^G and T^M .

We use $R(T)$ to denote the set of nodes and $E(T)$ to denote the set of edges of tree T . The requirement that $T|_{x \in X(H)}$ be a connected subtree is also known as the *running intersection property* [15]. We write $\mathcal{T}(L(H))$ or $\mathcal{T}(H)$ to denote the set of unrooted join trees of H . We use T_r to denote a join tree rooted at $r \in R(T)$. When there is no ambiguity, we simply write T . The depth of a node r_i in the rooted tree T_r , denoted $d(T_r, r_i)$, is defined as its distance from the root.

Four common notions of hypergraph acyclicity are defined in increasing order of strictness [15], namely α -acyclic \supseteq β -acyclic \supseteq γ -acyclic \supseteq Berge-acyclic.

Definition 10 (Hypergraph Acyclicity). *A hypergraph H is:*

- α -acyclic if it admits a join tree as in Definition 9;
- β -acyclic if every subgraph of H is α -acyclic;
- γ -acyclic if it does not contain any γ cycle. A γ cycle is a sequence of length $k \geq 3$ of distinct hyperedges and distinct vertices $(r_0, x_0, \dots, r_{k-1}, x_{k-1})$ such that every $x_{i \in [0, k-2]}$ belongs to $r_i \cap r_{i+1}$ and no other r_j while x_{k-1} belongs to $r_{k-1} \cap r_0$ and possibly other hyperedges;
- Berge-acyclic if it does not contain any Berge cycle. A Berge cycle is a sequence of length $k \geq 2$ of distinct vertices and distinct hyperedges $(r_0, x_0, \dots, r_{k-1}, x_{k-1})$ such that $\forall i \in [k] : x_i \in r_i \cap r_{(i+1) \bmod k}$.

The following classic result relates join trees of H to maximum spanning trees of $L(H)$:

Theorem 11 (Maier [29]). *Given an α -acyclic hypergraph $H = (X, R)$, a tree with nodes in R is a join tree of H if and only if it is a maximum spanning tree of $L(H)$.*

Join trees for acyclic hypergraphs can be constructed by a procedure called *GYO reduction*.

Definition 12. *A GYO reduction order is a sequence of hyperedges r_1, r_2, \dots, r_k such that for each $r_{i < k}$, there is some $r_{p > i}$, called the parent of r_i , such that $\forall r_{j > i} : r_i \cap r_j \subseteq r_p$.*

The GYO reduction algorithm [46] finds such an order iteratively, and attaches each hyperedge to its parent to form a join tree. It generates the join tree T^G as shown in Figure 1c, where, for example, T is the parent of W , denoted as $p(W) = T$ and the root P has no parent.

Another way to construct join trees is via the Maximum Cardinality Search (MCS) algorithm [41], and we present a simplified version in Algorithm 1. The join tree produced by the algorithm is called an MCS tree. Given an input hypergraph H and a hyperedge $r \in R(H)$ chosen as the root, Algorithm 1 constructs a rooted MCS tree by iteratively expanding toward hyperedges connected with those already processed. The algorithm maintains a set R of unprocessed hyperedges and a set X of unvisited vertices. Initially, the MCS

tree contains no edges, and the first hyperedge to be processed is the root r . In each iteration, the algorithm processes the current hyperedge r by examining all of its unvisited vertices in $r \cap X$. Whenever such a vertex x is encountered, every remaining hyperedge $r' \in R$ that contains x records r as its tentative parent $p(r') \leftarrow r$. Once all vertices of r are marked as visited, they are removed from X . The hyperedge r is marked as processed and removed from R . After processing each r , the edge between r and its recorded parent $p(r)$ is added to the MCS tree as $\{r, p(r)\}$. The algorithm selects as the next hyperedge an element of R that maximizes the number of already visited vertices, namely a hyperedge r that maximizes $|r \setminus X|$, breaking ties arbitrarily. This process continues until all hyperedges have been processed, when the recorded edges $\{r, p(r)\}$ form an MCS tree rooted at r .

Example 13. On H_6 in Figure 1a with $r = P$, Algorithm 1 examines all vertices of P . P also becomes the tentative parent of five hyperedges, $p(S) = p(T) = p(U) = p(W) = p(Y) \leftarrow P$. It proceeds to check the next hyperedge with the most visited vertices as in Line 9. At this point, each of the five has one visited vertex, Algorithm 1 breaks the tie arbitrarily, say proceeding with S , and examining all its vertices. Its parent is finalized as $p(S) = P$. Each of the remaining four again has one visited vertex, so Algorithm 1 proceeds to check T . Its parent is finalized as $p(T) = P$. While checking T , Algorithm 1 examines the vertex c that is also in U, W , their parents are updated to $p(U) = p(W) \leftarrow T$. It continues until all hyperedges are processed. By connecting each hyperedge with its parent, Algorithm 1 constructs the MCS tree T^M in Figure 1d.

Algorithm 1 can generate different join trees by varying the choice of root and tie-breaking strategy. However, it does not guarantee to generate all possible trees. For example, it never generates the join tree T^G shown in Figure 1c.

We extend the notions of parent, children and siblings to edges of a rooted tree. Let $e \in E(T)$ be an edge of tree T rooted at r (and e not incident to r). Its parent $p(e) \in E(T)$ is the unique tree edge incident to e and lying closer to the root r . We let $c(e)$ denote the set of children of e in T . The siblings of e is the set of edges incident to the parent of e , namely $s(e) = \{e' \in E(T) \setminus \{e\} \mid p(e') = p(e)\}$. Edges incident to the root do not have a parent and are all siblings.

Lemma 14 shows that an MCS tree is “somewhat monotonic” in that every edge must contain some variable not in its parent. If two edges share any variable not in their parents, they must be siblings.

Lemma 14. *Let T be an MCS tree. For an edge $e \in T$ that has a parent, then*

1. $e \not\subseteq p(e)$
2. For another edge $e' \in T : (e \setminus p(e)) \cap (e' \setminus p(e')) \neq \emptyset \implies p(e) = p(e')$.

Computation model. Throughout the paper we assume the Random Access Machine model of computation, where one can allocate an array of size n in $\mathcal{O}(n)$ time. Constant-time operations include accessing and updating an array element, adding or deleting an element in a linked list, and the common arithmetic operations on integers.

4 Enumerating Join Trees

Our strategy for enumerating the join trees of a hypergraph H starts from Theorem 11 which allows us to reduce the problem to the enumeration of maximum spanning trees of the line graph $L(H)$. The best known algorithm for MST enumeration³ is due to Eppstein [14] by deriving from the input graph G a so-called *equivalent graph* G^\equiv . Every spanning tree of G^\equiv corresponds to an MST of G and vice versa. Eppstein gives an algorithm (and proves a matching lower bound) in time $\mathcal{O}(m + n \log n)$ to construct the equivalent graph from an arbitrary weighted graph with m edges and n vertices. Then to enumerate all k MSTs of G , he applies existing algorithms to enumerate the spanning trees of G^\equiv . Since there are optimal spanning tree enumeration algorithms that run in $\mathcal{O}(m + n + k)$ time [24, 37], the overall time complexity to enumerate MSTs is $\mathcal{O}(m + n \log n + k)$. The main result of this section is an algorithm for enumerating join trees leveraging the structure of acyclic hypergraphs and their line graphs. In particular:

³Eppstein focuses on minimum spanning trees, but the same algorithm applies to maximum spanning trees with flipped comparisons.

- Given the line graph L of an α -acyclic hypergraph, we can construct an equivalent graph of L in $\mathcal{O}(|L|)$, thus enumerating the join trees in $\mathcal{O}(|L| + k)$ time (Theorem 27).
- Given any γ -acyclic hypergraph H , we can construct an equivalent graph of $L(H)$ in $\mathcal{O}(|H|)$, lowering the overall time complexity of enumeration to $\mathcal{O}(|H| + k)$ (Theorem 28).

Note that $|L(H)|$ can be quadratic in $|H|$ while $|H| \in \mathcal{O}(|L(H)|)$, so the second item above yields a better bound.

In the rest of this section, we first define what is an equivalent graph. Then, we present the algorithm for enumerating join trees of α -acyclic hypergraphs. Finally, we adapt the algorithm to enumerate join trees of γ -acyclic hypergraphs more efficiently.

4.1 Equivalent Graph

The key idea of Eppstein's algorithm [14] is to construct an *equivalent graph* G^\equiv whose spanning trees one-to-one correspond to the MSTs of the input graph G , thereby reducing MST enumeration to spanning tree enumeration.

Definition 15 (Equivalent Graph [14]). *Given a weighted graph $G = ((R, E, \rho), \omega)$, a multigraph $G^\equiv = ((R, E, \rho^\equiv), \omega)$ is an equivalent graph of G if the spanning trees of G^\equiv one-to-one correspond to the maximum spanning trees of G : a set of edges $E^T \subseteq E$ induces a spanning tree of G^\equiv if and only if E^T induces an MST of G .*

In what follows, we may use the notation $G^\equiv(L(H))$ or $G^\equiv(H)$ to refer to an equivalent graph of the line graph of hypergraph H . Note that G^\equiv shares the same vertices and edges of G , and they only differ in the incidence function mapping each edge to its endpoints. Such an equivalent graph can be constructed from G by applying a series of *sliding transformations*: for two incident edges e, e' with $\rho(e) = \{r_1, r_2\}$ and $\rho(e') = \{r_2, r_3\}$, we can *slide* e' along e by updating $\rho(e') = \{r_1, r_3\}$, if $\omega(e') < \omega(e)$. Note that the edge e' retains its identity after sliding, which explains the need for the incidence function ρ . Furthermore, we will use a given rooted MST T as a guide, and only slide edges towards the root. More formally:

Definition 16 (Sliding Transformation [14]). *Let T be a MST of G rooted at r . Let e be an edge such that $\rho(e) = \{r_1, r_2\}$ with r_1 closer to the root than r_2 in T . If another edge e' shares r_2 with e , i.e., $\rho(e') = \{r_2, r_3\}$, and $\omega(e') < \omega(e)$, then sliding e' along e results in a graph G' that is identical to G , except that the incidence function maps e' to $\{r_1, r_3\}$.*

The key result by Eppstein shows that applying sliding transformations on G along a rooted MST T to a fixpoint results in an equivalent graph of G . To state this formally, we first define an ordering on graphs based on sliding transformations. It is easy to verify the following is a partial order:

Definition 17 (Sliding Partial Order). *We write $G \leq_T G'$ if G' can be obtained from G by applying a sequence of sliding transformations along a rooted MST T of G .*

Applying sliding transformations to a fixpoint therefore yields a maximal element under the sliding partial order.

Theorem 18 (Sliding produces G^\equiv at fixpoint [14]). *Given a weighted graph G and a rooted MST T , and let G^\equiv be a maximal element under \leq_T . Then G^\equiv is an equivalent graph of G .*

The choice of the initial rooted MST can affect the structure of the equivalent graph, but the order of sliding transformations performed has no impact [14].

