Subgraph Matching: A New Decomposition Based Approach

Qiyan Li
The Chinese Univ. of Hong Kong
Hong Kong, China
qyli@se.cuhk.edu.hk

Jeffrey Xu Yu
The Chinese Univ. of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

Zongyan He
The Chinese Univ. of Hong Kong
Hong Kong, China
zyhe@se.cuhk.edu.hk

ABSTRACT

We study the subgraph matching problem, which is to find all subgraph isomorphisms of a given pattern graph p in a data graph G. Traditional approaches typically use a backtracking search approach or worst-case optimal join, both of which directly operate on p. In this paper, we revisit the tree decomposition based approach. For a complex pattern graph p, we find its optimal tree decomposition T based on the fractional hypertree width, where a node in T represents a subgraph of p, which is also called a bag, and a node in p may appear in multiple bags in T. The tree decomposition based approach initially computes and materializes the matches of subgraphs specified by the bags, then treats these matches as new relations and employs an acyclic join to compute the matches of p itself. However, previous approaches fail to integrate the tree decomposition with effective join attribute orders, and conversely, previous join attribute ordering approaches do not consider the need to share computations in multiple bags. Additionally, the materialization strategies in previous tree decomposition based approaches can lead to high computation costs. In this paper, we propose a new subgraph matching algorithm ASDMatch (Adaptive Shared Decomposition-based matching). We propose a new dynamic programming approach that finds optimal attribute orders for each bag based on a cost model that incorporates the computation sharing. Furthermore, we introduce a new adaptive materialization strategy to reduce the computation cost. We confirmed that our ASDMatch outperforms state-of-the-art algorithms and can process many challenging queries that previous algorithms can not finish within the time limit.

PVLDB Reference Format:
Qiyan Li, Jeffrey Xu Yu, and Zongyan He. Subgraph Matching: A New Decomposition Based Approach. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/magic62442/ASDMatch.

1 INTRODUCTION

Subgraph matching is one of the fundamental problems in graph analysis and graph database systems. Given a large data graph G and a pattern graph p, subgraph matching is to find all subgraphs of G that are isomorphic to p. This problem has a wide range of applications such as motif discovery [58], fraud detection [42], question answering [53], and functional prediction [6]. The challenge arises from the fact that the subgraph isomorphism problem is NP-hard.

Most of the existing works follow a *filtering-ordering-enumerating* framework [49, 64]. Here, filtering is to eliminate data nodes and edges that cannot be potential matches. A data structure known as the candidate set is used to maintain candidate data nodes/edges for each query node/edge. Various techniques have been proposed to tighten this data structure [4, 5, 19, 20, 29, 51, 65]. The ordering involves selecting a total order of pattern graph nodes. Most previous works use some heuristics to select the node order for a single query, such as prioritizing attributes with larger degrees and fewer candidates [6, 21, 51]. In [38], a cost model is proposed for node orders, along with a dynamic programming approach to optimize the cost. For the enumerating phase, most existing works directly process the pattern graph p as a single query. There are two main categories in the enumeration-based approaches, namely the exploration-based approaches and join-based approaches. The exploration-based approaches [5, 20, 26, 29, 35, 45] follow the classic Ullmann's backtracking algorithm [55]. For join-based approaches, while some earlier works [31, 32, 54] use binary joins, recent approaches [2, 38, 51] use the worst-case optimal join algorithm (WCOJ) [39]. Ullman's backtracking and WCOJ are essentially equivalent for subgraph matching under certain assumptions [51].

As surveyed in [64], in the literature, the focus of subgraph matching research [3, 9, 24, 26, 29, 35, 51] is on enhancing backtracking searches for a query p. They accelerate the enumeration by pruning futile search branches, identifying symmetric search branches or reducing backtracking levels. In this paper, we focus on decomposition-based enumeration. Here, a decomposition-based approach [31, 32, 40, 54], instead of processing a pattern graph as a single query, divides the query into multiple sub-queries $\{p_i\}$ and assembles the sub-queries to get the result of the query p. A powerful tool to decompose a query is tree decomposition [14]. For a complex pattern p, tree decomposition produces a tree-shaped pattern, T, where a tree node in T, also called a bag, represents a subgraph of p, and a node in p may appear in multiple bags in T. EmptyHeaded [2] uses tree decomposition to process subgraph queries. We explore such tree decompositions in this paper. We address the issues in EmptyHeaded and develop a new tree decomposition based approach that is significantly faster than the state-of-the-art enumeration-based approaches for subgraph matching. The first issue of EmptyHeaded is its inefficient computation of a tree decomposition with good quality. We have proposed a new algorithm to compute a good tree decomposition efficiently [22]. Second, it does not select good attribute orders. A simple solution is to use existing approaches like [38] to find an optimal attribute order. However, in the context of tree decomposition, we need to

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

process multiple bags, where some computations can be shared. The attribute order of one bag affects not only the computation of itself, but also the computation in other bags that contain common attributes. To address this issue, we optimize the attribute orders of all bags together, taking the effect of computation sharing into consideration. We propose a dynamic programming approach that efficiently finds the optimal attribute orders with the minimum overall computation costs. Third, a tree-decomposition-based approach needs to materialize subgraph matches of bags in memory. To materialize all such matches is not realistic for large graphs. Some recent works on local subgraph counting propose new strategies to only materialize some attributes in memory [34, 63]. However, they choose materialized attributes before executing the query, which does not make full use of the memory and can increase computation costs. We study an adaptive materialization strategy to choose materialized attributes at runtime and materialize as much as possible, which is much more efficient than the strategies in [34, 63].

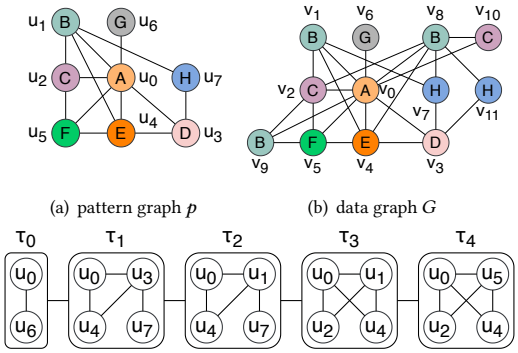
Main Contributions: First, we revisit the tree decomposition based approach for subgraph matching. We identify and address its issues and explore this framework. Second, we study a new optimization problem that minimizes the cost of computing all bags in the tree decomposition, where we consider both the computation cost in a bag and the cost that can be shared across bags. It is challenging to optimize as the two costs are interrelated. We discuss the optimal substructure property and give a dynamic programming approach to address it. Third, we propose a new adaptive approach to materialize subgraph matches of bags, which makes full use of the memory and reduces the computations in existing solutions [34, 63]. Fourth, we compare our ASDMatch with the state-of-the-art subgraph matching algorithms in 8 real large data graphs. ASDMatch outperforms existing algorithms significantly. ASDMatch can process many hard queries that previous algorithms can not, and in three large data graphs, it is the only algorithm that can compute all pattern graphs within the time limit.

Organizations. We give preliminaries and the problem statement in Section 2. In Section 3, we discuss the existing tree-decomposition-based approaches and outline our new algorithm. We study a new query optimization problem and propose a new dynamic programming algorithm to solve it in Section 4 and propose a new adaptive multi-join algorithm in Section 5. We discuss related works in Section 6. We conduct comprehensive experimental studies and report the results in Section 7, and conclude this paper in Section 8.

2 PRELIMINARIES

We model a graph as a simple labeled undirected graph $G = (V, E, L, \Sigma)$. Here, V is a set of nodes, E is a set of undirected edges, Σ is a set of labels, and L is the mapping function that maps a node $u \in V$ to a label denoted as $L(u)$. We denote neighbors of node u in G as $N(u) = \{v | (u, v) \in E\}$. There are no self-loops or multiple edges between two vertices.

Given a data graph $G = (V, E, L, \Sigma)$ and a pattern graph $p = (V_p, E_p, L, \Sigma)$. A homomorphism of p to G is a function $f : V_p \mapsto V$ such that (1) for every $u \in V_p$, $L(u) = L(f(u))$, and (2) for every $(u, v) \in E_p$, $(f(u), f(v)) \in E$. A subgraph isomorphism of p to G is a homomorphism of p to G under the condition that f is an injective function, where $f(u) \neq f(v)$ for any pair of u and v in V_p if $u \neq v$. A homomorphism (or subgraph isomorphism) function f of p induces a subgraph $G_f = (V_f, E_f, L, \Sigma)$ in G, where V_f is the set of nodes, $f(u)$, for every u in V_p, and E_f is the set of edges, $(f(u), f(v))$, for every edge (u, v) in E_p. We say G_f is a subgraph matching of p to G by subgraph isomorphism if f is a subgraph isomorphism function.



(a) pattern graph p (b) data graph G

(c) tree decomposition T

Figure 1: A pattern graph, a data graph, and tree decomposition

Problem Statement: In this work, we study the subgraph matching problem. Given a pattern graph p and a data graph G, subgraph matching returns all subgraph isomorphisms of p to G. As we will discuss later, our techniques can be extended to handle general cyclic join queries.

Example 2.1: Fig. 1 shows an example of the subgraph matching problem. We show a pattern graph p in Fig. 1(a) and a toy data graph G in Fig. 1(b). Here, nodes with different labels are distinguished by colors. There are three subgraph isomorphisms of p to G, i.e. $f_1 = \{u_0 \mapsto v_0, u_1 \mapsto v_1, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4, u_5 \mapsto v_5, u_6 \mapsto v_6, u_7 \mapsto v_7\}$, $f_2 = \{u_0 \mapsto v_0, u_1 \mapsto v_8, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4, u_5 \mapsto v_5, u_6 \mapsto v_6, u_7 \mapsto v_7\}$, and $f_3 = \{u_0 \mapsto v_0, u_1 \mapsto v_8, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4, u_5 \mapsto v_5, u_6 \mapsto v_6, u_7 \mapsto v_{11}\}$.

3 A TREE DECOMPOSITION APPROACH

Tree decomposition is a powerful tool [14]. With a bound by tree decomposition, deciding whether a graph G has a subgraph isomorphic to a pattern graph p becomes fixed-parameter tractable [12, 36]. EmptyHeaded [2] is the first graph system that uses tree decomposition for subgraph matching, and tree decomposition is also used in local subgraph counting [34, 63]. Below, we discuss hypergraph, tree decomposition, fractional hypertree decomposition used in EmptyHeaded, its issues, possible solutions, and challenges.

A hypergraph is defined as $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is a set of hyperedges, where a hyperedge $e \in \mathcal{E}$ is a subset of $\mathcal{V}$. A pattern graph $p = (V_p, E_p)$ is a special hypergraph where every hyperedge only has two nodes.

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, a tree decomposition of $\mathcal{H}$ is a tree $T = (V_T, E_T)$. We use τ_i (simply τ) to denote a tree node in V_T, where τ_i maintains a nonempty subset of $\mathcal{V}$, called a bag and denoted as $\mathcal{V}(\tau_i)$ ($\subseteq \mathcal{V}$), with which an induced subgraph of $\mathcal{H}$ can be constructed, denoted as $\mathcal{H}(\tau_i)$. We say a node v in $\mathcal{V}$ appears in τ_i if $v \in \mathcal{V}(\tau_i)$. The three conditions on T are as follows. (1) Every

node in $\mathcal{H}$ is covered by T such that $\mathcal{V} = \bigcup_{\tau_i \in \mathcal{V}_T} \mathcal{V}(\tau_i)$. (2) Every edge in $\mathcal{H}$ is covered by T such that for every edge $(u, v) \in \mathcal{E}$, both u and v appear in at least one τ_i. (3) Nodes in T are connected if they all contain a pattern graph node. That is, if a node $v \in \mathcal{V}$ appears in both τ_i and τ_j, then v appears in every τ_k on the path that connects τ_i and τ_j in T. In the following, we use $V_S(\tau_i)$ to denote the nodes in a bag τ_i that also appear in neighbors of τ_i.