Example 19. *Given the 6-clique line graph L_6 in Figure 1b, we find a rooted MST T_W as shown in Figure 2a. Among all the tree edges shown as black solid lines, we can only apply the sliding transformation to the edge $\{T, P\}$ whose weight $\omega(\{T, P\}) = |\{a\}| = 1$ is lighter than its parent tree edge $\{W, T\}$ with $\omega(\{W, T\}) = |\{a, c\}| = 2$. We slide along the tree edge $\{W, T\}$ to the root so that the edge $\{T, P\}$ becomes $\{W, P\}$ as shown by the solid curve in Figure 2b. Non-tree edges can slide similarly. We consider two examples, $\{S, U\}$ illustrated with a dashed line and $\{Y, T\}$ with a dotted line in Figure 2a. All other non-tree edges are shown*

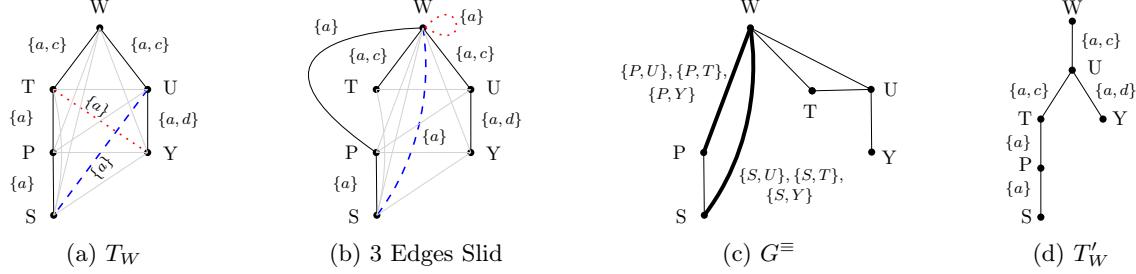


Figure 2: T_W is an MST of the line graph in Figure 1b where the black solid lines stand for the tree edges, the dashed line for an MST edge (S, U) , the dotted line for a non-MST edge (T, Y) and gray solid lines for the remaining non-tree edges. G^{\equiv} is the resulting equivalent graph where the thick lines highlight the parallel edges.

in light gray. We can slide $\{S, U\}$ along tree edge $\{U, W\}$ to become $\{S, W\}$ as shown by the dashed curve in Figure 2b. Both ends of $\{Y, T\}$ can slide along the tree edges to the root so that the edge becomes a self-loop as shown by the dotted loop in Figure 2b. This edge will not appear in any spanning tree of L_6^{\equiv} , and therefore not a part of any MST of L_6 . We refer to such an edge as a non-MST edge, as opposed to an MST edge. Denoting the graph after sliding as L'_6 , ordering $L_6 \leq_{T_W} L'_6$ holds. By applying sliding transformations to a fixpoint, we obtain an equivalent graph L_6^{\equiv} , where there are two sets of parallel edges highlighted by thick lines in Figure 2c (self-loops are omitted). For example, the tree edge $\{T, P\}$ in T_W and non-tree edges $\{P, U\}, \{P, T\}, \{P, Y\}$ become parallel in L_6^{\equiv} between P and W . Therefore $L_6 \leq_{T_W} L'_6 \leq_{T_W} L_6^{\equiv}$ holds under the sliding partial order. We can easily verify that each spanning tree of L_6^{\equiv} corresponds to an MST of L_6 , such as T'_W in Figure 2d.

4.2 Enumerating Join Trees of α -Acyclic Hypergraphs

The bottleneck of Eppstein’s algorithm for constructing equivalent graphs of arbitrary weighted graphs lies in a subroutine that identifies where each edge will eventually slide to. Because each edge can only slide along a heavier edge, it will eventually be “blocked” by a lighter or equally weighted edge along its path to the root. The subroutine essentially performs binary search to find the blocking edge, leading to the $\log n$ factor in the overall complexity. The key to our improvement is to show that for every acyclic hypergraph H , we can construct an equivalent hypergraph H^* whose join trees one-to-one correspond to those of H , but one special join tree of H^* has *monotonically increasing weight* from root to leaf, which enables constant-time identification of the blocking edge.

Definition 20. A monotonic weight join tree $\hat{T} \in \mathcal{T}(H)$ is a rooted join tree of H such that for any $e \in E(\hat{T})$ that has a parent edge $p(e)$, $\omega(e) > \omega(p(e))$.

We can always construct such a H^* given any α -acyclic hypergraph H :

Theorem 21. Given an α -acyclic hypergraph $H = (X, R, \chi)$, there exists an equivalent hypergraph $H^* = (X^*, R, \chi^*)$ over the same hyperedge set R that

- admits a monotonic weight join tree $\hat{T} \in \mathcal{T}(H^*)$, and
- $\mathcal{T}(H^*) = \mathcal{T}(H)$.

Example 22. H_6^* in Figure 3a is a hypergraph with similar structure to H_6 in Figure 1a. H_6^* differs from H_6 by one vertex d' and admits the same set of join trees. Therefore, finding an equivalent graph $G^{\equiv}(L(H_6^*))$ is sufficient for enumerating the join trees of H_6 . H_6^* also admits a monotonic weight join tree \hat{T}_P in Figure 3b.

A monotonic weight join tree \hat{T} can accelerate the construction of the equivalent graph in several ways. First, all tree edges are already “in place”, because they cannot slide along their lighter parent edges. We therefore need only consider non-tree edges. For each non-tree edge $e = \{r_i, r_j\}$, there are two possible cases.

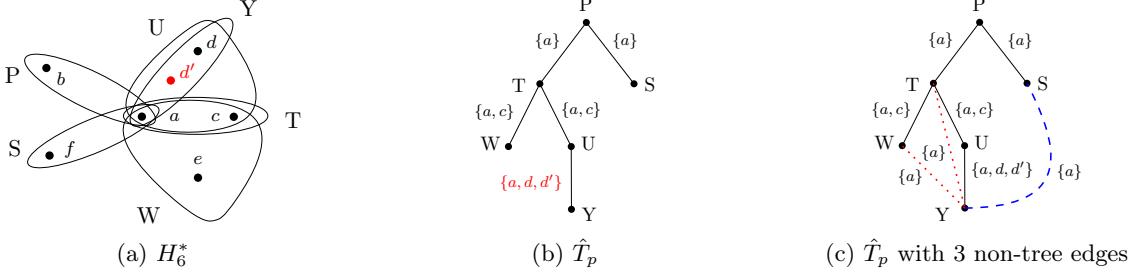


Figure 3: An equivalent hypergraph H_6^* and its monotonic weight join tree \hat{T}_P rooted at P .

First, if one endpoint, say r_i , is an ancestor of the other, say r_j , then e can only slide to become parallel to the edge right below r_i on the path connecting r_i and r_j in \hat{T} . This is because all edges on that path must be no lighter than e due to the running intersection property of join trees, but since \hat{T} has monotonically increasing weight, at most one edge (the one right below r_i) can have equal weight to e . If this edge has weight equal to e , it becomes the blocking edge, and e slides to become parallel to it; otherwise, e slides to a self-loop at r_i . In the second case, if neither endpoint is an ancestor of the other, there are at most two blocking edges, namely those below the lowest common ancestor of r_i and r_j on the path between them. We therefore only need to compare the weight of e with these two potentially blocking edges to determine the destination of e .

Example 23. Consider again \hat{T}_P in Figure 3c. First note that every tree edge is heavier than its parent edge, and each non-tree edge is no heavier than any edge on the path connecting its endpoints in the tree. There is one edge, $\{T, Y\}$, where one endpoint (T) is an ancestor of the other (Y). The only potentially blocking edge is $\{T, U\}$ right below T . But because $\{T, U\}$ is heavier, $\{T, Y\}$ slides to become a self-loop at T . For the edge $\{W, Y\}$, all edges on the tree path connecting the endpoints are heavier, so $\{W, Y\}$ also slides to a self-loop at their LCA T . Finally, $\{Y, S\}$ is blocked by the two edges $\{P, T\}$ and $\{P, S\}$ below their LCA P , as all three edges have a weight of 1. Therefore $\{Y, S\}$ slides to $\{T, S\}$.

Our algorithm for constructing the equivalent graph leverages the insight above to slide each edge in constant time. In the following, we first define a few helpful notations and describe a preprocessing step to build helper data structures, before presenting the algorithm.

Given a rooted MCS tree T_r of a line graph L , a non-tree edge is $e = \{r_i, r_j\} \in E(L) \setminus E(T_r)$. There is a path in T_r between r_i and r_j via their lowest common ancestor $\text{LCA}(r_i, r_j)$. We define the *LCA edges* $\lambda(e) = \lambda(r_i, r_j)$ as a set of at most two tree edges on the path and incident to $\text{LCA}(r_i, r_j)$. If r_i, r_j are ancestor and child, then $|\lambda(e)| = 1$, otherwise, $|\lambda(e)| = 2$.

During pre-processing, we first conduct a breadth-first search on T_r to obtain the depth table \mathbf{d} of each tree node in $\mathcal{O}(|T_r|)$. Then we build two data structures in $\mathcal{O}(|T_r|)$ to facilitate the constant-time query of the lowest common ancestor (LCA) [3] given two nodes, and the level ancestor (LA) [4] of a node r , which is the ancestor of r at a given depth. Algorithm 2 assumes these data structures are prebuilt and available, which allows finding $\lambda(e)$ of any non-tree edge e in constant time as follows:

$$\lambda(\{r_i, r_j\}) = \begin{cases} \{\{r_i, \text{LA}(r_j, \mathbf{d}(r_i) + 1)\}\} & \text{if } r_i = \text{LCA}(r_i, r_j) \\ \{\{r_j, \text{LA}(r_i, \mathbf{d}(r_j) + 1)\}\} & \text{if } r_j = \text{LCA}(r_i, r_j) \\ \{\{l, \text{LA}(r_i, d)\}, \{l, \text{LA}(r_j, d)\}\} & \text{otherwise, } l := \text{LCA}(r_i, r_j), d := \mathbf{d}(l) + 1 \end{cases}$$

We are now ready to present Algorithm 2 which constructs G^\equiv by sliding each non-tree edge $e = \{r_i, r_j\}$ with weight ω^* . There are two cases. First, if $\lambda(e)$ returns a single edge, then one endpoint is an ancestor of the other. By the definition above $\lambda(e)$ always returns the ancestor node l first, while the other node is denoted as l' . The weight of $\{l, l'\}$ is then compared with ω^* : if ω^* is lighter, e slides to a self-loop at l ; otherwise the weights must be equal, and e slides to be parallel to $\{l, l'\}$. Second, if $\lambda(e)$ returns two edges, then neither endpoint is an ancestor of the other. In this case we compare ω^* with the two edges in $\lambda(e)$. If ω^* is lighter than both, e slides to a self-loop at l ; if ω^* equals one of them, e slides to be parallel to that

Algorithm 2: *buildEG*

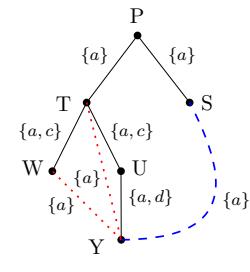
Input: Line graph $L = ((R, E, \rho), \omega)$ of α -acyclic
 H and an MCS tree T_r