A fractional hypertree decomposition (FHD) of a hypergraph $\mathcal{H}$ is (T, γ) where T is the tree decomposition and γ is the function that assigns each bag $\tau_i \in T$ a fractional edge cover $\gamma(\tau_i)$ [16]. The quality of a tree/hypertree decomposition is measured by the size of the bag, which is called the width. The width of a fractional hypertree decomposition (T, γ) is $\max_{\tau_i \in T} \gamma(\tau_i)$. The fractional hypertree width of $\mathcal{H}$, denoted as $\mathsf{fhtw}(\mathcal{H})$, is the minimum width of a FHD of $\mathcal{H}$, and is known as the tightest upper bound of $\mathcal{H}$. With the bound of $\mathsf{fhtw}(p)$ for a pattern graph $p = (V_p, E_p)$, a subgraph matching algorithm over a data graph $G = (V, E)$ can be bounded by $O(|E|^{\mathsf{fhtw}(p)} + OUT)$, where OUT refers to the output size of the subgraph matching [2, 37].

Example 3.1: Consider the pattern graph p with 8 nodes in Fig. 1(a). The tree T by fractional hypertree decomposition for p is shown in Fig. 1(c). In T, there are five tree nodes or bags, τ_i, for $0 \leq i \leq 4$, that represent five subgraphs, $p(\tau_i)$. Every node and edge of the query graph is contained in at least one bag, and for each node, the bags containing that node are connected. For the bag τ_1, nodes u_0, u_4 and u_7 are shared with its neighbor bags, so $V_S(\tau_1) = \{u_0, u_4, u_7\}$.

Below, we discuss how EmptyHeaded [2] handles a complex pattern graph $p = (V_p, E_p)$ over a massive graph $G = (V, E)$ for subgraph matching. EmptyHeaded stores G in edge tables. For each edge (u, u') in p, it can find its edges in a relation $\mathbf{R}(L(u), L(u'))$, where $\mathbf{R}(L(u), L(u')) = \{(v, v') | L(u) = L(v), L(u') = L(v'), (v, v') \in E\}$. Such relations can be preprocessed to filter some data edges that can not appear in a subgraph match [64]. Then EmptyHeaded takes a join approach to process subgraph matching. ❶ It finds a hypertree decomposition T for p with which EmptyHeaded decomposes a complex pattern graph p into several smaller patterns $p(\tau_i)$ for every tree node τ_i in T. In other words, instead of conducting subgraph matching for p over G directly, EmptyHeaded first conducts subgraph matching for every $p(\tau_i)$ over G. The time complexity can be downgraded to $O(|E|^{\mathsf{fhtw}(p)} + OUT)$. ❷ EmptyHeaded processes every pattern graph $p(\tau_i)$ by a WCOJ algorithm [51] over a data graph $G = (V, E)$. Note that a WCOJ algorithm [37] conducts a multiway join following a join attribute order ($\mathcal{JAO}$) in a pattern graph $p(\tau_i)$. Here, $\mathcal{JAO}$ refers to the order of computing all nodes in $p(\tau_i)$. Note that a data graph G is stored in edge relations, where an edge relation has two attributes and stores edges having the same node labels. A survey of WCOJ algorithms can be found in [37]. The output of the WCOJ algorithm for a pattern graph $p(\tau_i)$ is materialized in an edge relation $\mathbf{R}(\tau_i)$. ❸ Given all $p(\tau_i)$ for every tree node τ_i in T are computed, EmptyHeaded computes subgraph matching for p over G by join all $\mathbf{R}(\tau_i)$ using Yannakakis' algorithm [62], which is known the best for processing acyclic join queries.

Example 3.2: A data graph $G = (V, E)$ can be stored in edge tables, where an edge table corresponds to a pair of node labels. Reconsider Example 3.1. The pattern graph $p = (V_p, E_p)$ in Fig. 1(a) has 8 nodes and 13 edges. The edges in G that match the edge (u_0, u_4) in p can be found in a relation $R(A, E)$ as the label of u_0 is A and the label of u_4 is E. For simplicity, we may use an edge relation $R(u_i, u_j)$ to refer to a relation $R(L(u_i), L(u_j))$. There are 13 edge relations to be used for subgraph matching of p over G. Here, the tree T by hypertree decomposition is shown in Fig. 1(c) with 5 tree nodes, τ_0, τ_1, τ_2, τ_3, and τ_4. To process a tree node, τ_i, in T, EmptyHeaded uses a WCOJ algorithm. Take τ_1 as an example. Its pattern graph $p(\tau_1)$ is an induced subgraph of p over $\{u_0, u_3, u_4, u_7\}$ with 4 edges as presented in Fig. 1(c). EmptyHeaded processes $\mathbf{R}(u_0, u_3) \bowtie \mathbf{R}(u_0, u_4) \bowtie \mathbf{R}(u_3, u_4) \bowtie \mathbf{R}(u_3, u_7)$ or more precisely $\mathbf{R}(A, D) \bowtie \mathbf{R}(A, E) \bowtie \mathbf{R}(D, E) \bowtie \mathbf{R}(D, H)$, where $\bowtie$ is a natural join, using a WCOJ algorithm. A WCOJ algorithm is a multiway join algorithm following $\mathcal{JAO}$. Such a $\mathcal{JAO}$ is $u_0u_3u_4u_7$. EmptyHeaded materializes the join result in a relation $\mathbf{R}(\tau_1)$. Finally, EmptyHeaded computes a join query, $R(\tau_0) \bowtie R(\tau_1) \bowtie R(\tau_2) \bowtie R(\tau_3) \bowtie R(\tau_4)$ to get the final result of subgraph matching for p. The last join query is an acyclic join query and is computed by Yannakakis' algorithm [62].

The three main issues: In the literature [64], the approach taken by EmptyHeaded is not an up-to-date algorithm. There are many algorithms that outperform EmptyHeaded significantly. There are some main reasons that the tree decomposition based approach, as presented in EmptyHeaded, is not used for subgraph matching. (**Issue-1**) It is NP-hard to find an optimal tree T by hypertree decomposition for a large pattern graph p [16]. That is, the size of a pattern graph cannot be large, which limits its usage. Therefore, the tree decomposition approach by hypertree decomposition is not the first choice, even though the performance can be bounded. (**Issue-2**) EmptyHeaded computes a pattern graph $p(\tau_i)$ for a tree node in T, using a WCOJ algorithm with a $\mathcal{JAO}$ in $p(\tau_i)$, which is not the optimal. Therefore, EmptyHeaded does not perform well [38, 48]. (**Issue-3**) The relations, $\mathbf{R}(\tau_i)$, for $0 \leq i \leq 4$, can be large to be joined at the end, and will incur high computing cost.

To address **Issue-1**, we proposed a novel branch-&-bound algorithm to compute fractional hypertree decomposition (FHD) [22]. We show that FHD can be computed using a dynamic programming (DP) algorithm, and we give a branch-&-bound algorithm with several upper/lower bounds, which makes our DP algorithm much more efficient. Our branch-&-bound algorithm is an anytime algorithm that can terminate at any time. In other words, we can give a feasible solution within the time limit, and we can give a better solution if we have more time. Togeter with the techniques to reduce the cost of computing some fundamental operations used in FHD computing, we confirm that our algorithm is effective and efficient by testing all 3,648 hypergraphs given in *Hyperbench* [11].

To address **Issue-2**, Mhedhbi and Salihoglu in [38] propose a DP algorithm to compute the cost-based optimal join attribute order for all nodes in a pattern graph p. The DP algorithm can get the optimal for the entire pattern graph p, and can be applied to $p(\tau_i)$ for every tree node τ_i in the tree T by fractional hypertree decomposition. But, the DP algorithm given in [38] cannot be effectively used to compute the optimal node order for the pattern graph p given the hypertree decomposition tree T. That is, it does not necessarily

Algorithm 1: ASDMatch (p, G)

Input: pattern graph p, data graph G
Output: Subgraph isomorphisms of p to G

1 $C \leftarrow$ generate and filter candidate data nodes and edges;
2 $T \leftarrow$ optimal fractional hypertree decomposition of p;
3 $T_A^* \leftarrow \mathsf{DPTree}(C, p, T)$;
4 $\mathsf{ASDJoin}(T_A^*, C)$;

mean optimal for p if every $p(\tau_i)$ has an optimal node order to be computed.

We explain it using an example. Consider Example 3.2. By fractional hypertree decomposition, we have 5 tree nodes, τ_i, for $0 \leq i \leq 4$ in the tree T for a pattern graph p. We can apply the DP algorithm in [38] to obtain the optimal join attribute order for each pattern graph $p(\tau_i)$, and then use a WCOJ algorithm to compute $p(\tau_i)$ over a data graph G. However, it may not be the optimal still, as we repeat computing something that can be possibly shared. For example, we have u_0, u_4 in both $p(\tau_1)$ and $p(\tau_2)$. By reducing the cost for nodes, u_0 and u_4, that can be shared across different tree nodes, say τ_1 and τ_2, we can significantly reduce the cost further. The issue of sharing across different tree nodes in T has not been studied, which we study in this work.

To address **Issue-3**, instead of computing every $p(\tau_i)$ with a WCOJ algorithm followed by computing all the materialized relations $\mathbf{R}(\tau_i)$ in the second step, we propose a new approach that is to compute the first and the second step simultaneously by maximizing the usage of main memory in an adaptive manner.

Our New Tree Decomposition Based Algorithm: We present our new algorithm named ASDMatch in Algorithm 1. First, we compute and filter candidate data nodes and edge relations using an existing approach [5]. Second, to address **Issue-1**, we compute the optimal fractional hypertree decomposition using our branch-&-bound algorithm [22] (line 2). Third, to address **Issue-2**, we find the optimal join attribute order by taking sharing into consideration (line 3). Fourth, to address **Issue-3**, we compute the pattern graph p and $p(\tau_i)$ simultaneously by maximizing the usage of main memory in an adaptive manner.

4 OPTIMIZING $\mathcal{JAO}$ WITH SHARING

In this section, we first introduce the cost model used in [38], and the DP algorithm to find the optimal $\mathcal{JAO}$ in [38] for a single pattern graph p without the issue of sharing. Then, we focus on how to find the optimal $\mathcal{JAO}$ for multiple interrelated pattern graphs $p(\tau_i)$ for all tree nodes, τ_i, in a tree T by fractional hyper tree decomposition for a pattern graph p.

4.1 The Optimal $\mathcal{JAO}$ without sharing

We introduce multiway join algorithms on which WCOJ is designed to process subgraph matching for a given single pattern graph $p = (V_p, E_p)$ over a data graph $G = (V, E)$ which does not have the sharing issue. The core idea of a multiway join is to process an induced subgraph $p_k \subseteq p$ with k nodes by extending an induced subgraph $p_{k-1} \subseteq p_k$ with $k-1$ nodes in a way to join the k-th node that does not appear in p_{k-1} with the subgraph p_{k-1} found following a $\mathcal{JAO}$ until $k = |V_p|$.