Output: Equivalent Graph G^{\equiv}

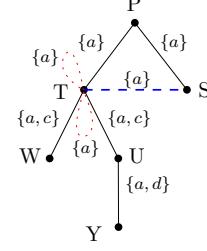
```

1  $G^{\equiv}(R, E, \rho^{\equiv}) \leftarrow (R, E, \rho)$ 
2 for  $e \in E(L) \setminus E(T_r)$  do
3    $\omega^* \leftarrow \omega(e)$ 
4    $r_i, r_j \leftarrow \rho(e)$ 
5   if  $|\lambda(e)| = 1$  then
6      $\{\{l, l'\}\} \leftarrow \lambda(e)$ 
7     if  $\omega^* < \omega(l, l')$  then
8       set  $\rho^{\equiv}(e)$  to  $\{l\}$ 
9     else
10      set  $\rho^{\equiv}(e)$  to  $\{l, l'\}$ 
11   else
12      $\{\{l, l_1\}, \{l, l_2\}\} \leftarrow \lambda(e)$ 
13     if  $\omega^* < \omega(l, l_1) \wedge \omega^* < \omega(l, l_2)$  then
14       set  $\rho^{\equiv}(e)$  to  $\{l\}$ 
15     else if  $\omega^* = \omega(l, l_1)$  then
16       set  $\rho^{\equiv}(e)$  to  $\{l, l_1\}$ 
17     else if  $\omega^* = \omega(l, l_2)$  then
18       set  $\rho^{\equiv}(e)$  to  $\{l, l_2\}$ 
19     else
20       set  $\rho^{\equiv}(e)$  to  $\{l_1, l_2\}$ 
21 return  $G^{\equiv}$ 

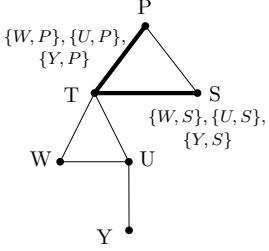
```



(a) MCS Tree $T_P(H_6)$



(b) 3 Non-tree Edges Processed



(c) Equivalent Graph $G^{\equiv}(H_6)$

Figure 4: Algorithm 2 on H_6 of Figure 1a

edge; otherwise, ω^* equals both, and e slides to be incident to each of the highest depth endpoints of both edges. The four cases are exhaustive, as the non-tree edge never outweighs any of its LCA edges by the running intersection property.

Theorem 24. *Given a hypergraph $H^* = (X^*, R, \chi^*)$ with a monotonic weight join tree \hat{T} , Algorithm 2 produces an Equivalent Graph $G^\equiv(L(H^*))$.*

The notation H^* highlights later applications to equivalent hypergraphs, although Theorem 24 applies to any hypergraph admitting a monotonic weight join tree.

Although Algorithm 2 constructs an equivalent graph of $L(H^*)$ in constant time per edge, such an H^* may be asymptotically larger than H . Perhaps surprisingly, running Algorithm 2 *directly on H* produces the same equivalent graph! This is because H^* is constructed in a way that preserves the structure of H , keeping the relative weights of an edge and its LCA edges invariant. Intuitively, we construct H^* by making “local copies” of the vertices in H : this perturbs the weights to be monotonic along \hat{T} , while ensuring the weights are adjusted consistently. In particular, when making a new copy x' of a vertex x , we add x' to *all hyperedges* that contain x , namely $H|_x$. This way, $\chi(e) \subset \chi(e')$ in $L(H)$ if and only if $\chi(e) \subset \chi(e')$ in $L(H^*)$. As a result, all comparisons in Algorithm 2 return the same result when running on H or H^* . We illustrate this with an example, while deferring the detailed construction of H^* to Algorithm 6 in Section C.

Example 25. *In Figure 3a, we made a copy d' of d and added it to all hyperedges containing d in H_6 . This maintains the relative weights between the tree edge $\{U, Y\}$ and the non-tree edges highlighted in Figure 3c: all three non-tree edges remain lighter than $\{U, Y\}$. Were we to make a copy of a to add to $H_6|_a$ (which contains every hyperedge in H_6), the weights of the non-tree edges relative to their respective LCA edges would also remain unchanged.*

Finally, because H and H^* share the same join trees, there must be $G^\equiv(L(H))$ that is the same as $G^\equiv(L(H^*))$. Together with Theorem 24, this means running Algorithm 2 directly on $L(H)$ produces an equivalent graph $G^\equiv(L(H))$ of $L(H)$:

Theorem 26. *Given the line graph L and an MCS tree T_r of an α -acyclic hypergraph H , Algorithm 2 returns an equivalent graph $G^\equiv(L)$ in time $\mathcal{O}(|L|)$.*

Once we compute the equivalent graph $G^\equiv(L)$, we can apply existing algorithms [24, 37] to enumerate all its spanning trees by edits with amortized constant delay. And because the MCS tree can be constructed in linear time from H which in turn can be recovered from $L(H)$ in linear time, the overall time complexity of enumerating all join trees of H is linear in the size of L plus the number of join trees:

Theorem 27. *Given the line graph L of an α -acyclic hypergraph H , the join trees of H can be enumerated in time $\mathcal{O}(|L| + |\mathcal{T}(H)|)$.*

4.3 γ -Acyclic Queries

The run time of Algorithm 2 depends on the size of the line graph which can be quadratically larger than the input hypergraph. If H is γ -acyclic, we can bring the total time complexity of enumeration down to $\mathcal{O}(|H| + |\mathcal{T}(H)|)$. Using an algorithm by Leitert [27], we can construct $L(H)$ from H in time $\mathcal{O}(|H| + |L(H)|)$ when H is γ -acyclic; furthermore, because every edge in $L(H)$ of a γ -acyclic H is an MST edge [27], we have $|H| + |\mathcal{T}(H)| \in \Omega(|L(H)|)$, and so $\mathcal{O}(|H| + |L(H)| + |\mathcal{T}(H)|) = \mathcal{O}(|H| + |\mathcal{T}(H)|)$, hiding the $|L(H)|$ term in the overall complexity. However, the $L(H)$ constructed by Leitert is *unweighted*, so we need to modify Algorithm 2 slightly to slide each non-tree edge based on the weights of its LCA edges alone. We first modify MCS (Algorithm 1) to track the weight of each tree edge as in Algorithm 3. Then, we update the cases of Algorithm 2 to those in Algorithm 4. First, if $\lambda(e)$ returns a single edge, we directly slide e to be parallel to it, because we know e is an MST edge and therefore will not become a self-loop. Otherwise, if $\lambda(e)$ returns two edges, we *compare the weights of those two edges*, instead of comparing them with $\omega(e)$. This is again because we know e is an MST edge, so it must have weight equal to at least one of its LCA edges. If one of the LCA edges is lighter, e must have equal weight to that edge, and we slide e to be parallel to it. Otherwise, if both LCA edges have the same weight, e must also have the same weight, and we slide e to be under both edges to form a triangle. In all cases we only use the weights of the tree edges and do not need to know $\omega(e)$.

Algorithm 3: MCS (γ -acyclic)

Input: γ -acyclic Hypergraph H and root r
Output: T_r and ω

```
1  $p(r) \leftarrow \text{null}; E(T_r) \leftarrow \emptyset$ 
2 while  $R \neq \emptyset$  do
3   for  $x \in r \cap X$  do
4     for  $r' \in R : x \in r'$  do
5        $p(r') \leftarrow r$ 
6      $X \leftarrow X \setminus r$ 
7      $R \leftarrow R \setminus \{r\}$ 
8      $E(T_r) \leftarrow E(T_r) \cup \{\{r, p(r)\}\}$ 
9      $\omega(\{r, p(r)\}) \leftarrow |r \cap p(r)|$ 
10     $r \leftarrow \arg \max_{r \in R} |r \setminus X|$ 
11 return  $T_r, \omega$ 
```

Algorithm 4: buildEG (γ -acyclic)

Input: Weighted T_r , unweighted $L(H)$
Output: Equivalent Graph G^\equiv

```
2 for  $e \in E(L) \setminus E(T_r)$  do
3   ...
5   if  $|\lambda(e)| = 1$  then
6     ...
7   else
8      $\{\{l, l_1\}, \{l, l_2\}\} \leftarrow \lambda(e)$ 
9     else if  $\omega(l, l_1) < \omega(l, l_2)$  then
10      set  $\rho^\equiv(e)$  to  $\{l, l_1\}$ 
11    else if  $\omega(l, l_1) > \omega(l, l_2)$  then
12      set  $\rho^\equiv(e)$  to  $\{l, l_2\}$ 
13    else set  $\rho^\equiv(e)$  to  $\{l_1, l_2\}$ 
14 return  $G^\equiv$ 
```

Theorem 28. *The join trees of a γ -acyclic H can be enumerated in time $\mathcal{O}(|H| + |\mathcal{T}(H)|)$.*

5 The Canonical Join Tree of a Berge-Acyclic Query

An acyclic query can have exponentially many join trees with respect to its size. For example, the line graph of a clique query with n relations is an n -clique K_n with n^{n-2} join trees by Cayley's formula [8]. Enumerating all join trees can be prohibitive for large queries. The query optimizer does not need to consider all possible join trees to achieve good performance. For example, the implementation of Yannakakis' algorithm by Zhao et al. [48] achieves similar performance on any join tree rooted at the largest relation. An alternative to enumeration is therefore to simply construct one join tree for a given root. This can be done in linear time by the Maximum Cardinality Search (MCS) algorithm from a chosen relation as shown in Algorithm 1. In this section, we prove that for a Berge-acyclic query the MCS algorithm produces a *shallowest tree*, where the depth of each tree node is minimized. We prove that this shallowest tree is unique, and therefore call it the *canonical join tree*.

Definition 29. *A join tree T_r rooted at r is canonical if $d(T_r, r_i) \leq d(T'_r, r_i)$ for any other join tree T'_r rooted at r and any $r_i \in R(T_r) = R(T'_r)$.*

A shallow join tree has practical benefits. For example, the depth of the join tree determines the number of sequential steps required in a parallel join algorithm. A shallow join tree also tends to be wide and have more leaves, allowing better utilization of indices.

Although Berge-acyclicity was thought to be too restrictive when it was first introduced to database theory [15], we found it to be general enough to cover almost all acyclic queries encountered in the wild. As shown in Table 1, among 10 454 queries from five popular benchmarks, 9285 are α -acyclic, and only 8 of these are not Berge-acyclic. In retrospect, this should not be surprising, as most joins in relational databases are over primary/foreign keys. Emerging workloads in graph databases usually involve simple graphs and seldom require composite key joins. A query without composite key joins admits a *linear* hypergraph, where each pair of hyperedges shares at most one vertex. The following result establishes an equivalence between α -acyclicity with linearity and Berge-acyclicity.

Proposition 30. *An α -acyclic hypergraph is Berge-acyclic if and only if it is linear.*

By Proposition 30, every edge in the line graph of a Berge-acyclic hypergraph has a weight of 1. Every spanning tree is a maximum spanning tree, therefore a join tree.

Corollary 31. *For a Berge-acyclic hypergraph H , any spanning tree of $L(H)$ is a join tree.*

Name	# Queries	# α -Acyclic	# Composite-Key Joins	# Berge-Acyclic
TPC-H[42]	22	21	2	19
JOB[26]	113	113	0	113
STATS-CEB[19]	2603	2603	0	2603
CE[9]	3004	1839	0	1839
Spider-NLP[47]	4712	4709	6	4703

Table 1: Acyclic queries in the benchmarks (all α -acyclic queries are also γ -acyclic.)