The join attribute order, multiway join algorithms, and the bottleneck: Let π be a $\mathcal{JAO}$ and π_i be the i-th node in the $\mathcal{JAO}$ order for $i \geq 1$ for all nodes in a pattern graph $p = (V_p, E_p)$. The length of π is $|V_p|$. Suppose that we find a subgraph p_{k-1} in G following the $\mathcal{JAO}$ $\pi[1..k-1]$. Such a subgraph p_{k-1} found can be represented as $p_{k-1} = f(\pi_1)f(\pi_2)\cdots f(\pi_{k-1})$, where $f(\pi_i)$, for $1 \leq i \leq k-1$, denotes the node $f(\pi_i)$ in G that π_i in p_{k-1} matches. To extend p_{k-1} to p_k, it needs to further join $X\pi_k$ over each edge relation $R(\pi_l, \pi_k)$ if $l \in \pi[1..k-1]$. Let the relation P_l be the relation by projecting π_k from relation $R(\pi_l, \pi_k)$ if its π_l value in $R(\pi_l, \pi_k)$ matches $f(\pi_l)$. As there are possibly multiple P_l projected relations, the k-th match of π_k, indicated by $f(\pi_k)$, in a subgraph $p_k = f(\pi_1)f(\pi_2)\cdots f(\pi_k)$ found is taken from the intersection of all projected relations P_l. The bottleneck of a multijoin algorithm is the cost of the intersection.

Example 4.1: Let a pattern graph p be $p(\tau_1)$ in Fig. 1(c), and suppose the $\mathcal{JAO}$ is $\pi = \pi_1\pi_2\pi_3\pi_4 = u_4u_3u_0u_7$. A match of subgraph p_2 is $f(\pi_1)f(\pi_2) = f(u_4)f(u_3) = v_4v_3$ in G (Fig. 1(b)). To extend p_2 to p_3 it needs to further join $\pi_3 = u_0$. Here, it has two edge relations, $\mathbf{R}(u_3, u_0)$ and $\mathbf{R}(u_4, u_0)$, to find matches. It computes $P_0 = \Pi_{u_0}(\sigma_{u_3=v_3}\mathbf{R}(u_3, u_0)) \cap \Pi_{u_0}(\sigma_{u_4=v_4}\mathbf{R}(u_4, u_0))$ where Π and σ are projection and selection operators. Here, $v_4v_3v_0$ is a match of p_3 as $v_0 \in P_0$.

The cost model and the optimal $\mathcal{JAO}$: We introduce the cost model with which the optimal $\mathcal{JAO}$ can be found by DP for a single pattern graph p in [38]. With the optimal $\mathcal{JAO}$, namely, $\pi = \pi_1\pi_2\cdots$, the cost of intersections in a multijoin algorithm can be significantly reduced. The cost for π_1 is as follows.

$$\text{cost}(\pi_1) = |C(\pi_1)| \tag{1}$$

Here, $C(\pi_i)$ is the set of nodes in G that have the same label $L(\pi_i)$ in the data graph G. Given a partial match p_{k-1} by $\pi[1..k-1]$ for some $k > 1$. To match p_k by extending $\pi[1..k-1]$ to $\pi[1..k]$, for each $(\pi_l, \pi_k) \in E_p$, where $l < k$, the cost estimated is as follows.

$$\mu_{\pi_k}^{\pi_l} = |\mathbf{R}(\pi_l, \pi_k)|/|C(\pi_k)| \tag{2}$$

Let $\mathbb{R}_{k-1}$ be relation storing all partial matches for $\pi[1:k-1]$ and $\mu(\mathbb{R}_{k-1})$ be the estimated cardinality of $\mathbb{R}_{k-1}$. Then the candidates for π_k is computed $\mu(\mathbb{R}_{k-1})$ times. When there is no such $l < k$ such that $(\pi_l, \pi_k) \in E_p$, it estimates it by $C(\pi_k)$ as candidates. The cost is as follows.

$$\text{cost}(\pi_k) = \mu(\mathbb{R}_{k-1})|C(\pi_k)| \tag{3}$$

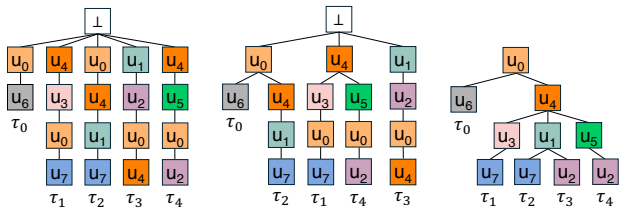
Eq. (1) is a special case of Eq. (3) where $\mu(\mathbb{R}_0) = 1$. When there exists $l < k$ such that $(\pi_l, \pi_k) \in E_p$, the cost for computing each intersection is estimated as the total size of the sets. By multiplying the number of intersections, the cost is as follows.

$$\text{cost}(\pi_k) = \mu(\mathbb{R}_{k-1}) \sum_{l<k, (\pi_l, \pi_k) \in E_p} \mu_{\pi_k}^{\pi_l} \tag{4}$$

The total cost estimated is the sum of cost for each π_i.

$$\text{cost}(\pi) = \sum_{i=1,\cdots|\pi|} \text{cost}(\pi_i) \tag{5}$$

Example 4.2: Consider matching the subgraph in τ_1 in Fig. 1(c). Let $\pi = u_4u_3u_0u_7$. For $\pi_1 = u_4$, $\text{cost}(u_4) = |C(u_4)|$. For $\pi_2 = u_3$, it matches u_3 for every match of $f(u_4)$ in G by traversing the set $\Pi_{u_3}(\sigma_{u_4=f(u_4)}\mathbf{R}(u_3, u_4))$. It estimates the number of such matches



(a) no share (b) simple share (c) optimal

Figure 2: Three attribute trees

as $\mu(\mathbb{R}_1)$, and $\text{cost}(u_3) = \mu(\mathbb{R}_1)\mu_{u_3}^{u_4}$. In a similar manner, $\text{cost}(u_0) = \mu(\mathbb{R}_2)(\mu_{u_0}^{u_3} + \mu_{u_0}^{u_4})$, where $\mu(\mathbb{R}_2)$ is the esitamted cardinality of the subgraph induced by $\{u_4, u_3\}$. For u_7, $\text{cost}(u_7) = \mu(\mathbb{R}_3)\mu_{u_7}^{u_3}$. The total cost is the total cost is $\text{cost}(u_4)+\text{cost}(u_3)+\text{cost}(u_0)+\text{cost}(u_7)$.

The optimal $\mathcal{JAO}$ is the order by minimizing Eq. (5). Let U be a subset of V_p with k nodes, $Perm(U)$ be the set of all permutations of nodes in U, and $\text{cost}^*(U) = \min_{\pi \in Perm(U)} \text{cost}(\pi)$. Furthermore, let $\pi^*(U \setminus \{u\})$ be the optimal permutation of $U \setminus \{u\}$ and $[\pi^*(U \setminus \{u\}), u]$ be a permutation made by putting u at the end of $\pi^*(U \setminus \{u\})$. The cost of matching u is $\text{cost}_{[\pi^*(U\setminus\{u\}),u]}(u)$ An important property of the cost function is that both the cardinality $\mu(\mathbb{R}_{k-1})$ and the set sizes are determined by the set $U \setminus \{u\}$ itself, and is independent of how the set $U \setminus \{u\}$ is permuted. Therefore, for the optimal $\mathcal{JAO}$ of U, denoted as $\pi^*(U)$, where the last node is u, the optimal $\mathcal{JAO}$ for $U \setminus \{u\}$ must be the order that deletes u at the end of $\pi^*(U)$. The DP state transition formula is as follows.

$$\text{cost}^*(U) = \min_{u \in U}(\text{cost}^*(U \setminus \{u\}) + \text{cost}_{[\pi^*(U\setminus\{u\}),u]}(u)) \quad (6)$$

Based on Eq. (6), a DP algorithm to find the optimial $\mathcal{JAO}$ with the minimum cost is proposed [38].

4.2 The optimal $\mathcal{JAO}$ with sharing

To deal with sharing over bags in a tree T by FHD, we use an *attribute tree*, which is defined below.

Definition 4.1: (The Attribute Tree) Given a set of bags $T(\tau) = \{\tau_1, \tau_2, \ldots, \tau_k\}$ in a tree T by FHD, an attribute tree is denoted as $T_A = (V_A, E_A)$ where every bag τ_i is represented as an entire path from the root to the leaf, denoted as $p_i = \text{path}(\tau_i)$. The order of the attributes on p_i is the order from the root to the leaf, denoted as $\pi^{(i)} = \text{ord}(p_i)$. In particular, we use $\text{leaf}(p_i)$ to denote the leaf attribute in p_i, and $p_i[u]$ to denote the prefix of the path p_i from the root to the attribute u. The root can be $\perp$ if no two paths for two bags have a common prefix to share, and can be an attribute shared by all bags.

Example 4.3: Fig. 2 shows three attribute trees for the tree T by FHD in Fig. 1. There are five bags in the tree T. Fig. 2(a) shows an attribute three without sharing among bags. This is an attribute tree where the $\mathcal{JAO}$ order for each bag is optimal from the root to the leaf computed by the DP algorithm given in [38]. Fig. 2(b) shows an attribute three with sharing where two paths for two bags may have some common prefix. Fig. 2(c) shows the best attribute tree with the sharing. In this attribute tree, u_0 is shared by all five bags. Here, the matches of the induced subgraph over $\{u_0, u_4, u_1\}$ can be shared by two bags, τ_2 and τ_3. On the other hand, the induced subgraph over $\{u_0, u_4, u_1\}$ in the attribute tree in Fig. 2(a) needs to be computed twice.

The cost for an attribute tree $T_A = (V_A, E_A)$ is defined as follows.

$$\text{cost}(T_A) = \sum_{u \in V_A} \text{cost}(u) \quad (7)$$

where $\text{cost}(u)$ is either Eq. (3) if u is the first attribute in a bag or Eq. (4) otherwise. The optimal attribute tree $T_A = (V_A, E_A)$ is the one with the minimum cost among all possible attribute trees as follows.

$$\begin{aligned} &\text{minimize} \quad \text{cost}(T_A) = \sum_{u \in V_A} \text{cost}(u) \\ &\text{subject to} \quad T_A \in \mathbf{T} \end{aligned} \quad (8)$$

where $\mathbf{T}$ is the set of all possible attribute trees for a given hypertree decomposition T.

Hardness of the Optimization Problem: The optimization problem to minimize Eq. (8) is a harder problem compared to the problem of optimizing the attribute order for a single bag [38], as it introduces the sharing issue. We discuss it below. ❶ Given a set of bags $\tau = \{\tau_1, \tau_2, \ldots, \tau_k\}$ for a tree T by FHD, the number of possible attribute trees to consider is extraordinarily large. This number grows exponentially with both the number of bags and the number of attributes within each bag. Specifically, merely considering the combinations of attributes orders within each bag, i.e, $\pi^{(1)}, \pi^{(2)}, \ldots, \pi^{(k)}$, entails up to $\Pi_{i=1}^{k}|\tau_i|!$ possible combinations, where $\pi^{(i)}$ corresponds to a bag τ_i and $|\tau_i|$ is the number of attributes in τ_i. ❷ Constructing the optimal attribute tree, T_A, requires to consider both optimization of the attribute order within each bag and optimization of the sharing across bags. And the two issues are interdependent. In other words, the optimization of the attribute tree T_A needs to consider both simultaneously, and cannot be done in two steps, that is, either optimizing the attribute order for each bag followed by optimizing sharing or optimizing the sharing for all bags followed by optimizing attribute order for each bag.

The DP algorithm: we propose a dynamic programming (DP) algorithm to optimize the attribute tree T_A. We first discuss two properties of the optimal attribute tree, namely, the prefix independent property and the subset property, and then explore the optimal substructure based on the two properties.

Let π'' and π' be two attribute orders. We say π'' is a *prefix* of π' denoted as: $\pi'' \preceq \pi'$. The longest common prefix is the longest attribute order π satisfies $\pi \preceq \pi''$ and $\pi \preceq \pi'$. We denote the operator of longest common prefix as $\wedge$, i.e., $\pi = \pi'' \wedge \pi'$. For a set of attribute orders $\Pi = \{\pi^{(1)}, \pi^{(2)}, \ldots, \pi^{(k)}\}$, the longest common prefix of Π is $\Lambda_{i=1}^{k}\pi^{(i)}$.

Lemma 4.1: *(**The Prefix Independent Property**) Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_k\}$ be a set of bags, where each τ_i has an attribute order $\pi^{(i)}$. The optimal attribute tree T_A^* on τ, must be shared on the longest common prefix $\Lambda_{\tau_i \in \tau}\pi^{(i)}$ where possible.*

Proof Sketch: See [1]. □

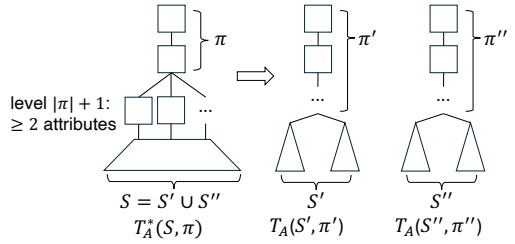


Figure 3: An optimal attribute tree and two subtrees

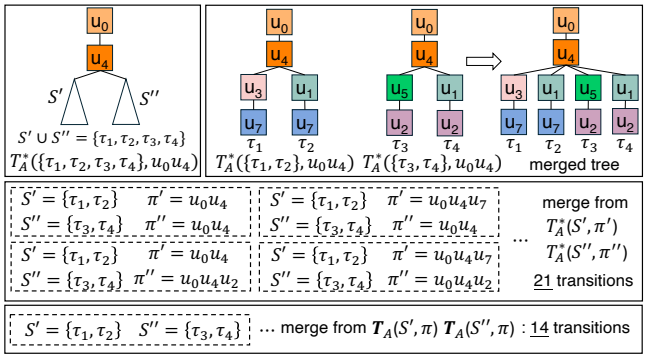


Figure 4: State transition optimization

Lemma 4.2: *(**The Subset Property**) Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_k\}$ be a set of bags, where each τ_i has an attribute order $\pi^{(i)}$. Let $\mathcal{S} \subseteq \tau$. We have $\Lambda_{\tau_i \in \tau} \pi^{(i)} \preceq \Lambda_{\tau_j \in \mathcal{S}} \pi^{(j)}$.*

We omit the proof as it is obvious. The Lemma 4.1 implies a property that makes the attribute order and the attribute sharing irrelevant in the sense that the longest common prefix on the attribute tree can be independent with the remaining attribute order determined by combinations and sharing. The Lemma 4.2 further implies how to construct the optimal attribute tree by considering the longest common prefix if the optimal attribute tree is known on the subset. Thus, we can design the optimal substructure as below.

Below, we use $T_A(\mathcal{S}, \pi)$ to denote an attribute tree T_A, which has a longest common prefix π over a set of bags $\mathcal{S}$.

Lemma 4.3: *Given a set of bags $\mathcal{S}$ and let the attribute T_A for $\mathcal{S}$ has the longest common prefix on $\mathcal{S}$. The optimal attribute tree $T^*_A(\mathcal{S}, \pi)$ can be constructed by merging two optimal attribute trees $T^*_A(\mathcal{S}', \pi')$ and $T^*_A(\mathcal{S}'', \pi'')$, where $\mathcal{S} = \mathcal{S}' \cup \mathcal{S}''$, $\mathcal{S}' \cap \mathcal{S}'' = \emptyset$, and $\pi = \pi' \wedge \pi''$. The cost of $T^*_A(\mathcal{S}, \pi)$ can be computed by:*

$$\text{cost}^*(\mathcal{S}, \pi) = \min_{\substack{\mathcal{S}=\mathcal{S}' \cup \mathcal{S}'', \\ \mathcal{S}' \cap \mathcal{S}''=\emptyset, \\ \pi=\pi' \wedge \pi''}} (\text{cost}^*(\mathcal{S}', \pi') + \text{cost}^*(\mathcal{S}'', \pi'') - \text{cost}(\pi)) \tag{9}$$

Proof Sketch: We prove this lemma by contradiction. Suppose that there exists an optimal attribute tree $T^*_A(\mathcal{S}, \pi)$ with $|\mathcal{S}| \geq 2$ that cannot be constructed by merging two optimal attribute trees. Note that, since π is the longest common prefix, the nodes at level $|\pi_i|+1$ on $T^*_A(\mathcal{S}, \pi)$ must have different attributes, thus $T^*_A(\mathcal{S}, \pi)$ can be divided into two attribute trees $T_A(\mathcal{S}', \pi')$ and $T_A(\mathcal{S}'', \pi'')$ (as shown in Fig. 3). Then, we only need to discuss the optimality of $T_A(\mathcal{S}', \pi')$ and $T_A(\mathcal{S}'', \pi'')$, where $T_A(\mathcal{S}', \pi')$ is on set of bags $\mathcal{S}'$ with the longest common prefix π', and $T_A(\mathcal{S}'', \pi'')$ is an attribute tree on $\mathcal{S}''$ with the longest common prefix π''. Without loss of generality, we assume that $T_A(\mathcal{S}', \pi')$ is not the optimal attribute tree with (S', π'). Then there exists another optimal attribute tree $T^*_A(S', \pi')$ such that $\text{cost}^*(T^*_A(S', \pi')) < \text{cost}^*(T_A(\mathcal{S}', \pi'))$. An attribute tree T' can be constructed by merging $T^*_A(S', \pi')$ and $T_A(\mathcal{S}'', \pi'')$ with cost:

$$\begin{aligned} \text{cost}(T') &= \text{cost}(T^*_A(S', \pi')) + \text{cost}(T_A(S'', \pi'')) - \text{cost}(\pi) \\ &< \text{cost}(T_A(S', \pi')) + \text{cost}(T_A(S'', \pi'')) - \text{cost}(\pi) \\ &= \text{cost}(T^*_A(\mathcal{S}, \pi)) \end{aligned}$$

Therefore, T' is a better attribute tree and $T^*_A(\mathcal{S}, \pi)$ is not optimal, which contradicts the assumption. □

Example 4.4: Fig. 4 shows a state and some state transitions. The tree decomposition is the same as in Fig. 1(c). Here, $\pi = [u_0, u_4]$, $\mathcal{S} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, which aims to find the best attribute tree for the four bags where the longest common prefix is $[u_0, u_4]$. Following Eq (9), to compute $T^*_A(\mathcal{S}, \pi)$, we need to consider all $\mathcal{S}', \mathcal{S}''$ and all pairs of π', π'' for each $\mathcal{S}', \mathcal{S}''$, and merge two subtrees as its attribute tree. Consider $\mathcal{S}' = \{\tau_1, \tau_2\}, \mathcal{S}'' = \{\tau_3, \tau_4\}$. There are two prefixes that can extend π for $\mathcal{S}'$, $[u_0, u_4]$ and $[u_0, u_4, u_7]$, and two in $\mathcal{S}''$, so there are 4 transitions. An example of the transition is shown in the upper right. Overall, there are 21 transitions.

It is sufficient to consider merging two subtrees on the longest common prefix in an iterative manner. All possible cases of sharing will be considered. We explain it using Fig. 3. Suppose that the third attribute of π' is u and u does not appear in π'' but appears in the left subtree of $T_A(S'', \pi'')$. By merging two subtrees iteratively, it is possible that the left subtree of $T_A(S'', \pi'')$ in the form of $T_A(S_u, \pi''u)$, where $S_u \subset S''$ and $\pi''u$ is appending u to π'', can be merged with $T_A(S', \pi')$ in some iteration if it leads to the optimal.

State Transition Optimization: As discussed, for a state $(\mathcal{S}, \pi)$, with $\mathcal{S}$ is divided into $\mathcal{S}'$ and $\mathcal{S}''$. As shown in Fig. 4, for $\mathcal{S}' = \{\tau_1, \tau_2\}, \mathcal{S}'' = \{\tau_3, \tau_4\}$, it is needed to traversal all pairs of permutations, i.e, $\{[u_0, u_4], [u_0, u_4, u_7]\} \times \{[u_0, u_4], [u_0, u_4, u_2]\}$. The pairwise transitions must be considered because we require that the chosen π' on $\mathcal{S}'$ must exactly satisfy $\pi' \wedge \pi'' = \pi$ with the chosen π'' on $\mathcal{S}''$. This strict condition forces us to traverse all attribute orders on $\mathcal{S}'$ and $\mathcal{S}''$ to find all pairs of attribute orders that meet the condition. However, such pairwise traversal would result in a high computational cost of $(|\tau_i|!)^2$ for each transition, which is prohibitively expensive and unsustainable.

To reduce this computational overhead, we consider a weaker condition, namely, $\pi' \wedge \pi'' \succeq \pi$. This weaker condition does not break the optimal substructure property, and the optimal attribute tree can still be constructed by merging two optimal attribute trees that satisfy this condition. Under this weaker condition, the prefixes of the attribute order, π' on $\mathcal{S}'$ and π'' on $\mathcal{S}''$, only need to satisfy $\pi' \succeq \pi$ and $\pi'' \succeq \pi$, which allows for independent traversal. Eq. (9) is rewritten as:

$$\begin{aligned} \text{cost}^*(\mathcal{S}, \pi) = \min_{\substack{\mathcal{S}' \cup \mathcal{S}''=\mathcal{S}, \\ \mathcal{S}' \cap \mathcal{S}''=\emptyset}} (& \min_{\pi' \succeq \pi} \text{cost}^*(\mathcal{S}', \pi') \\ & + \min_{\pi'' \succeq \pi} \text{cost}^*(\mathcal{S}', \pi'') - \text{cost}(\pi)) \end{aligned} \tag{10}$$

Note that $\min_{\pi' \succeq \pi} \text{cost}^*(\mathcal{S}', \pi')$ and $\min_{\pi'' \succeq \pi} \text{cost}^*(\mathcal{S}'', \pi'')$ are

Algorithm 2: DPTree(τ)

Input: A set of bags $\tau = \{\tau_1, \tau_2, \ldots, \tau_k\}$
Output: An optimal attribute tree T_A^* for τ with its cost $\text{cost}^*(\tau)$

1 **foreach** $\tau_i \in \tau$ **do**
2 **foreach** $\pi \in \text{Perm}(\tau_i)$ **do**
3 compute $\text{cost}^*(\{\tau_i\}, \pi)$ using $\text{cost}(\pi)$ (Eq. (5));
4 $T_A^*(\{\tau_i\}, \pi) \leftarrow \text{InitialTree}(\pi)$;
5 **foreach** $\pi' \preceq \pi$ **do**
6 **if** $\text{cost}^*(\{\tau_i\}, \pi) < \textbf{cost}(\{\tau_i\}, \pi')$ **then**
7 $\textbf{cost}(\{\tau_i\}, \pi') \leftarrow \text{cost}^*(\{\tau_i\}, \pi)$;
8 $\mathbf{T}_A(\{\tau_i\}, \pi') \leftarrow T_A^*(\{\tau_i\}, \pi)$;
9 **foreach** $j = 2$ *to* k **do**
10 **foreach** $\mathcal{S} \subset \tau, |\mathcal{S}| = j$ **do**
11 $Attr \leftarrow \cap_{\tau_i \in \mathcal{S}} \tau_i$;
12 **foreach** $\mathcal{S}' \subseteq \mathcal{S}, \mathcal{S}'' = \mathcal{S} \setminus \mathcal{S}', \pi \in \cup_{A \subseteq Attr} \text{Perm}(A)$ **do**
13 $NewCost \leftarrow \textbf{cost}(\mathcal{S}', \pi) + \textbf{cost}(\mathcal{S}'', \pi) - \text{cost}(\pi)$;
14 **if** $\text{cost}^*(\mathcal{S}, \pi) > NewCost$ **then**
15 $\text{cost}^*(\mathcal{S}, \pi) \leftarrow NewCost$;
16 $T_A^*(\mathcal{S}, \pi) \leftarrow MergeTree(\mathbf{T}_A(\mathcal{S}', \pi), \mathbf{T}_A(\mathcal{S}'', \pi))$;
17 **foreach** $\pi \in \cup_{A \subseteq Attr} \text{Perm}(A), \pi' \preceq \pi$ **do**
18 **if** $\text{cost}^*(\mathcal{S}, \pi) < \textbf{cost}(\mathcal{S}, \pi')$ **then**
19 $\textbf{cost}(\mathcal{S}, \pi') \leftarrow \text{cost}^*(\mathcal{S}, \pi)$;
20 $\mathbf{T}_A(\mathcal{S}, \pi') \leftarrow T_A^*(\mathcal{S}, \pi)$;
21 $Attr \leftarrow \cap_{\tau_i \in \tau} \tau_i$;
22 **foreach** $\pi \in \cup_{A \subseteq dAttr} \text{Perm}(A)$ **do**
23 **if** $\text{cost}^*(\tau, \pi) < \text{cost}^*(\tau)$ **then**
24 $\text{cost}^*(\tau) \leftarrow \text{cost}^*(\tau, \pi), T_A^*(\tau) \leftarrow T_A^*(\tau, \pi)$;

independent, where the former depends only on $\mathcal{S}'$ and π', and the latter depends only on $\mathcal{S}''$ and π''. We can precompute and maintain the results of each of them in advance to avoid redundant calculations. We introduce a cost function $\textbf{cost}(\mathcal{S}, \pi)$ to represent it, and $\mathbf{T}_A(\mathcal{S}, \pi)$ accordingly for the attribute tree built on $\textbf{cost}(\mathcal{S}, \pi)$.