In the rest of this section, we prove the existence and uniqueness of the canonical join tree rooted at any relation of a Berge-acyclic hypergraph, and show that it can be constructed by MCS as in Algorithm 1.

The key insight leading to the existence and uniqueness of the canonical join tree is that the line graph L of a Berge-acyclic hypergraph is *geodetic* [33], meaning that there is a unique shortest path between any pair of vertices. The canonical join tree can then be constructed by taking the union of all shortest paths from the root to each other vertex, as this will guarantee minimal depth for each vertex. The key step to establish the geodetic property is to show that L is a special class of *chordal graphs* called *block graphs*. Chordal graphs are intimately related to acyclic hypergraphs [11], and a block graph is a special chordal graph defined as follows:

Definition 32 (Chordal and Block). *A simple graph $G = (R, E)$ is*

- chordal [12] if every cycle of length at least 4 has a chord, i.e., an edge that is not part of the cycle but connects two vertices of the cycle;
- a block graph [20] if it is chordal and diamond-free, i.e., no subgraph induced by any $R' \subseteq R$ is a diamond (Definition 4).

Lemma 33. *The line graph L of a Berge-acyclic hypergraph H is a block graph.*

Together with the fact that every block graph is *geodetic* (with a unique shortest path between any two vertices) [33], Lemma 33 implies the following corollary.

Corollary 34. *Let L be the line graph of a Berge-acyclic hypergraph H . There is a unique shortest path between any two vertices in L .*

We are now ready to prove the existence and uniqueness of the canonical join tree.

Theorem 35. *A Berge-acyclic hypergraph H has a unique canonical tree.*

Proof. Let L be the line graph of H , and $\mathsf{P}(r, r')$ be the shortest path in L between $r, r' \in R(L)$, we prove that $T_r = \bigcup_{r' \in R(L)} \mathsf{P}(r, r')$ is the unique canonical join tree for H rooted at r .

By Corollary 31 any spanning tree of L is a join tree. T_r is connected and spans all vertices in $R(L)$, because it contains the shortest path from r to every $r' \in R(L)$. It remains to show that T_r is acyclic, which we prove by induction on the distance $\text{dist}(r, r')$ between r and r' . Let $R_d = \{r' \in R(L) \mid \text{dist}(r, r') \leq d\}$. $R_0 = \{r\}$ contains only the root. The subgraph $T_r|_{R_0}$ is trivially acyclic. Assuming that $T_r|_{R_{d>0}}$ is acyclic, we consider a vertex $r' \in R_{d+1} \setminus R_d$. Corollary 34 guarantees a unique shortest path between each pair of vertices $r, r' \in R(L)$. Each r' is connected to a unique neighbor $r'' \in R_d \setminus R_{d-1}$ that is at distance d from r . Otherwise, there are at least two distinct shortest paths from r to r' . Therefore $T_r|_{R_{d+1}}$ is acyclic, and T_r is a spanning tree thus a join tree of H .

The join tree T_r is canonical, because the path from r to each $r' \in R(L)$ is the shortest and therefore minimizing the depth $\mathsf{d}(T_r, r')$. The canonical tree is unique by Corollary 34. \square

The canonical join tree can be constructed by the MCS algorithm as shown in Algorithm 1:

Theorem 36. *Given a Berge-acyclic hypergraph H , running Algorithm 1 from $r \in R(H)$ constructs the canonical tree $T_r(H)$.*

Algorithm 5: Binary-to-JT Conversion

Input: Left-deep plan (r_1, \dots, r_n)
Output: Rooted join tree T_{r_1}

```

1  $p(r_1) \leftarrow \text{null}; E(T_{r_1}) \leftarrow \emptyset$ 
2 for  $i \leftarrow 2$  to  $n$  do
3    $\text{key} \leftarrow r_i \cap \bigcup_{k < i} r_k$ 
4   for  $j \leftarrow 1$  to  $i - 1$  do
5     if  $\text{key} \subseteq r_j$  then
6        $E(T_{r_1}) \leftarrow E(T_{r_1}) \cup \{(r_i, r_j)\}$ 
7       break
8 return  $T_{r_1}$ 

```

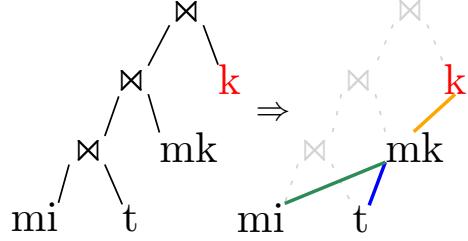


Figure 5: Algorithm 5[22] on JOB-3a[26]

6 Converting a Binary Join Plan to a Join Tree

Recent approaches [22, 2] that convert a binary join plan into a join tree have gained popularity as they allow system builders to leverage existing query optimizers designed for binary join plans. In this section, we focus on an algorithm by Hu et al. [22] to convert left-deep linear join plans into join trees as shown in Figure 5. We prove that the algorithm converts any connected left-deep linear join plan into a join tree if and only if the query is γ -acyclic. This can be seen as a new characterization of γ -acyclic queries. We formally define binary join plans and describe the algorithm by Hu et al. in Algorithm 5.

Definition 37. A left-deep linear plan is a sequence of hyperedges (r_1, r_2, \dots, r_n) . It is connected if for each $r_{i \geq 2}$, $\exists r_{j < i} : r_i \cap r_j \neq \emptyset$.

Query optimizers strive to produce connected plans, to avoid expensive Cartesian products. Many optimizers produce exclusively left-deep linear plans. Plans that are not left-deep are called *bushy*, and such plans may still be decomposed into left-deep fragments [44].

Given a left-deep linear plan, Hu et al. [22] generate a join tree with Algorithm 5. The algorithm chooses the first relation r_1 as the root and iterates through the rest of the plan. For each relation $r_{i \in [2, n]}$, it finds the first relation r_j that contains all attributes shared by r_i with all previous relations, $\text{key} := r_i \cap \bigcup_{k < i} r_k$, and assigns r_j as the parent of r_i . The algorithm constructs a join tree if it finds a parent for each r_i .

Hu et al. [22] proved that the algorithm succeeds whenever the input plan is the reverse of a GYO-reduction order. They also observed that every left-deep linear plan produced for queries in standard benchmarks is indeed the reverse of a GYO-reduction order. This is not a coincidence, as we show that every connected left-deep linear join plan must be the reverse of a GYO-reduction order if and only if the query is γ -acyclic.

Theorem 38. A query is γ -acyclic if and only if every connected left-deep linear join plan for the query is the reverse of a GYO-reduction order.

Immediately following Theorem 38, we can conclude the following.

Corollary 39. For any γ -acyclic query, Algorithm 5 converts a given connected left-deep linear join plan to a join tree.

7 Conclusion and Future Work

We proposed three approaches for constructing join trees. Our enumeration algorithm in Section 4 generates join trees by edits with amortized constant delay; in Section 5, we showed that the Maximum Cardinality Search algorithm constructs the unique shallowest join tree for any Berge-acyclic query; in Section 6, we characterize the class of binary join plans that can be converted to join trees. Practitioners can choose from the three approaches to integrate instance-optimal algorithms into their system: a cost-based optimizer can use our enumeration algorithm to generate candidate query plans; a system aiming to support very large

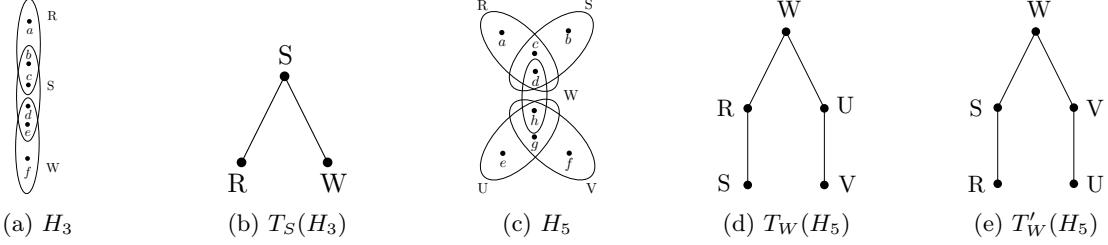


Figure 6: H_3 is not Berge-acyclic but admits a unique canonical join tree $T_S(H_3)$ with any relation chosen as root, such as $T_S(H_3)$. H_5 is γ -acyclic and does not admit unique canonical join trees at any relation chosen as root. For example, $T_W(H_5)$ and $T'_W(H_5)$ are MCS trees generated by Algorithm 1. Neither of them is a canonical join tree rooted at W .

queries can implement the MCS algorithm to generate shallow join trees to improve parallelism; and a more conservative extension can derive join trees using existing optimization infrastructure, while our final result guarantees the validity of the output.

Future work includes compact representations of join trees for dynamic programming, as in binary plan optimizers, and the challenging cost estimation for Yannakakis-style algorithms: the random-walk approach [28] models joint probabilities for binary joins, whereas an efficient and accurate solution for semijoins remains to be found.

Our work also raises further theoretical questions. Can join tree enumeration achieve worst-case constant delay? We proved *Berge-acyclicity* sufficient for the existence and uniqueness of the canonical join tree, but it is not necessary, and *γ -acyclicity* is insufficient as shown in Figure 6. What is the precise characterization of hypergraphs that admit a unique canonical join tree for any root, or for *some* root? How can our algorithms be extended to enumerate tree decompositions à la Carmeli et al. [7]? One possible direction is to further develop connections between MSTs of the line graph and tree decompositions of the hypergraph. For example, we are considering using the determinants of the line graph Laplacian as heuristics to guide the search for good tree decompositions.

References

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: <http://webdam.inria.fr/Alice/>.
- [2] Liese Bekkers, Frank Neven, Stijn Vansumeren, and Yisu Remy Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, 2025. URL: <https://www.vldb.org/pvldb/vol18/p2413-vansumeren.pdf>.
- [3] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
- [4] Michael A Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- [5] Alain Bretto. *Hypergraph Theory: An Introduction*. Springer Publishing Company, Incorporated, 2013.
- [6] Nicolas Bruno, César A. Galindo-Legaria, and Milind Joshi. Polynomial heuristics for query optimization. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, 2010.
- [7] Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM*

SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017, pages 273–287. ACM, 2017. doi:10.1145/3034786.3056109.

- [8] Arthur Cayley. A theorem on trees. *Quarterly Journal of Pure and Applied Mathematics*, 23:376–378, 1889.
- [9] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proceedings of the VLDB Endowment*, 15(8):1533–1545, 2022.
- [10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 4th edition, 2022. Sections 22.2–22.3.
- [11] Alessandro D’atri and Marina Moscarini. On hypergraph acyclicity and graph chordality. *Information Processing Letters*, 29(5):271–274, 1988.
- [12] Pablo De Caria and Marisa Gutierrez. On basic chordal graphs and some of its subclasses. *Discrete Applied Mathematics*, 210:261–276, 2016. LAGOS’13: Seventh Latin-American Algorithms, Graphs, and Optimization Symposium, Playa del Carmen, México — 2013. URL: <https://www.sciencedirect.com/science/article/pii/S0166218X15002243>, doi:10.1016/j.dam.2015.05.002.
- [13] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 785–796, 2007.
- [14] David Eppstein. Representing all minimum spanning trees with applications to counting and generation. UC Irvine, 1995.
- [15] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM (JACM)*, 30(3):514–550, 1983.
- [16] Leonidas Fegaras. A new heuristic for optimizing large queries. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer, 1998.
- [17] Pit Fender and Guido Moerkotte. Counter strike: Generic top-down join enumeration for hypergraphs. *Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1822–1833, 2013.
- [18] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. Fast parallel hypertree decompositions in logarithmic recursion depth. *ACM Trans. Database Syst.*, 49(1):1:1–1:43, 2024. doi:10.1145/3638758.
- [19] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endow.*, 15(4):752–765, 2021. URL: <https://www.vldb.org/pvldb/vol115/p752-zhu.pdf>, doi:10.14778/3503585.3503586.
- [20] Frank Harary. A characterization of block-graphs. *Canadian Mathematical Bulletin*, 6(1):1–6, 1963.
- [21] Zongyan He and Jeffrey Xu Yu. A branch-&-bound algorithm for fractional hypertree decomposition. *Proc. VLDB Endow.*, 17(13):4655–4667, 2024. URL: <https://www.vldb.org/pvldb/vol17/p4655-he.pdf>.
- [22] Zeyuan Hu, Yisu Remy Wang, and Daniel P Miranker. Treetracker join: Simple, optimal, fast. *arXiv preprint arXiv:2403.01631*, 2024.
- [23] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984. doi:10.1145/1270.1498.
- [24] Sanjiv Kapoor and H. Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM Journal on Computing*, 24(2):247–265, 1995. arXiv:<https://doi.org/10.1137/S009753979225030X>, doi:10.1137/S009753979225030X.

- [25] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann, 1986. URL: <http://www.vldb.org/conf/1986/P128.PDF>.
- [26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.
- [27] Arne Leitert. Computing the union join and subset graph of acyclic hypergraphs in subquadratic time. In Anna Lubiw and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 17th International Symposium, WADS 2021, Virtual Event, August 9-11, 2021, Proceedings*, volume 12808 of *Lecture Notes in Computer Science*, pages 571–584. Springer, 2021. doi:10.1007/978-3-030-83508-8_41.
- [28] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 615–629. ACM, 2016. doi:10.1145/2882903.2915235.
- [29] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983.
- [30] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06*, page 930–941. VLDB Endowment, 2006.
- [31] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552. ACM, 2008. doi:10.1145/1376616.1376672.
- [32] Thomas Neumann. Query simplification: Graceful degradation for join-order optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 403–414, 2009.
- [33] Øystein Ore. *Theory of Graphs*, volume 38 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, Providence, RI, 1962. doi:10.1090/coll/038.
- [34] Benjamin Scheidt and Nicole Schweikardt. Counting homomorphisms from hypergraphs of bounded generalised hypertree width: A logical characterisation. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science, MFCS 2023, August 28 to September 1, 2023, Bordeaux, France*, volume 272 of *LIPICS*, pages 79:1–79:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: <https://doi.org/10.4230/LIPIcs.MFCS.2023.79>, doi:10.4230/LIPIcs.MFCS.2023.79.
- [35] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, Boston, MA, 4th edition, 2011. Chapter 4: BreadthFirstPaths.
- [36] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.
- [37] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM J. Comput.*, 26(3):678–692, 1997. doi:10.1137/S0097539794270881.
- [38] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

- [39] Vaishali Surianarayanan, Anikait Mundhra, Ajaykrishnan E. S, and Daniel Lokshtanov. Fast hyper-tree decompositions via linear programming: Fractional and generalized. *Proc. ACM Manag. Data*, 3(3):159:1–159:27, 2025. doi:10.1145/3725296.
- [40] Arun N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 367–376, 1989.
- [41] 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.
- [42] Transaction Processing Performance Council. TPC Benchmark H (Decision Support). https://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf, 2023. Standard Specification, Revision 3.0.1.
- [43] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proc. ACM Manag. Data*, 3(3):235:1–235:28, 2025. doi:10.1145/3725423.
- [44] Yisu Remy Wang, Max Willsey, and Dan Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proceedings of the ACM on Management of Data*, 1(2):1–23, 2023.
- [45] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB ’81, page 82–94. VLDB Endowment, 1981.
- [46] C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society’s Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*, pages 306–312. IEEE, 1979. doi:10.1109/CMPSC.1979.762509.
- [47] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun’ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 3911–3921. Association for Computational Linguistics, 2018. URL: <https://doi.org/10.18653/v1/d18-1425>. doi:10.18653/V1/D18-1425.
- [48] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huachen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.
- [49] Yun-zhou Zhu. Line graph of gamma-acyclic database schemes and its recognition algorithm. In *VLDB*, pages 218–221, 1984.

A Data Structures for Constant Time Operations on Graphs

Given line graph L and MCS tree T , Algorithm 2 iterates over non-tree edges of L . Moreover, the algorithm uses data structures $d(\cdot)$, $LA(\cdot)$ and $LCA(\cdot)$, in determining the LCA edges of a non-tree edge. In this section, we describe how to enumerate over non-tree edges in time linear in the number of vertices and edges of L and how to determine the LCA edges of a non-tree edge in constant time.

Given a graph $G = (V, E)$, we assume the vertices are represented by integers from 1 to $|V|$. Given a tree T of G rooted at node r , data structure $d_T(\cdot)$ stores for each node r_i the length of the shortest path between r and r_i in T . Data structure $LCA_T(\cdot)$ stores for each pair of nodes r_i, r_j the lowest common ancestor of r_i, r_j in T . Finally, data structure $LA_T(\cdot)$ stores for each node-integer pair r_i, j , the ancestor of node r_i at

depth j . Each of these data structures can be established from a tree T in time linear in the size of the tree and support constant time look-ups.

Observation 40. *Given rooted tree T , data structures $\mathbf{d}(\cdot)$, $\mathbf{LCA}(\cdot)$ and $\mathbf{LA}(\cdot)$ can be established in time $\mathcal{O}(|T|)$ and support $\mathcal{O}(1)$ time look-ups [10, 35, 3, 4].*

Given the line graph L of an α -acyclic hypergraph H and an MCS tree T of H , recall that the LCA edges $\lambda(e)$ of a non-tree edge $e = (r_i, r_j)$ of T is defined as the set of edges incident to $\mathbf{LCA}(r_i, r_j)$ that lie on the unique path between r_i and r_j in T . The following observation shows that the LCA edges of a non-tree edge can be found in constant time given data structures $\mathbf{d}(\cdot)$, $\mathbf{LCA}(\cdot)$ and $\mathbf{LA}(\cdot)$.

Observation 41. *Given the line graph L of an α -acyclic hypergraph H , an MCS tree T and data structures $\mathbf{d}(\cdot)$, $\mathbf{LCA}(\cdot)$ and $\mathbf{LA}(\cdot)$, the LCA edges of any non-tree edge can be found in constant time.*

Proof. Given a non-tree edge $e = (r_i, r_j)$ we find $\mathbf{LCA}(r_i, r_j) = l$ in constant time by Observation 40. The depth $d = \mathbf{d}(l) + 1$ is likewise found in constant time. The edge $e_1 = (l, r_1)$, where node $r_1 = \mathbf{LA}(r_i, d)$ is found by a constant time look-up, is adjacent to l and lies on the unique path between r_i and r_j . The same holds for edge $e_2 = (l, r_2)$ where node $r_2 = \mathbf{LA}(r_j, d)$. \square

In the remainder of this section, we describe how Algorithm 2 iterates over and performs sliding transformations on non-tree edges.

We use an array L to represent the adjacency list of a line graph where entry $L[i]$ consists of a linked list of tuples (j, ω) where j is a neighbor of i and ω is the weight of the edge (i, j) . A rooted tree T_r is likewise implemented as an array, where the index i represents the vertex i , and the i -th entry $T_r[i]$ stores (p, ω) where p is the parent of i and ω is the weight of the edge (i, p) . The depth table of the tree is also implemented as an array to enable constant time lookup for the depth of a vertex $\mathbf{d}(T_r, i)$.

As we iterate over each edge (j, ω) of $L[i]$, we check if the current edge (i, j) is a tree edge by performing constant-time query of $T_r[i]$ and $T_r[j]$ to see if one of them is the parent of the other. From Observation 41 we know that we can find the LCA edges $\lambda(i, j) = \{e_1, e_2\}$ in constant time. There's no general solution to lookup an entry in a linked list in constant time. Instead of iterating over the linked-list representation of L to find the edge weights of e_1 and e_2 , we can lookup their weights from the array representation of T_r in constant time. Let $e_1 = \{r_1, p(r_1)\}$ and $e_2 = \{r_2, p(r_2)\}$ with weights ω_1 and ω_2 . Because both e_1 and e_2 are tree edges, we can find their weights by querying $T_r[r_1] = (p(r_1), \omega_1)$ and $T_r[r_2] = (p(r_2), \omega_2)$. Each of these operations takes constant time.

B Missing Proofs in Section 3

We prove Lemma 14 of section 3 below.

Lemma 14 *Let T be an MCS tree. For an edge $e \in T$ that has a parent, then*

1. $e \not\subseteq p(e)$
2. For another edge $e' \in T : (e \setminus p(e)) \cap (e' \setminus p(e')) \neq \emptyset \implies p(e) = p(e')$.

Proof. In Algorithm 1, a child r_c is only connected to a parent r if they share some previously unmarked vertex x . Because every vertex is marked only once, and the parent r_p of r was labeled before r , x cannot be marked by r 's parent r_p . Let e be edge between r and r_c , then the edge between r and r_p is $p(e)$, and $x \in e$ but $x \notin p(e)$, therefore $e \not\subseteq p(e)$.

We prove the second statement by contrapositive, assuming $p(e) \neq p(e')$. We also treat each edge as a set of vertices below, as we can identify each tree edge with its set of incident vertices.

Case 1: $p(e') = e$, then $(e \setminus p(e)) \cap (e' \setminus e) \subseteq (e \cap e') \setminus e = \emptyset$.

Case 2: $p(e') \neq e$. Without loss of generality, we suppose e is no further from the root than e' . Let $p(e') = e''$, then e'' is on the path between e and e' in T . If $(e \setminus p(e)) \cap (e' \setminus e'') \neq \emptyset$, then $\exists x \in (e \cap e') \setminus e''$, which violates the running intersection property of join trees.

\square

C Missing Proofs in Section 4

In this section we present the missing proofs of Section 4. In particular, in Section C.1 we show that there exists an equivalent hypergraph H^* of a given hypergraph H that admits a monotonic weight join tree and preserves the set of join trees of H . This is proven in Theorem 21. Thereafter, we prove Theorem 24 showing that Algorithm 2 produces an equivalent graph when given input $L(H^*)$ and a monotonic weight join tree of H^* . Next we show that Algorithm 2 behaves identically on the input of line graph $L(H)$ and an MCS tree of H as it does on $L(H^*)$ and a monotonic weight join tree of H^* . Algorithm 2 therefore produces an equivalent graph of $L(H)$. This is proven in Theorem 26. Finally, we prove Theorem 27, showing that the join trees of an α -acyclic hypergraph can be enumerated in time $\mathcal{O}(|L| + |\mathcal{T}(H)|)$ given its line graph L .

In Section C.2 we prove Theorem 28 showing that the join trees of a γ -acyclic H can be enumerated in time $\mathcal{O}(|H| + |\mathcal{T}(H)|)$.