$$\begin{aligned} \textbf{cost}(\mathcal{S}, \pi) &= \min_{\pi' \succeq \pi} \text{cost}^*(\mathcal{S}', \pi') \\ \mathbf{T}_A(\mathcal{S}, \pi) &= \arg\min_{\pi' \succeq \pi} \text{cost}^*(\mathcal{S}', \pi') \end{aligned} \tag{11}$$

Here, Eq. (11) holds if we replace $\mathcal{S}'$ and π' with $\mathcal{S}''$ and π''.

Finally, the optimized state transition function is written as:

$$\text{cost}^*(\mathcal{S}, \pi) = \min_{\substack{\mathcal{S}' \cup \mathcal{S}'' = \mathcal{S}, \\ \mathcal{S}' \cap \mathcal{S}'' = \emptyset}} (\textbf{cost}(\mathcal{S}', \pi) + \textbf{cost}(\mathcal{S}'', \pi) - \text{cost}(\pi)) \tag{12}$$

As shown in Fig. 4, using the optimized state transition, when $\mathcal{S}' = \{\tau_1, \tau_2\}, \mathcal{S}'' = \{\tau_3, \tau_4\}$, as we do not have to consider all perfixes extending $\pi = \{u_0, u_4\}$, there is only 1 transtition, whereas there are 4 transitions based on Eq. (9). Overall, the number of transitions is reduced from 21 to 14.

We show our DP algorithm, DPTree, in Algorithm 2, which takes the set of bags, denoted as τ. It first initializes the states for every bag τ_i (lines 1-8). Notices that, for each single bag τ_i, the longest common prefix can only be any permutation of τ_i, i.e, $\pi \in \text{Perm}(\tau_i)$ (line 2). The attribute tree is initialized as a linear tree with the order π, and the cost is initialized as the cost of π (lines 3-4). Next, DPTree calculates the optimal attribute tree and cost for each subset $\mathcal{S} \subseteq \tau$ following Lemma 4.3 and Eq. (12) with increasing the size of $\mathcal{S}$ (lines 9-20). In the process of computing $\mathcal{S}$, the algorithm computes the shared attributes, $Attr$ (line 11). Note that the possible longest common prefix π can only be a permutation of some subset of $Attr$, i.e, $\pi \in \cup_{A \subseteq Attr} \text{Perm}(A)$ (line 12). The algorithm outputs the optimal attribute tree and cost for the whole bag set τ by traversing all

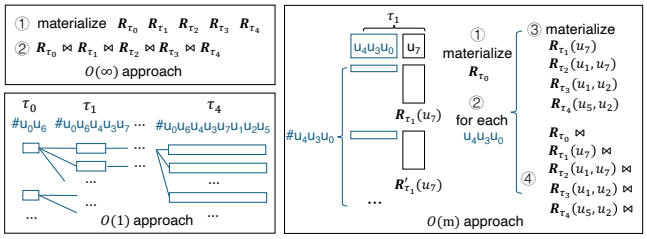


Figure 5: Materialization strategies

permutations of shared attributes (lines 21-24). The time complexity is $O(k \cdot 3^k \cdot (tw(T)+1)! + 2^k \cdot (\text{tw}(T)+1) \cdot (tw(T)+1)!)$, and the space complexity is $O(k \cdot 3^k \cdot (tw(T)+1)!$. The detailed complexity analysis is in [1].

5 ADAPTIVE COMPUTATION

As discussed in Section 3, the last step in EmptyHeaded is to join all the materialized relations, $\mathbf{R}(\tau_1) \bowtie \cdots \bowtie \mathbf{R}(\tau_k)$, if there are k bags (or tree nodes) in the tree T by FHD, or k leaf attributes in the corresponding attribute tree T_A. We illustrate it in the upper left in Fig. 5. The I/O cost to join can be high, in particular, in the case that we cannot hold all such materialized relations in main memory. We call it an $O(\infty)$-memory approach. To avoid such a possible high I/O cost, some techniques used in local subgraph counting can be used [34, 63]. In [63], it takes an $O(1)$-memory approach, which does not materialize bags, and does not make use of the main memory, even if it is available. We illustrated it in the lower left in Fig. 5. For every match of τ_0, it enumerates the matches of τ_1 and produces a match of $\tau_0 \cup \tau_1$, and so on for the remaining bags. As a result, the enumeration cost can be high, as it needs to repeat enumerating all matches of bags, and the number of repetitions is large. In [34], it takes an $O(m)$-memory approach that materializes the last two attributes of each bag if these two attributes are an edge in p or only the last one. Therefore, the szie of materailized matches is bounded by m where $m = |E|$ for a data graph $G = (V, E)$. We illustrated it in the right part in Fig. 5. For $\tau_1, \tau_2, \tau_3, \tau_4$, it does not materialze u_4, u_3, u_0 but materializes other attributes. For τ_1, for a match f of u_4, u_3, u_0, it materializes all matches of u_7 given $f(u_4), f(u_3), f(u_0)$. For another match of u_4, u_3, u_0, it materializes new matches of u_7 and discards old ones. Similarly, it only materializes some attributes in other bags. Then it joins all such materialized tables for all bags. To compute the full result, it traverses all matches of u_4, u_3, u_0 and all such partially materialized tables.

We discuss their costs as follows. Let τ_i be a bag in the attribute tree T_A, and let $n(\tau_i)$ be the attributes of τ_i. In the $O(\infty)$-memory approach, the enumeration cost to materialize the entire bag τ_i is $\#(n(\tau_i))$, where $\#(n(\tau_i))$ is the number of matches of τ_i in the data graph G that needs to enumerate. In the $O(\infty)$-memory approach, the total enumeration cost is $\sum_{\tau_i} \#(n(\tau_i))$ together with c_j, where c_j is the cost of joining all the materialized bags. With the $O(1)$-memory approach, the enumeration cost becomes $\#(\cup_{\tau_i} n(\tau_i))$, and the cost $c_j = 0$. For example, in Fig. 5, the cost for τ_1 is $\#(\tau_0 \cup \tau_1)$=$\#(u_0, u_6, u_4, u_3, u_7)$. With the $O(m)$-memory approach, if two attributes of a bag τ_i is materailized, they are enumerated for $\#(\cup_{\tau_i}(n(\tau_i) - 2))$ times, where $n(\tau_i) - 2$ is the attributes on the

Algorithm 3: ASDJoin (T_A, C)

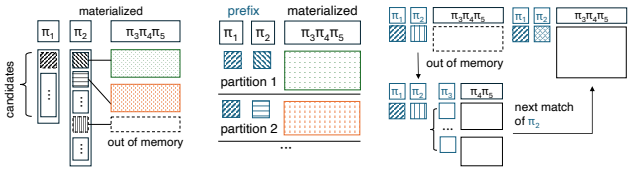
Input: the attribute tree T_A for a tree T with bags $\tau_1, \cdots, \tau_k$, the set of candidates to match C
Output: the result of subgraph matching

1 let $\pi(T_A)$ be the order of preorder tree traversal of T_A;
2 TJoin($\pi_1, C(\pi_1), \emptyset$);
3 output $\mathbf{R}(\tau_1) \bowtie \cdots \bowtie \mathbf{R}(\tau_k)$;
4 **Procedure** TJoin(π_c, I_c, f)
5 mvector $\leftarrow$ MoveOrKeep(π_c);
6 **while** $I_c \neq \emptyset$ **do**
7 $v \leftarrow$ Inext(I_c); $I_c \leftarrow I_c \setminus \{v\}$;
8 $f \leftarrow f \cup \{\pi_c \mapsto v\}$;
9 **if** π_c *is the leaf of* τ_i **then**
10 Materialize($\mathbf{R}(\tau_i)$, mvector, f);
11 $\pi_n \leftarrow$ Tnext(π_c, mvector);
12 **if** $\pi_n \neq \emptyset$ **then**
13 $I_n \leftarrow$ Intersect (π_n, f);
14 TJoin(π_n, I_n, f);
15 **if** *memory is limited and all bags are computed* **then**
16 partialJoin($\pi(T_A)$, mvector, $\mathbf{R}(\tau_1), \cdots, \mathbf{R}(\tau_k)$);
17 $f \leftarrow f \setminus \{\pi_c \mapsto v\}$;
18 $\pi_n \leftarrow$ Tnext(π_c, mvector);
19 **if** $\pi_n \neq \emptyset$ **then**
20 $I_n \leftarrow$ Intersect (π_n, f);
21 TJoin(π_n, I_n, f);

path from the root to the leaf of τ_i excluding the two materialized attributes. For example, the cost for τ_2 is $\#(u_4, u_3, u_0, u_1, u_7)$. Its cost of c_j is smaller compared to that for the $O(\infty)$-memory approach.

In this paper, to maximize the usage of main memory and reduce repetitions in the enumeration, we propose an approach to reduce the enumeration cost significantly. Our approach is to materialize as much as possible. Let prefix_i be the prefix of the path from the root to the leaf for a bag τ_i in T_A, where $\text{prefix}_i \leq n(\tau_i)$. For τ_i, we traverse all matches of prefix_i, and compute a materialized table for each match. When $\text{prefix}_i = 0$, the enumeration cost is the same as the $O(\infty)$-memory approach. When $\text{prefix}_i = n(\tau_i)$, the enumeration cost is the same of the $O(1)$-memory approach. And $\text{prefix}_i = n(\tau_i) - 2$, the enumeration cost is the same of the $O(m)$-memory approach. In general, a smaller prefix_i has a smaller enumeration cost. Although the state-of-the-art $O(m)$-approach bounds the memory usage, the bound is loose and its prefix_i can be large. Both the $O(1)$ approach and the $O(m)$ approach determine prefix_i at the query planning time. We determine the initial prefix_i in the execution and take a novel approach to make prefix_i as small as possible by adjusting prefix_i dynamically.