C.1 Missing Proofs in Section 4.2

We define the notion of an equivalent hypergraph below.

Definition 42. *Given an α -acyclic hypergraph $H = (X, R, \chi)$, hypergraph $H^* = (X^*, R, \chi^*)$, over the same hyperedge set R , is an equivalent hypergraph of H , if H^* ,*

- *admits a monotonic weight join tree $\hat{T} \in \mathcal{T}(H^*)$, and*
- $\mathcal{T}(H^*) = \mathcal{T}(H)$.

We formalize the relationship between an α -acyclic hypergraph H and its equivalent hypergraph H^* using the concept of *hypergraph homomorphisms*. Note the definition of homomorphisms is not standardized in the literature,⁴ and we choose ours carefully to simplify the proofs.

Definition 43 (Hypergraph Homomorphism). *Let H_1 and H_2 be hypergraphs. A homomorphism from H_1 to H_2 is a pair of functions $(f_X : X(H_1) \rightarrow X(H_2), f_R : R(H_1) \rightarrow R(H_2))$, such that for every hyperedge $r_1 \in R(H_1)$ and every vertex $x_1 \in r_1$, the image $f_X(x_1)$ belongs to the hyperedge $f_R(r_1)$ in H_2 .*

We write $H_1 \rightarrow H_2$ to denote the existence of such a homomorphism.

When there is no ambiguity from context, we simply write f for f_X or f_R . Intuitively, homomorphisms preserve relations. A *strong* homomorphism also *reflects* relations:

Definition 44 (Strong Homomorphism). *Let $f : H_1 \rightarrow H_2$ be a hypergraph homomorphism. We call f strong if it also reflects non-incidence, in the following sense: for every hyperedge $r_1 \in R(H_1)$ and every vertex $x_1 \in X(H_1)$ that does not belong to r_1 , the vertex $f_X(x_1)$ does not belong to the hyperedge $f_R(r_1)$ in H_2 .*

We write $H_1 \twoheadrightarrow H_2$ to denote the existence of a strong homomorphism from H_1 to H_2 .

Equivalently, under a strong homomorphism, a vertex belongs to a hyperedge in H_1 if and only if its image belongs to the image of that hyperedge in H_2 .

We only consider hypergraphs over the same set of edges,⁵ and from now on we assume f_R is the identity function.

We show that if there is a strong homomorphism $f : H' \twoheadrightarrow H$, then every join tree of H is also a join tree of H' .

Lemma 45. *If $H' \twoheadrightarrow H$ then $\mathcal{T}(H) \subseteq \mathcal{T}(H')$.*

Proof. Let T be a join tree of H , and x' be any vertex in H' . Consider the neighborhood $H'|_{x'}$ of x' in H' comprising the set of hyperedges $\{r' \in R(H') \mid x' \in r'\}$. Since $H' \twoheadrightarrow H$, we have that $x' \in r'$ if and only if $f(x') \in f(r') = r'$. The neighborhood $H|_{f(x')}$ of $f(x')$ in H comprises of the set of hyperedges $\{r \in R(H) \mid f(x') \in r\}$. Therefore, $r' \in R(H'|_{x'}) \iff f(r') = r' \in R(H|_{f(x')})$. Because T is a join tree of H , $H|_{f(x')}$ is connected in T , therefore $H'|_{x'}$ is also connected in T and T is also a join tree of H' . \square

⁴A textbook on hypergraphs [5] contains two different definitions of hypergraph homomorphism; one is the same as ours, while the other is also used by Scheidt and Schweikardt [34].

⁵The hypergraphs share the same hyperedge set R but differ in incidence function χ , i.e., the same edge may be incident to different vertices in different hypergraphs.

Algorithm 6: Construction of H^*

Input: α -acyclic hypergraph H and rooted MCS tree T_r
Output: H^* in Theorem 21

```

1  $q \leftarrow \text{queue}()$ 
2  $H^*(X^*, R) \leftarrow H(X, R)$ 
3 for  $r' \in c(r)$  in  $T_r$  do
4    $e \leftarrow \{r, r'\}; q.\text{enqueue}(e)$ 
5 while  $q \neq \emptyset$  do
6    $e \leftarrow q.\text{dequeue}()$ 
7    $\Delta \leftarrow \omega(p(e)) - \omega(e) + 1$ 
8   if  $\Delta > 0$  then
9      $x \leftarrow \text{a vertex in } \chi(e) \setminus \chi(p(e))$ 
10    make  $\Delta$  duplicates of  $x$  in  $X^*$ 
11   for  $e' \in c(e)$  do
12      $q.\text{enqueue}(e')$ 
13 return  $H^*$ 
```

Our goal will be to modify an input hypergraph H such that its line graph admits a monotonic weight join tree. The lemma above ensures that if this is done in such a way that a strong homomorphism between H and its modification is preserved then the set of join trees of H is preserved.

Definition 46 (Vertex Duplication). *Given a hypergraph H and a vertex $x \in X(H)$, duplicating x produces a new hypergraph H' that differs from H by only one vertex $\{x'\} = X(H') \setminus X(H)$ which is added to each $r \in H|_x$.*

We will refer to such a hypergraph H' as a *vertex duplication* of H or simply a *duplicated* hypergraph of H . We generalize duplication to a set of vertices in the natural way. Importantly, vertex duplication preserves the set of join trees of the original hypergraph.

Lemma 47. *Let H' be a duplicated hypergraph of H . Then,*

1. $H' \twoheadrightarrow H$ and $H' \twoheadleftarrow H$
2. $\mathcal{T}(H') = \mathcal{T}(H)$
3. $L(H') = L(H)$

Proof. **1.** We only need to show that $H' \twoheadrightarrow H$ ($H' \twoheadleftarrow H$) holds for H' derived from H by a single vertex duplication. $H' \twoheadrightarrow H$ ($H' \twoheadleftarrow H$) will then hold for any duplicate H' derived by a series of vertex duplications since strong homomorphisms are closed under composition.

Suppose H' is derived from H by duplicating $x \in X(H)$ to x' .

Consider a pair of functions $(f_X : X(H') \mapsto X(H), R(H)_{\text{id}})$ where f_X maps x' to x . Otherwise, it is an identity map $X(H)_{\text{id}}$. This function pair constitutes a strong hypergraph homomorphism $H' \twoheadrightarrow H$. Likewise, the pair of functions $(X(H)_{\text{id}}, R(H)_{\text{id}})$ constitutes a homomorphism $H \twoheadrightarrow H'$.

2. It follows from Lemma 45 and the facts above.

3. By the strong homomorphisms, the hyperedge sets have the same cardinality $|R(H)| = |R(H')|$. So do the vertex sets of the line graphs, $R(L(H)) = R(H)$ and $R(L(H')) = R(H')$. By construction, the vertex duplication adds x' to each hyperedge containing x , therefore does not create any additional edges in the line graph, $E(L(H')) = E(L(H))$. \square

Vertex duplication allows us to modify a hypergraph while preserving its set of join trees. In particular, duplicating vertices can selectively increase the weights of chosen hyperedge intersections without affecting the line graph or the join tree structure (cf. Lemma 47).

We use this operation to enforce monotonicity of edge weights along a given join tree. Given an α -acyclic hypergraph H together with a rooted MCS tree T , we can construct an equivalent hypergraph H^* by duplicating vertices along the edges of T . The construction proceeds in a breadth-first manner from the root of T and duplicates vertices only when necessary to ensure that each tree edge is strictly heavier than its parent.

Obtaining an equivalent graph in this manner always succeeds.

Theorem 21 *Given an α -acyclic hypergraph $H = (X, R, \chi)$, there exists an equivalent hypergraph $H^* = (X^*, R, \chi^*)$ over the same hyperedge set R that*

- *admits a monotonic weight join tree $\hat{T} \in \mathcal{T}(H^*)$, and*
- $\mathcal{T}(H^*) = \mathcal{T}(H)$.

Proof. To prove existence, we construct hypergraph H^* by vertex duplication (Algorithm 6). Lemma 47 will guarantee that $\mathcal{T}(H^*) = \mathcal{T}(H)$. Therefore, we only need to show the existence of monotonic weight join tree $\hat{T} \in \mathcal{T}(H^*)$. Algorithm 6 takes as input hypergraph H along with an MCS tree T of H . The algorithm constructs H^* by vertex duplication in a breadth-first manner from the root of T to make it a monotonic weight join tree in H^* .

Algorithm 6 starts with enqueueing each edge of T incident to the root. Once a tree edge $e \in E(T_r)$ is dequeued, we check for the number of duplications needed to make it heavier than its parent edge $p(e)$, namely $\Delta = \omega(p(e)) - \omega(e) + 1$. The duplication is performed only if $\Delta > 0$. Lemma 14 guarantees that $\chi(e) \setminus \chi(p(e)) \neq \emptyset$. Therefore, we can always duplicate a vertex $x \in \chi(e) \setminus \chi(p(e))$ for Δ times such that $\omega(e) > \omega(p(e))$.

We prove that this produces a monotonic weight join tree by induction on a breadth-first ordering (with respect to root r) of the edges in T .

Base case: The base case is an edge e incident to the root. Since e has no parent $\omega(e) > \omega(p(e))$ holds vacuously.

Ind. step: Let (e_1, \dots, e_m) be a breadth-first ordering of the edges in T and assume $\omega(e_i) > \omega(p(e_i))$ for all $i < k - 1$.

Let $e_k = (r_u, r_d)$ where $d(T, r_u) < d(T, r_d)$ and let $x \in \chi(e_k) \setminus \chi(p(e_k))$. Vertex x only occurs in the subtree rooted at r_u . Otherwise, it violates the running intersection property. Therefore, the duplication of x does not affect the weight of any edge e_i , $i \in [k - 1]$ outside of subtree rooted at r_u and therefore $\omega(e_i) > \omega(p(e_i))$ holds under any number of duplications of x . For any e_i lying within the subtree rooted at r_u , e_i is necessarily a sibling of e_k since edges of T are breadth-first ordered. Hence $p(e_i)$ lies outside this subtree and any number of vertex duplications of x cannot decrease the weight of e_i with respect to its parent. Therefore, $\omega(e_i) > \omega(p(e_i))$ for all $i \in [k]$ after performing Δ many vertex duplications of x .

Consequently, tree T has monotonic weights when Algorithm 6 terminates. \square

The above theorem guarantees that for any α -acyclic hypergraph we are able to obtain an equivalent hypergraph H^* . Next we prove Theorem 24 showing that Algorithm 2 produces the equivalent graph of line graph $L(H^*)$.

Theorem 24 *Given a hypergraph $H^* = (X^*, R, \chi^*)$ with a monotonic weight join tree \hat{T} , Algorithm 2 produces an Equivalent Graph $G^{\equiv}(L(H^*))$.*

Proof. We denote the line graph of H^* as L^* . The proof is by contradiction. We assume that the resulting graph G is reached from a series of valid sliding transformations with respect to \hat{T} but is not an equivalent graph. By Theorem 18, an equivalent graph $G^{\equiv}(H^*)$ exists such that $G \leq_{\hat{T}} G^{\equiv}(H^*)$. By Definition 16, an edge $e^* \in E(G)$ can be further slid towards the root of \hat{T} . This edge has to be a non-tree edge $e^* \in E(L^*) \setminus E(\hat{T})$, otherwise \hat{T} is not monotonic weight, violating Definition 20. Thus there must be a non-tree edge that can be slid. Algorithm 2 iterates over all non-tree edges. If an edge e has the same weight as exactly one (tree) edge e' in $\lambda(e)$ the algorithm slides e parallel to this edge. If edge e could slide further in G , then so could tree edge e' , again contradicting monotonicity of \hat{T} . If instead $|\lambda(e)| = 2$ and e has the same weight as both edges $e', e'' \in \lambda(e)$ then the algorithm slides e such that it becomes incident to each of the highest depth endpoints of e' and e'' . By the definition of a sliding transformation e can therefore not slide.

Of note is the case where a non-tree edge e weighs less than all edges in $\lambda(e)$. The algorithm then slides e to a self loop at $l = \text{LCA}(e)$. Such an edge may still be able to slide if edge $\{l, p(l)\}$ weighs more than edge e . In this sense graph G returned by Algorithm 2 may not be maximal under partial order $\leq_{\hat{T}}$. Therefore it remains to show that these self loops do not prevent G from being an equivalent graph. To see this, notice that no self loop can take part in a spanning tree. Therefore any spanning tree of graph obtained upon sliding self loops of G to a fixpoint (G^{\equiv}) is also a spanning tree of G . Hence, G and G^{\equiv} have the same set of spanning trees and G is an equivalent graph. \square

The next result follows from the construction of the equivalent hypergraph.

Lemma 48. *Let $H^* = (X^*, R, \chi^*)$ be an equivalent hypergraph derived from $H = (X, R, \chi)$ by vertex duplication. For any non-tree edge e of an MCS tree T of H ,*

- If $\chi(e) = \chi(e')$ for some $e' \in \lambda(e)$, then $\chi^*(e) = \chi^*(e')$
- If $\chi(e) \subset \chi(e')$ for some $e' \in \lambda(e)$, then $\chi^*(e) \subset \chi^*(e')$

Proof. The duplication of a vertex $x \in \chi(e)$ adds x' to both $\chi(e)$ and $\chi(e')$. If $\chi(e) = \chi(e')$, $\chi^*(e) = \chi(e) \cup \{x'\} = \chi(e') \cup \{x'\} = \chi^*(e')$. A similar argument applies if $\chi(e) \subset \chi(e')$. \square

As Section 4 shows, running Algorithm 2 on the MCS tree $T(H_6)$ in Figure 1d produces $G^{\equiv}(H_6)$ identical to $G^{\equiv}(H_6^*)$ in Figure 4c. Algorithm 2 does this in time linear in the size of the line graph. We give the formal proof below.

Theorem 26 *Given the line graph L and an MCS tree T_r of an α -acyclic hypergraph H , Algorithm 2 returns an equivalent graph $G^{\equiv}(L)$ in time $\mathcal{O}(|L|)$.*

Proof. Let $H^* = (X^*, R, \chi^*)$ be an equivalent hypergraph derived from $H = (X, R, \chi)$ by vertex duplication. One can readily check that for any non-tree edge e of T , the conditions of each if statement in Algorithm 2 hold under weight function ω of $L(H)$ if and only if they hold under weight function ω^* of $L(H^*)$ by Lemma 48. Therefore Algorithm 2 returns an equivalent graph $G^{\equiv}(L)$.

The construction of each data structure $d(\cdot)$, $\text{LA}(\cdot)$ and $\text{LCA}(\cdot)$ can be done in $\mathcal{O}(m+n)$ [3, 4]. Execution of the rest of the algorithm also takes time $\mathcal{O}(m+n)$, as it requires examining each non-tree edge to slide it directly to its LCA edges in the MCS tree. For each non-tree edge, the lookup of at most two LCA edges takes time $\mathcal{O}(1)$, as does the comparison of weights and the direct sliding transformation. The overall runtime is thus $\mathcal{O}(|L|)$. \square

Given Algorithm 2, we now have a systematic way of enumerating join trees of an α -acyclic hypergraph.

Starting from the line graph $L(H)$ of an α -acyclic hypergraph H , we first reconstruct H and apply Algorithm 1 to obtain an MCS tree T . Using $L(H)$ and T as input, Algorithm 2 constructs an equivalent graph of $L(H)$. Finally, a standard spanning tree enumeration algorithm can be applied to this equivalent graph to enumerate all join trees of H . The overall running time of this process is $\mathcal{O}(|L| + |\mathcal{T}(H)|)$.

Theorem 27 *Given the line graph L of an α -acyclic hypergraph H , the join trees of H can be enumerated in time $\mathcal{O}(|L| + |\mathcal{T}(H)|)$.*

Proof. From $L(H)$ we first retrieve query hypergraph H . This is done in time linear in the size of the line graph. Algorithm 1 constructs an MCS tree T in $\mathcal{O}(|H|)$. With the inputs $L(H)$ and T , we can run Algorithm 2 to construct $G^{\equiv}(H)$ in $\mathcal{O}(|L|)$ by Theorem 26. Finally, letting n be the number of vertices and m the number of edges in $L(H)$, the spanning tree enumeration algorithm by Kapoor and Ramesh [24] on $G^{\equiv}(H)$ enumerates all join trees $|\mathcal{T}(H)|$ in $\mathcal{O}(m+n+|\mathcal{T}(H)|) = \mathcal{O}(|L(H)| + |\mathcal{T}(H)|)$. \square

C.2 Missing Proofs in Section 4.3

We prove Theorem 28 below.

Theorem 28 *The join trees of a γ -acyclic H can be enumerated in time $\mathcal{O}(|H| + |\mathcal{T}(H)|)$.*

Proof. By modifying the Algorithm 1 to Algorithm 3, where changes are highlighted in red, we can construct a weighted MCS tree T and an unweighted line graph $L(H)$ in $\mathcal{O}(|H| + |E(L(H))|)$. We simplify Algorithm 2 to Algorithm 4 where the changes are highlighted in red. As γ -acyclicity guarantees that every non-tree edge (r_i, r_j) is an MST edge, we check its LCA l and the weights of LCA edges ω_1, ω_2 to determine how to slide it. If the shallower endpoint is the LCA of the deeper $r_i = l$, we slide it to be parallel with the only LCA edge as in Line 6. Otherwise, if the LCA edges have different weights, we slide $\{r_i, r_j\}$ to be parallel to the heavier one as in Line 10 and Line 12. Finally if they have the same weight, we slide $\{r_i, r_j\}$ to form a triangle with them as in Line 13. Those four cases are exhaustive.

The number of join trees is lower bounded by the number of non-tree edges in its line graph, namely $|\mathcal{T}(H)| = \Omega(|E(L(H))| - |E(T)|)$. The total time complexity of enumerating all join trees is $\mathcal{O}(|H| + |E(L(H))| + |\mathcal{T}(H)|) = \mathcal{O}(|H| + |\mathcal{T}(H)|)$. \square

D Missing Proofs in Section 5

Proposition 30 *An α -acyclic hypergraph is Berge-acyclic if and only if it is linear.*

Proof. Fagin [15] showed that every Berge-acyclic hypergraph is linear. We prove the other direction by contradiction, assuming that H is α -acyclic and linear. We also assume that there is a Berge cycle $(r_0, x_0, \dots, r_{k-1}, x_{k-1})$ of length $k \geq 2$. Without loss of generality, we suppose r_1 is the first hyperedge on the cycle to be removed by GYO reduction. Then r_1 's parent r_p must satisfy $r_p \cap r_1 \supseteq (r_1 \cap \bigcup_{i \in [k] \wedge i \neq 1} r_i) = \{x_0, x_1\}$, contradicting H 's linearity. \square

Lemma 33 *The line graph L of a Berge-acyclic hypergraph H is a block graph.*

Proof. A Berge-acyclic hypergraph is γ -acyclic [15]. Zhu proves that the line graph of a γ -acyclic hypergraph is chordal [49]. We now prove that the line graph is diamond-free by showing that every diamond must be part of a K_4 . Assuming a diamond (r_1, r_2, r_3, r_4) in L without the edge (r_1, r_3) , it can be treated as two triangles (r_1, r_2, r_4) and (r_2, r_4, r_3) sharing an edge (r_2, r_4) . We now consider the possible edge labels, $r_i \cap r_j$, in each triangle:

Case 1: If all edges are labeled with distinct variables, each triangle forms a Berge-cycle, contradicting the Berge-acyclicity.

Case 2: If two edges are labeled with the same variable, all three vertices share that variable. The third edge must be labeled with the same variable.

Since the two triangles share the edge (r_2, r_4) , all their edges are labeled with the same variable x , namely $x \in r_1 \cap r_2 \cap r_3 \cap r_4$. There has to be an edge $(r_1, r_3) \in E(L)$, contradicting the assumption that (r_1, r_2, r_3, r_4) is a diamond. \square

Lemma 49. *Let T_r be the canonical join tree rooted at r of a Berge-acyclic hypergraph. Along its root-to-leaf path r, r_1, \dots, r_k , each pair of adjacent vertices shares a distinct variable.*

Proof. Let $r_0 = r$, Proposition 30 requires that $|r_{i-1} \cap r_i| = 1$ for all $i \in [k]$. By Theorem 35, the path r_0, \dots, r_k is the shortest path from r_0 to r_k . The variable shared by two adjacent vertices r_{i-1}, r_i is the edge label $r_{i-1} \cap r_i$. The edge labels along the path are all distinct. Otherwise, two consecutive edges with the same label allow for shortening the path by removing either, contradicting the shortest path property. Two non-consecutive edges with the same label violate the running intersection property. \square

Theorem 36 *Given a Berge-acyclic hypergraph H , Algorithm 1 from $r \in R(H)$ constructs the canonical tree $T_r(H)$.*

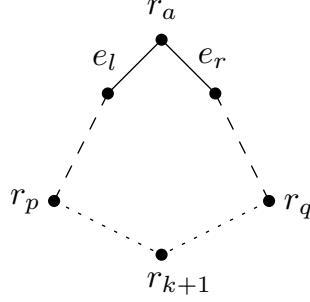


Figure 7: Possible cases for how the relation r_{k+1} attaches to the partial join tree G_k during the construction of the canonical join tree by Maximum Cardinality Search.

Proof. Denoting the edges labeled by Algorithm 1 as $r = r_1, \dots, r_{n=|R(H)|}$, when Algorithm 1 has labeled the first $i \in [n]$ hyperedges, we consider a graph G_i whose vertices are the labeled hyperedges $\{r_1, \dots, r_i\}$ and edges are the parent-child relationships $\{\{r_2, p(r_2)\}, \dots, \{r_i, p(r_i)\}\}$.