Our new algorithm, ASDJoin (Adaptive Shared Decomposition-based join), is given in Algorithm 3 to compute a pattern graph $p = (V_p, E_p)$ over a data graph $G = (V, E)$. ASDJoin takes two inputs, the optimal attribute tree T_A constructed for the tree T by FHD assuming that there are k bags in T_A, and the candidate sets C. First, we determine a tree traversal order of T_A, denoted as $\pi(T_A)$, which is a preorder tree traversal (line 1). Next, it calls a TJoin procedure to compute bags with parameters, namely, the first attribute in $\pi(T_A)$ denoted as π_1, its candidate set $C(\pi_1)$, and the partial match which is empty initially. Assume that the results by TJoin are fully or partially



(a) shortest prefix (b) process partitions (c) adaptive prefix

Figure 6: Selecting prefixes

materialized in $\mathbf{R}(\tau_1), \cdots, \mathbf{R}(\tau_k)$, it outputs $\mathbf{R}(\tau_1) \bowtie \cdots \bowtie \mathbf{R}(\tau_k)$ if needed (line 3). Next, we explain TJoin, which focuses on the current attribute π_c in T_A with its candidate set I_c, where f is a partial match that matches all attributes from the root of T_A to π_c. At line 5, MoveOrKeep(π_c) determines if π_c can be a prefix given the size of memory available, and mvector is a vector for all attributes in T_A, which keeps the information on the attributes that are a part of the prefix. At line (6-17), it takes a candidate, v, from I_c by Inext iteratively (line 7), and appends it to the partial match f. If π_c is the leaf of a bag τ_i, we materialize it regarding mvector, if memory is available (line 9-10). To extend it a partial match further, the next attribute of π_c to extend is π_n (line 11). We use a function Tnext that uses mvector to decide which attribute to traverse to, or return $\emptyset$ if it should not go to the next attribute. We will introduce the rules later. When there is an attribute to traverse, an intersection is needed to compute I_n, which is the same as in Leapfrog. With π_n, I_n, and f, we call TJoin recursively. When we go back from the next attribute or there is no next attribute, if we cannot materialize the entire bags, we will check whether we have computed full or partially materialized relations for all bags. If so, we conduct a partial join using currently materialized $\mathbf{R}(\tau_1), \cdots, \mathbf{R}(\tau_k)$ (line 15-16). At line 17, we remove the last match of $\pi_c \mapsto v$ from the current partial match f. After enumerating all candidates in I_c, if Tnext returns the next attribute to traverse, we continue the traversal (line 18-21).

Adaptive materialization: When we find that the memory can not afford to materialize a bag τ_i, we identify a prefix, prefix_i, for τ_i, and attempt to materialize as much as possible for the bag τ_i.

We explain it by illustrating it in Fig. 6(a). Initially, we try to materialize an entire bag, τ_i. Here, assume the attribute order to process τ_i is $\pi_1\pi_2\pi_3\pi_4\pi_5$ following $\pi(T_A)$. At runtime, suppose that we cannot materialize all matches in $\mathbb{R}(\tau_i)$. We will determine a prefix, for example, $\pi_1\pi_2$, and we materialize all the matches by $\pi_3\pi_4\pi_5$ for a single match v_1v_2 in G, if $\pi_1 \mapsto v_1$ and $\pi_2 \mapsto v_2$, as illustrated in Fig. 6(b). Since we can not materialize all $\pi_2\pi_3\pi_4\pi_5$, $\pi_1\pi_2$ is the shortest prefix that we can select. During the run time, there is only one partially materialized table for it. When we are at the second match of $\{\pi_1\pi_2\}$, we delete the first table (green box with dots) and materialize the corresponding table (orange box with vertical dashed lines) for this new match. Suppose that we cannot still materialize it for some v_1v_2, where $\pi_1 \mapsto v_1$ and $\pi_2 \mapsto v_2$ at runtime, we will temporarily enlarge the prefix_i from $\pi_1\pi_2$ to $\pi_1\pi_2\pi_3$. When we find that it can materialize more, we will reduce prefix_i from $\pi_1\pi_2\pi_3$ to $\pi_1\pi_2$, as illustrated in Fig. 6(c).

Traversal of the attribute tree T_A: We explain the traversal in Algorithm 3 using Fig. 7. It uses the query graph and tree decomposition in Fig. 1 and the attribute tree in Fig. 2(b). There are several

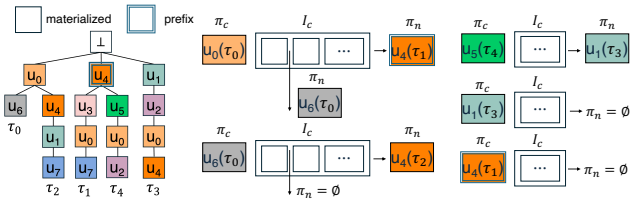


Figure 7: The next attribute in the traversal

bags τ_i in an attribute tree T_A. We process the bags following the preorder tree traversal of T_A. Here, we have 5 bags, and the order to process those bags is τ_0, τ_2, τ_1, τ_4, and τ_3. Suppose that in mvector, the attribute u_4 of the bag τ_1 (i.e., the one below $\perp$) is the prefix, and others are materialized. In Algorithm 3, the traversal is controlled by Tnext called in lines 11 and 18, and we explain the rules of these two cases.

In line 11, we have not traversed all candidates to match π_c, and Tnext goes to the first child or returns $\emptyset$ if π_c is a leaf. In Fig. 7, if we matched the u_0 of τ_0, the next attribute π_n will be the u_6 of τ_0. For the u_6 of τ_0, since there is no child, $\pi_n = \emptyset$ at line 11. It repeats lines 6-17 and materializes all tuples of τ_0 for a given match of u_0.

In line 18, we have traversed all I_c. If π_c is materialized, Tnext goes to the next sibling if there is one. When π_c does not have the next sibling, if the parent is not the prefix, Tnext returns $\emptyset$ to make the traversal go back; if the parent is a prefix, it goes to the next bag, and π_n is its first materialized attribute in $\pi(T_A)$. If π_c is a prefix, $\pi_n = \emptyset$. We explain it in Fig. 7. For u_6 of τ_0, in line 18, $\pi_n = \emptyset$ and it goes back to u_0 of τ_0. For u_0 of τ_0, π_n is its next sibling, i.e. u_4 of τ_1. For the u_5 of τ_4, when we are at line 18, since u_4 is a prefix, we have computed the partial materialization of τ_4, which materializes $u_5u_0u_2$. We need to compute the next partial materialization of the next bag τ_3, so π_n is the u_1 of τ_3. When u_1 of τ_3 comes to line 18, since it does not have the next sibling, $\pi_n = \emptyset$. The algorithm goes back to line 18 in the TJoin of u_5 of τ_4, then line 18 of u_3 of τ_1, then line 14 of u_4 of τ_1. At this time, τ_0 and τ_2 have materialized their full table, τ_1, τ_4, τ_3 have materialized partial matches for the given match of u_4, so we execute the partialJoin in line 16. When u_4 of τ_1 is at line 18, since it has traversed its next sibling after traversing u_5, it does not go to the next sibling, and $\pi_n = \emptyset$.

6 RELATED WORK

In-memory Subgraph Matching. Due to its importance and wide applications, subgraph matching has been studied for decades. Many works follow the classic filtering-ordering-enumerating framework [49]. For pre-computing and filtering the search space, rules have been proposed based on vertex labels, query graph structures and isomorphic constraints [4, 5, 10, 19, 20, 29, 30, 50, 51, 65]. To select a good matching order, various heuristics based on prioritizing dense nodes in the pattern graph and nodes with infrequent labels have been proposed [5, 19, 20, 29, 30, 38, 45, 48, 50, 51]. There are also cost-based approaches that find attribute orders with minimum cost [38]. Recent works improve the backtracking enumeration by pruning invalid search branches [3, 19] or reusing previous results [23, 29, 30, 35]. On top of the classic Ullmann's backtracking or WCOJ, new algorithms have been proposed to reduce the recursion levels using independent sets of the pattern graph [26, 35, 48, 60]. In [43], it studies optimizing multiple subgraph matching queries. It assumes that the larger the shared subgraph is, the more beneficial it will be. The heuristic is not as effective as our cost-based approach. In [18], it studies the computation reuse with a vertex order for a given query, which is different from the reuse of different bags (sub-queries), and it does not optimize attribute orders. Some works also use a decomposition-based approach to process subgraph queries, but their decompositions are not tree decompositions. [25, 47] study preprocessing the query during the visual construction by making use of query fragments. [59] studies a partial topology query to address the difficulty for users to formulate a subgraph matching query, which takes a set of disconnected subgraphs as input and returns graphs connecting them. Some recent surveys and experimental studies can be found in [49, 64].

Parallel and Distributed Subgraph Matching. There are also distributed approaches that focus on increasing parallelism, balancing workloads, and reducing communication costs. Some earlier works propose various sub-structures, joining the sub-structures first and then assembling the matches of sub-structures to obtain the pattern's results [31, 32, 41, 44]. HUGE [61] switch between BFS and DFS to strike a balance between memory efficiency and parallelism, and use a push-pull hybrid communication model to reduce the communication cost. TenGraph [33] uses binary joins and Pytorch tensor operators to process subgraph queries. Subgraph matching has also been studied in new hardware such as GPU [8, 17, 52], FPGA [27] and PIM [7].

Tree decomposition and Query Processing. Tree-decomposition and its variants (e.g. FHD) have been widely used for constraint satisfying problems [15], cyclic conjunctive queries [14] and their extensions [56]. The state-of-the-art approach to compute an optimal FHD is BB4FHD [22]. For applying tree decompositions to query processing, EmptyHeaded [2] follows a classic two-phase approach that computes bags first and then joins the materialized views of bags. However, it materializes all bags and has a high memory cost. DISC [63] and SCOPE [34] use tree decomposition for local subgraph counting. When bags are too large to materialize, DISC does not materialize them, and SCOPE materializes the matches of one attribute or edge to bound the memory consumption. They convert the input counting query to the sum of a set of queries, where each query becomes decomposable, and then process each query based on tree decomposition. CacheTrieJoin [28] uses tree decomposition to accelerate the WCOJ of the full query rather than decomposing the query. To avoid materializing bags, it maintains caches for each bag instead of storing all tuples. However, it is hard to predict whether tuples in the cache will be used later and set caching strategies and sizes. Our new approach handles the repeated computations in different tree nodes and also makes full use of materialization to reduce repeated computations.

7 EXPERIMENTS

Algorithms: We compare with seven representative baselines that have emerged in recent years: RapidMatch [51], VEQ [29], GuP [3], BICE [9], IVE [24], Circinus [26] and BSX [35]. We omit earlier subgraph matching approaches since they are considered as outdated [49, 64]. In ASDMatch, we use the branch-&-bound algorithm in [22] to compute a fractional hypertree decomposition, and the

Table 1: Data graphs

Category	Dataset	Name	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d
Biology	Yeast [49]	*ys*	3,112	12,519	71	8.0
Biology	Human [49]	*hm*	4,674	86,282	44	36.9
Lexical	Wordnet1 [49]	*wn1*	76,853	120,399	5	3.1
Lexical	Wordnet2 [64]	*wn2*	146,005	656,999	15	9.0
Social	DBLP [49]	*db*	317,080	1,049,866	15	6.6
Web	Stanford [64]	*sf*	281,903	1,992,636	30	14.1
Social	Youtube [49]	*yt*	1,134,890	2,987,624	25	5.3
Social	Twitch [64]	*tw*	168,114	6,797,557	60	80.9
Web	eu2005 [49]	*eu*	862,664	16,138,468	40	37.4
Citation	US Patents [49]	*up*	3,774,768	16,518,947	20	8.8

Table 2: Number of unfinished queries

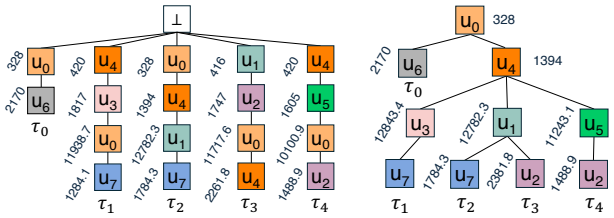
Algorithm	*ys*	*wn1*	*wn2*	*hm*	*db*	*sf*	*yt*	*tw*	*eu*	*up*
RM	363	439	698	866	634	546	450	444	701	380
VEQ	2	473	0	721	78	56	50	147	428	5
GuP	2	535	9	832	287	444	82	199	688	38
BICE	4	492	6	827	311	397	97	248	779	29
IVE	3	335	0	788	156	32	3	18	474	0
Circinus	7	271	1	696	92	17	11	27	331	1
BSX	1	483	0	681	1	6	29	34	220	0
ASDMatch	2	278	0	677	30	0	0	0	213	0

stratified graph sampling proposed in [46] for cardinality estimation. We incorporate the filtering method in CFL [5] due to its simplicity and efficiency. It first obtains a BFS tree of q. Next, it builds candidates for each pattern graph node u level-by-level in this BFS tree and filters candidates v' of pattern nodes u' in the previous level that $(u, u') \in E_p$ but $C(u) \cap N(v') = \emptyset$. Lastly, it refines $C(u)$ in a bottom-up order based on this filtering rule. As indicated in [13, 57], for acyclic joins, WCOJ demonstrates good performance, especially if the join results become larger as more relations and attributes are joined. Therefore, we also use WCOJ for joining the materialized relations.