To prove the claim by induction, we show a loop invariant that G_i is a subtree of T_r for all $i \in [n]$. An empty graph is trivially a subtree of any graph. When the root r is labeled as r_1 , this graph of a single vertex is also trivially a subtree of T_r . Assuming G_k is a subtree of T_r , Algorithm 1 proceeds to label r_{k+1} and assigns its parent $p(r_{k+1}) = r_{p \leq k}$. Suppose for the sake of contradiction that r_{k+1} has a different parent r_q in T_r . Let us consider the possible relationships between r_p and r_q . First, r_q cannot be a descendant of r_p , otherwise there would be a shorter path from r_{k+1} to the root r going through r_p instead of r_q , violating Theorem 35. Suppose r_q is an ancestor of r_p . By the running intersection property, $r_q \cap r_{k+1}$ must contain $r_p \cap r_{k+1}$, but that would imply every vertex in $r_p \cap r_{k+1}$ is already marked by r_q and the algorithm would not have assigned r_p as the parent of r_{k+1} . Therefore, r_p and r_q are not descendants of each other. Denote their lowest common ancestor in T_r as $r_a = \text{LCA}(r_p, r_q)$. Let $\lambda(r_p, r_q) = \{e_l, e_r\}$ as shown by the solid lines in Figure 7, P_p be the path from r_a to r_p excluding e_l and P_q be the path from r_a to r_q excluding e_r as shown by the dashed lines in Figure 7.

By Lemma 49, the variables on P_p and P_q are all distinct. Consider the following possible cases where $\chi^l = \bigcap_{r \in \rho(e_l)} r$ and $\chi^r = \bigcap_{r \in \rho(e_r)} r$:

Case 1: If $r_{k+1} \cap r_p = r_{k+1} \cap r_q$ and $\chi^l = \chi^r$, then P_p, P_q form a Berge-cycle.

Case 2: If $r_{k+1} \cap r_p = r_{k+1} \cap r_q$ and $\chi^l \neq \chi^r$, then P_p, r_a, P_q form a Berge-cycle.

Case 3: If $r_{k+1} \cap r_p \neq r_{k+1} \cap r_q$ and $\chi^l = \chi^r$, then P_p, P_q, r_{k+1} form a Berge-cycle.

Case 4: If $r_{k+1} \cap r_p \neq r_{k+1} \cap r_q$ and $\chi^l \neq \chi^r$, then P_p, r_a, P_q, r_{k+1} form a Berge-cycle.

All possible cases above contradict Berge-acyclicity, therefore, Algorithm 1 must correctly assign r_p as the parent of r_{k+1} . \square

E Missing Algorithm and Proofs in Section 6

Theorem 38 *A query is γ -acyclic if and only if every connected left-deep linear join plan for the query is the reverse of a GYO-reduction order.*

We first prove “only if” direction of Theorem 38 by contradiction, assuming a query plan with a γ -cycle that is not the reverse of a GYO-reduction order.

Proposition 50. *If a left-deep linear plan is not the reverse of a GYO-reduction order, then there exists a relation r_i that has no parent among $\{r_{j < i}\}$, formally $\neg \exists r_{j < i} : (r_i \cap \bigcup r_{j < i}) \subseteq r_j$. We call r_i an orphan, and denote it as \hat{r}_i .*

We introduce an ordering on relations to compare them in a plan.

Definition 51. *Given a relation r , we write $r_1 \geq_r r_2$ if $r_1 \cap r \supseteq r_2 \cap r$.*

We first observe that \geq_r is a preorder (reflexive and transitive), but not a partial order (antisymmetry may fail). Moreover, if there is a greatest element r_p with respect to \geq_r , i.e., $\forall r' : r_p \geq_r r'$, then r_p is a parent of r . Those observations lead to the following.

Lemma 52. *Let \hat{r} be a maximal relation with respect to $\geq_{\hat{r}}$ for an orphan \hat{r} , meaning $\neg\exists r : r >_{\hat{r}} \hat{r}$, then there is a relation \bar{r} incomparable with \hat{r} , i.e., $\hat{r} \not\geq_{\hat{r}} \bar{r}$ and $\bar{r} \not\geq_{\hat{r}} \hat{r}$.*

Proof. If every relation is comparable with \hat{r} , the maximal relation \hat{r} is the greatest element with respect to $\geq_{\hat{r}}$, thus a parent of \hat{r} . It contradicts the fact that \hat{r} is an orphan. \square

Given a pair of incomparable relations \hat{r} and \bar{r} with respect to \hat{r} , we can find at least one variable in each relation that does not appear in the other relation but appears in \hat{r} , namely $\exists \bar{x} \in \bar{r} \cap \hat{r} : \bar{x} \notin \hat{r}$ and $\exists \hat{x} \in \hat{r} \cap \hat{r} : \hat{x} \notin \bar{r}$. We say that \hat{x} is a *dangling variable* of \hat{r} and \bar{x} is a *dangling variable* of \bar{r} . We are now ready to prove Theorem 38. Our strategy is to derive a γ -cycle from any orphan relation, leading to a contradiction.

Proof of Theorem 38. To prove the “only if” direction by contradiction, we assume that a connected left-deep linear plan for a γ -acyclic query is not the reverse of a GYO-reduction order. By Proposition 50, there is an orphan \hat{r} in the plan. Assuming that \hat{r} is the t -th relation in the plan, we now consider the prefix of the plan up to \hat{r} , namely $\text{pre}(t) = r_1, \dots, r_{t-1}, \hat{r}$. Let $\hat{r} \in \text{pre}(t-1)$ be a maximal relation with respect to $\geq_{\hat{r}}$, Lemma 52 guarantees an $\bar{r} \in \text{pre}(t-1)$ incomparable with \hat{r} . There are at least two dangling variables $\bar{x} \in \bar{r} \cap \hat{r} \setminus \hat{r}$ and $\hat{x} \in \hat{r} \cap \hat{r} \setminus \bar{r}$.

Because the shorter prefix $\text{pre}(t-1) = r_1, \dots, r_{t-1}$ is connected, there is a path in $\text{pre}(t-1)$ between \hat{r} and \bar{r} . The incomparability guarantees that $\bar{r} \cap \hat{r} \not\subseteq \hat{r} \cap \hat{r}$. We can choose a shortest path in $\text{pre}(t-1)$ that connects a variable in $\hat{r} \cap \hat{r}$ to a variable in $\hat{r} \setminus \hat{r}$. We now consider two exhaustive cases where each yields a γ -cycle:

Case 1: If the shortest path consists of a single relation \tilde{r} , $\tilde{r} \not>_{\hat{r}} \hat{r}$ (because \hat{r} is maximal) and $\tilde{r} \not\leq_{\hat{r}} \hat{r}$ (because \tilde{r} contains some variable in $\hat{r} \setminus \hat{r}$). Therefore, \tilde{r} is incomparable with \hat{r} . Since $\tilde{r} \cap \hat{r} \neq \emptyset$, $\hat{r}, \hat{x}, \hat{r}, \tilde{r}, \tilde{r}, \tilde{x}$ form a γ -cycle where $\hat{x} \in \hat{r} \cap \hat{r}$, $\tilde{x} \in \tilde{r} \cap \hat{r}$, and $\tilde{x} \in \tilde{r} \cap \hat{r}$.

Case 2: If the shortest path consists of at least two relations, let $r_0 = \hat{r}$, and the path be r_1, \dots, r_{k-1} . We now show that r_0, r_1, \dots, r_{k-1} form a *pure cycle* where each r_i is only adjacent to $r_{(i-1) \bmod k}$ and $r_{(i+1) \bmod k}$. First, if a $r_{i \in [2, k-2]}$ on the path is adjacent to a $r_{j \in [1, k-1]}$ other than its neighbors r_{i-1} or r_{i+1} , the path can be shortened, contradicting the shortest path assumption. Second, if the relation $r_{i \in [2, k-2]}$ is adjacent to r_0 , then r_i must contain variables in either $r_0 \cap \hat{r}$ or $r_0 \setminus \hat{r}$. We can shrink the path to r_1, \dots, r_i or r_i, r_{k-1} , which also contradicts the shortest path assumption. Therefore, each $r_{i \in [0, k-1]}$ on the cycle is only adjacent to $r_{(i-1) \bmod k}$ and $r_{(i+1) \bmod k}$. The cycle r_0, r_1, \dots, r_{k-1} is a pure cycle of length $k \geq 3$. A pure cycle is always a γ -cycle.

This concludes the proof for the “only if” direction.

Now we prove the “if” direction by contrapositive, namely that if a query contains a γ -cycle, then there exists a connected left-deep linear plan that is not the reverse of a GYO-reduction order. Let the γ -cycle of length $k \geq 3$ be $r_0, \dots, r_i, \dots, r_{k-1}$ where $r_{i \in [1, k-2]}$. By definition, there is an x_{i-1} appearing exclusively in r_{i-1} and r_i and an x_i appearing exclusively in r_i and r_{i+1} . We modify the original plan by moving r_i after r_{k-1} such that $r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_{k-1}, r_i$ becomes part of the new query plan.

The prefix $\text{pre}(k-1) = r_0, \dots, r_{i-1}, r_{i+1}, \dots, r_{k-1}$ is connected. Therefore, the new query plan is connected. However, r_i has no parent, because no relation in $\text{pre}(k-1)$ contains both x_{i-1} and x_i . This implies the plan is not the reverse of a GYO-reduction order. \square

F Notation Table

Variable	Definition
H, X, R	hypergraph, vertex set, hyperedge set
$\chi : R \rightarrow 2^X$	Incidence function mapping each hyperedge to its vertices.
$X(H), R(H)$	Vertex set and edge set of H .
$ H $	Size of hypergraph: $ H = \sum_{r \in R} r $.
$H _x$	Neighborhood of x in H : set of hyperedges containing x .
H^*, X^*, χ^*	Equivalent hypergraph, its vertices, and incidence function
f	A hypergraph homomorphism (pair (f_X, f_R)).
$H_1 \rightarrow H_2, H_1 \twoheadrightarrow H_2$	Homomorphism from H_1 to H_2 , strong homomorphism from H_1 to H_2 .
G, R, E	(Multi-)graph, nodes (corresponding to hyperedges), edges
$\rho : E \rightarrow 2^X, \omega : E \rightarrow \mathbb{N}$	Incidence function mapping each edge to its endpoints, weight function
$R(G), E(G)$	Vertex set and edge set of G .
$G^\equiv, \rho^\equiv, T^\equiv$	Equivalent graph (Def. 15), its incident function, and a spanning tree
$G \leq_T G'$	G can be slid into G' (Def. 16).
$L(H), L, \chi(e), \omega(e)$	Line graph of H , edge label $\chi(\{r_1, r_2\}) = r_1 \cap r_2$, weight $\omega(e) = \chi(e) $.
$ L $	Size of line graph: $ L = \sum_{e \in E(L)} \omega(e)$.
$G _x$	Subgraph of $G \subseteq L(H)$ induced by $H _x$.
T^G, T^M	Join tree generated by GYO and MCS, resp.
T, T_r	Unrooted and rooted (at r) join tree.
\hat{T}, \hat{T}_r	A monotonic weight join tree (Def. 20), one rooted at r .
$\mathcal{T}(L(H)), \mathcal{T}(H)$	Set of (unrooted) join trees of $L(H) / H$.
$d(T_r, r_i)$	Depth of node r_i in rooted tree T_r .
r_p	The parent of r_i in a GYO reduction order.
$c(r), p(e), s(e)$	Children nodes of r , parent edge of e , sibling edges of e
$\text{LCA}(r_i, r_j), \text{LA}(\cdot)$	Lowest common ancestor and level-ancestor.
$\lambda(e)$	LCA edges of a non-tree edge $e = (r_i, r_j)$.
Δ	Number of duplications needed to enforce monotonicity.

Table 2: Notation introduced in sections 3–4.