Dataset graphs: Our evaluation includes ten data graphs from a diverse range of categories, including web graphs, lexical graphs, biological networks, collaboration networks, and social networks. Table 1 shows the statistics. The last four columns are the number of nodes, the number of edges, the number of labels, and the average degree, respectively. All data graphs are taken from [49, 64]. We use two different versions of the Wordnet graph, *wn1* from [49] and *wn2* from [64].

Query graphs: Following previous works [9, 24, 26, 29, 49, 51, 64], we generate pattern graphs by extracting subgraphs from the data graph by a random walk process. For each data graph, we generate five sets of queries corresponding to five pattern graph sizes $n \in \{8, 10, 12, 14, 16\}$, where each set has 200 queries.

Settings: All experiments are conducted on a machine running CentOS 9.4, equipped with an Intel Xeon E7-8891 v3 80-core CPU and 256GB of memory. GuP is implemented in Rust and built using cargo 1.72.1 with the –release flag enabled. All other algorithms are written in C++ and compiled using g++ 11.4.1 with the -O3 optimization flag. The source codes for all baseline methods were obtained directly from the authors. Consistent with previous work [26], we report the time of enumerating all subgraph matches. We terminate a query if it can not be completed in 1 hour. All algorithms were executed in a single thread using one physical core



(a) costs of Fig. 2(a) (b) costs of Fig. 2(c)

Attribute tree	Fig. 2(a)	Fig. 2(b)	Fig. 2(c)	Fig. 2(a)	Fig. 2(a)
Materialization	$O(\infty)$	$O(\infty)$	$O(\infty)$	$O(1)$	$O(m)$
Time (s)	0.100	0.099	0.080	6.892	0.134

(c) Execution times

Figure 8: A case study

exclusively. By default, we set the memory budget of ASDMatch as 16 GB. To reduce the experiment time, we used up to 16 cores to run 16 experiments simultaneously. We also count the number of set intersections, which is an important metric used in previous works [26, 49, 51] that reflects the computation cost.

7.1 A simple case study

As a simple case study to start, we use the pattern graph in Fig 1. The data graph is sf. We have discussed three query plans in Fig. 2. The costs of each attribute of the plan with the optimal independent costs and with the optimal overall cost are shown in Fig. 8(a) and Fig. 8(b), respectively. Fig. 8(c) shows the execution times of five algorithms using different attribute trees and materialization strategies. For these two attribute trees, the total costs are 64,003.6 and 47,699.9, respectively. For τ_3, the total cost of the optimal order in Fig. 8(a) is 16,142.4. For the order in Fig. 8(b), the total cost is 16,886.1, but the costs of three attributes $\{u_0, u_4, u_1\}$ are shared with others. Therefore, it is possible that sub-optimal orders with more sharing can yield better attribute trees. Fig. 8(c) shows the execution time of each attribute tree. The optimal attribute tree is the fastest. Second, for materialization strategies, we compare with the $O(1)$ approach [63] and $O(m)$ approach [34]. We use the tree decomposition in Fig. 1(c) and the attribute tree in Fig. 2(a). For the $O(1)$-approach, no attribute is materialized. For the $O(m)$-approach, the prefixes are $\{u_0, u_3, u_4\}$, and the remaining attributes are materialized. For this query, since the memory is sufficient, our ASDJoin uses the $O(\infty)$ approach, and it is more efficient than the $O(m)$ approach.

7.2 Performance evaluation

The overall performance. We compare our ASDMatch with state-of-the-art labeled subgraph matching algorithms. First, Table 2 shows the number of unfinished queries within the 1 hour time limit across all dataset graphs and algorithms. Our ASDMatch has the least number of unfinished queries, particularly for sf, yt and tw, where ASDMatch is the only approach that finishes all queries. The *hm* graph is challenging due to its high density and skewed label distribution. Here, our ASDMatch has the least number of unfinished queries. An exception is that BSX performs exceptionally well in the *db* graph. Fig 9 shows the running time across all algorithms in all data graphs. Here, we collect all queries where

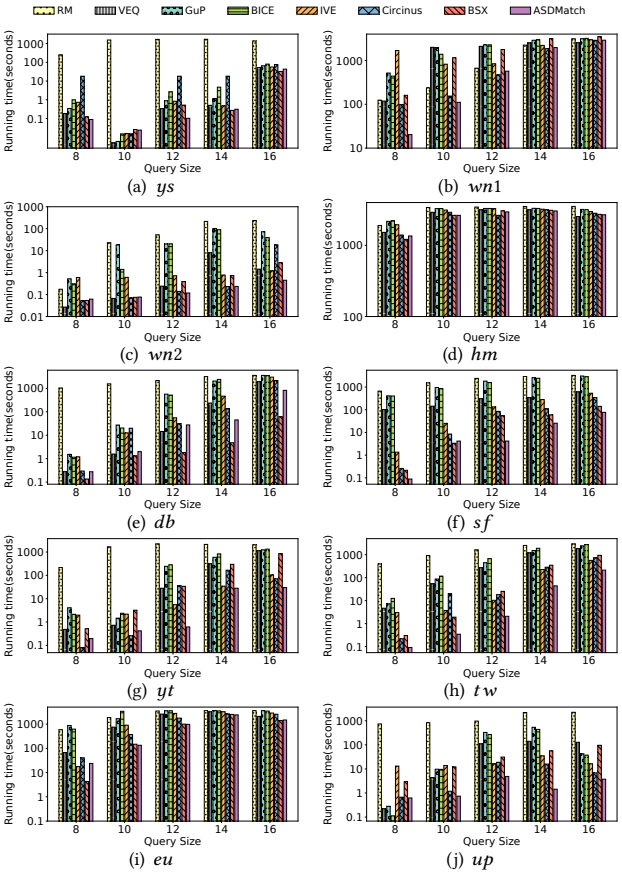


Figure 9: Running times on different data graphs

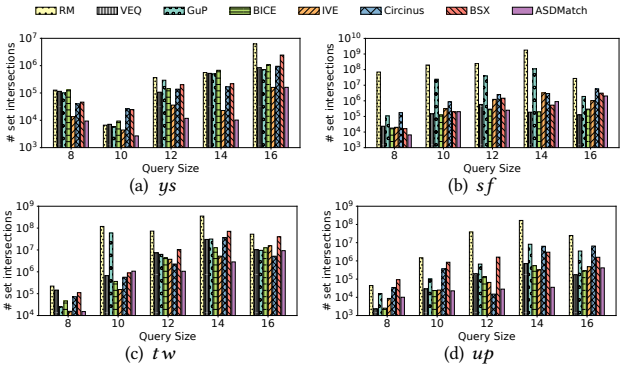


Figure 10: Number of intersections on different data graphs

at least one algorithm can compute within the time limit, set the time of timeout algorithms as 1 hour, and compute the average running time. We can see that ASDMatch is the fastest in most cases, followed by BSX, and then Circinus. The running times increase for larger query sizes, and our ASDMatch remains fast for various query sizes. On average, our ASDMatch is 6.17× faster than BSX, and 14.52× faster than the second-best baseline Circinus.

The number of set intersections. As discussed in previous works, a key performance factor of the backtracking enumeration and

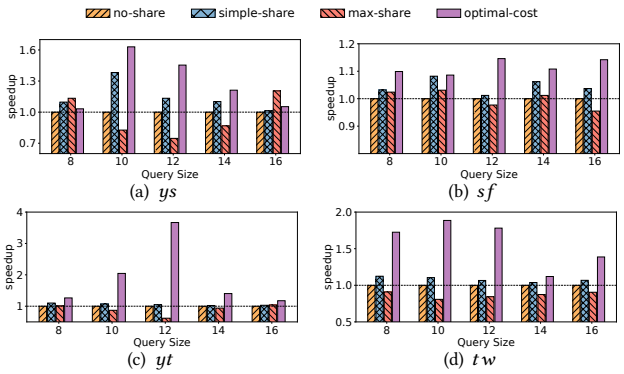


Figure 11: Speedups of computation sharing

WCOJ is the set intersections [26, 49, 51]. We collect the number of set intersections of all algorithms in four data graphs and plot them in Fig. 10. Here, we collect queries that all algorithms can process within the time limit. The trend is similar to that of the running time in Fig. 9. In most cases, our ASDMatch has the least number of set intersections. In our decomposition-based approach, we enumerate $p(\tau_i)$ rather than p. When joining $\mathbf{R}(\tau_1) \bowtie \cdots \bowtie \mathbf{R}(\tau_k)$, we only need to join on attributes that are contained in multiple bags. Therefore, the tree-decomposition-based framework can use fewer set intersections to compute the results. Additionally, we share computations in different bags and reduce the repeated computations by adaptively selecting attributes as the prefix, which further reduces the number of set intersections. Our ASDMatch reduces the number of set intersections of BSX by a factor of 15.90×, and 16.56× compared to Circinus. VEQ, GuP, BICE, and IVE use pruning during the backtracking process. While pruning is effective for reducing the number of intersections, it suffers from overheads such as maintaining the data structures used for pruning. Therefore, they tend to have a small number of intersections while having a long running time.

We analyze the cases where our ASDMatch does not perform the best. First, for the data graph, BSX performs well in the *db* dataset. We find that in the *db* graph, candidates are likely to have identical neighbors in the candidate space, and the techniques of BSX make use of this property. Second, from the perspective of query graphs, ASDMatch may not perform well for queries with large forest-structures [5]. Here, the forest-structure is induced by the edges of p that are not in the 2-core of p, and therefore does not have cycles. For such forest-structure, the fractional hypertree decomposition produces bags with a single edge. Their matches are simply the edge tables in the candidate set data structure. Therefore, the tree decomposition is not effective for processing such forest-structure. The online appendix [1] gives more details.

7.3 Query planning testings

Effectiveness of Attribute Tree Optimization. Our analysis demonstrates that optimizing attribute trees can enhance the algorithm's performance. We evaluate four attribute tree construction methods: ① directly using optimal per-bag attribute orders (no-share); ② simply merging optimal orders of bags (simple-share); ③ selecting trees with minimal attributes (i.e. with maximal sharing) (max-share); ④ identifying attribute trees with minimum overall

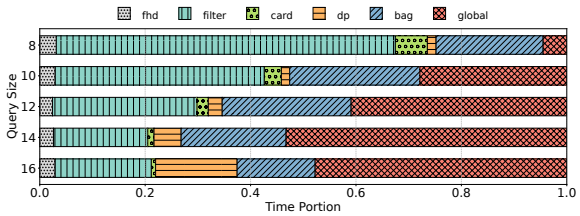


Figure 12: Running time breakdown

Table 3: Planning time(s) of different methods

query size	8	10	12	14	16
opt-order [38]	0.003	0.004	0.006	0.008	0.011
brute-force	0.057	0.342	305	2,109	3,346
simple-transfer	0.070	0.026	0.063	2.00	11.0
DPTree	0.004	0.005	0.009	0.020	0.169

costs via DPTree (optimal-cost). Fig. 11 compares the speedup ratios of ASDJoin using these methods across four data graphs, normalized against no-share. The speedup of noshare to itself (the yellow bar) is always 1.0. simple-share shows modest improvement (1.078× on average) by exploiting common prefixes of the orders produced by no-share. max-share exhibits degraded performance due to its oversharing heuristic. The optimal-cost method achieves superior results, validating our optimization approach. The average speedup is 1.47×. optimal-cost is not always the fastest since the cardinality estimator may under/over-estimate cardinalities, leading to possibly inefficient query plans.

The efficiency of the DPTree algorithm. Table 3 shows the planning time of the brute-force query optimizer and the DPTree algorithm in Section 4. These times include the FHD computation and cardinality estimation time. Here, the brute-force method enumerates all attribute orders of bags, then merges common prefixes for each combination of the orders. The running time is proportional to the product of factorials of the size of shared attributes. For queries with 16 nodes, it often runs out of time. Simple-transfer is a dynamic programming algorithm that uses the simple pairwise state transfer Eq. (10) given in the paper. DPTree further reduces the number of state transfers following Eq. (11-12) in the paper. DPTree is faster than simple-transfer, which verifies the effectiveness of our optimized state transfer. Opt-order is to find optimal orders of bags using the approach in [38] and then merges the orders by the common prefixes to get an attribute tree. Note that Opt-order does not find the optimal attribute tree, where optimizing the attribute tree is a much harder problem. By comparing DPTree and opt-order, we can see that DPTree does not incur much additional computation cost in finding the optimal attribute tree.

Processing time breakdown. Fig. 12 shows the breakdown of our ASDMatch algorithm. The results are averaged over all finished queries for each query size. Here, fhd is the time for computing a fractional hypertree decomposition using our algorithm [22], card is the time for cardinality estimation using [46], and dp is the time for DPTree (Algorithm 2). filter is the filtering time of CFL [5]. bag is the time for computing bags, and global is the time for the acyclic join and outputting results. The sum of FHD, card, and bag is the query planning time, and the sum of bag and global is the execution time. First, for query planning, the time for FHD computation and cardinality estimation is negligible, and the overall query planning time is less than 20%. In most cases, it is shorter than the filtering time. Second, the execution time dominates the overall time, especially for large query graphs. The bag processing time takes a non-trivial portion. Therefore, it is important to optimize the attribute tree.

Table 4: Comparing materialization strategies

		static		adaptive					
memory budget		$O(1)$	$O(m)$	512K	4M	32M	256M	2G	16G
db	time(s)	554.7	45.94	18.56	16.84	16.08	16.06	15.92	15.92
	#prefix	12	5.72	3.225	1.205	0.145	0.005	0	0
	usage(KB)	0	107.5	110.3	167.6	197.1	203.8	1,101	1,101
yt	time(s)	413.5	0.869	18.439	2.654	7.431	0.517	0.521	0.504
	#prefix	12	5.17	3.36	1.51	0.615	0.13	0.01	0
	usage(KB)	0	9.027	13.64	14.09	1,630	1,832	24,199	41,087
tw	time(s)	1,031	11.98	11.51	3.644	2.492	1.297	1.245	1.245
	#prefix	12	4.36	2.76	1.115	0.245	0.02	0	0
	usage(KB)	0	20.90	21.02	21.62	21.92	22.25	1,645	1,645

7.4 The effect of increasing materialization

We test the effect of materialization strategies (i.e., static or adaptive) and the impact of memory size on algorithm performance using three data graphs and queries with 12 nodes. Table 4 shows the results. For the adaptive approach, we vary the memory budget from 512KB to 16GB. We show the absolute memory usage of all compared methods. We focus on the memory for materializing tuples in bags, excluding the memory for the data graph, etc. We normalize the memory usage by subtracting the memory usage by the memory usage of the $O(1)$-approach. The actual memory usage is smaller than the memory budget since many queries do not need that much memory. The normalized actual memory usage reflects the memory utilization, and our adaptive materialization has better utilization than the static $O(m)$ approach. For the adaptive approach (i.e. ASDJoin), we record the maximum size of the prefix during the execution and report the average of the maximum numbers. In general, the adaptive approach outperforms the static approach, and when the memory budget increases, we use smaller prefix and the performance is better. There are exceptions where increasing the budget leads to slower performance, such as increasing memory from 4M to 32M results in slower performance in *yt*. The reason is that when there are bags that induce disconnected subgraphs, a larger prefix could be better since we implement some pruning by making use of the prefix. We leave the details in [1]. If we further increase the memory budget, prefix becomes smaller and the performance becomes better.

8 CONCLUSION

We propose a novel decomposition-based approach ASDMatch for subgraph matching. We study the new problem of optimizing join attribute orders with computation costs sharing and propose a dynamic programming algorithm to solve it. Besides, we propose a new adaptive multi-join algorithm to address the materialization problem in previous decomposition-based approaches. We confirm the superiority of our tree-decomposition-based matching by comparing our ASDMatch with seven state-of-the-art approaches in 8 large data graphs, using pattern graphs of various sizes. In three large data graphs, ASDMatch is the only algorithm that can compute all pattern graphs within the given time limit.

REFERENCES

[1] The online repository and appendix of the paper. https://github.com/magic62442/ASDMatch.

[2] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446. ACM, 2016.

[3] J. Arai, Y. Fujiwara, and M. Onizuka. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data*, 1(2):1–26, 2023.

[4] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, pages 1447–1462. ACM, 2019.

[5] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214. ACM, 2016.

[6] V. Bonnici, R. Giugno, A. Pulvirenti, D. E. Shasha, and A. Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.*, 14(S-7):S13, 2013.

[7] S. Cai, B. Tian, H. Zhang, and M. Gao. Pimpam: Efficient graph pattern matching on real processing-in-memory hardware. *Proc. ACM Manag. Data*, 2(3):161, 2024.

[8] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.*, 13(8):1190–1205, 2020.

[9] Y. Choi, K. Park, and H. Kim. BICE: exploring compact search space by using bipartite matching and cell-wide verification. *Proc. VLDB Endow.*, 16(9):2186–2198, 2023.

[10] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[11] W. Fischl, G. Gottlob, D. M. Longo, and R. Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings, 2020.

[12] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[13] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[14] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree Decompositions: Questions and Answers. In *Proc. of PODS'16*, pages 57–74, 2016.

[15] G. Gottlob, C. Okulmus, and R. Pichler. Fast and parallel decomposition of constraint satisfaction problems. In C. Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1155–1162. ijcai.org, 2020.

[16] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1), aug 2014.

[17] W. Guo, Y. Li, and K. Tan. Exploiting reuse for GPU subgraph enumeration (extended abstract). In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 3765–3766. IEEE, 2023.

[18] W. Guo, Y. Li, and K.-L. Tan. Exploiting reuse for gpu subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering*, 34(9):4231–4244, 2020.

[19] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*, pages 1429–1446, 2019.

[20] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*, pages 337–348, 2013.

[21] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418. ACM, 2008.

[22] Z. He and J. X. Yu. A branch-&-bound algorithm for fractional hypertree decomposition. *Proc. VLDB Endow.*, 17(13):4655–4667, 2024.

[23] X. Jian, Z. Li, and L. Chen. Suff: Accelerating subgraph matching with historical data. *Proc. VLDB Endow.*, 16(7):1699–1711, 2023.

[24] Z. Jiang, S. Zhang, X. Hou, M. Yuan, and H. You. IVE: accelerating enumeration-based subgraph matching via exploring isolated vertices. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 4208–4221. IEEE, 2024.

[25] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. Prague: towards blending practical visual subgraph query formulation and query processing. In *2012 IEEE 28th International Conference on Data Engineering*, pages 222–233. IEEE, 2012.

[26] T. Jin, B. Li, Y. Li, Q. Zhou, Q. Ma, Y. Zhao, H. Chen, and J. Cheng. Circinus: Fast redundancy-reduced subgraph matching. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[27] X. Jin, Z. Yang, X. Lin, S. Yang, L. Qin, and Y. Peng. FAST: fpga-based subgraph matching on massive graphs. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1452–1463. IEEE, 2021.

[28] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 282–293. OpenProceedings.org, 2017.

[29] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *SIGMOD*, pages 925–937. ACM, 2021.

[30] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB journal*, 32(2):343–368, 2023.

[31] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce: a cost-oriented approach. *VLDB J.*, 26(3):421–446, 2017.

[32] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.*, 10(3):217–228, 2016.

[33] G. Li, H. Zhang, X. Sun, Q. Luo, and Y. Zhu. Tengraph: A tensor-based graph query engine. *Proc. VLDB Endow.*, 17(13):4571–4584, 2024.

[34] Q. Li and J. X. Yu. Fast local subgraph counting. *Proc. VLDB Endow.*, 17(8):1967–1980, 2024.

[35] Y. Lu, Z. Zhang, and W. Zheng. BⓈX: Subgraph matching with batch backtracking search. *Proc. ACM Manag. Data*, 3(1), Feb. 2025.

[36] D. Marx and M. Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 542–553, 2014.

[37] A. Mhedhbi, A. Deshpande, and S. Salihoglu. Modern techniques for querying graph-structured databases. *Found. Trends Databases*, 14(2):72–185, 2024.

[38] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[39] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case Optimal Join Algorithms: [Extended Abstract]. In *Proc. of PODS'12*, pages 37–48, 2012.

[40] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *Proc. VLDB Endow.*, 11(2):176–188, 2017.

[41] M. Qiao, H. Zhang, and H. Cheng. Subgraph Matching: On Compression and Computation. *Proc. VLDB Endow.*, 11(2):176–188, 2017.

[42] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

[43] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment*, 10(3):121–132, 2016.

[44] X. Ren, J. Wang, W. Han, and J. X. Yu. Fast and robust distributed subgraph enumeration. *Proc. VLDB Endow.*, 12(11):1344–1356, 2019.

[45] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.

[46] W. Shin, S. Song, K. Park, and W. Han. Cardinality estimation of subgraph matching: A filtering-sampling approach. *Proc. VLDB Endow.*, 17(7):1697–1709, 2024.

[47] Y. Song, H. E. Chua, S. S. Bhowmick, B. Choi, and S. Zhou. Boomer: Blending visual formulation and processing of p-homomorphic queries on large networks. In *Proceedings of the 2018 International Conference on Management of Data*, pages 927–942, 2018.

[48] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *ICDE*, pages 232–243. IEEE, 2019.

[49] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098. ACM, 2020.

[50] S. Sun and Q. Luo. Subgraph matching with effective matching order and indexing. *IEEE Trans. Knowl. Data Eng.*, 34(1):491–505, 2022.

[51] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He. Rapidmatch: A holistic approach to subgraph query processing. *Proc. VLDB Endow.*, 14(2):176–188, 2020.

[52] X. Sun and Q. Luo. Efficient gpu-accelerated subgraph matching. *Proc. ACM Manag. Data*, 1(2):181:1–181:26, 2023.

[53] Y. Sun, G. Li, J. Du, B. Ning, and H. Chen. A subgraph matching algorithm based on subgraph index for knowledge graph. *Frontiers Comput. Sci.*, 16(3):163606, 2022.

[54] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, 2012.

[55] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[56] Q. Wang and K. Yi. Conjunctive queries with comparisons. *SIGMOD Rec.*, 52(1):54–62, 2023.

[57] Y. R. Wang, M. Willsey, and D. Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2):150:1–150:23, 2023.

[58] S. Wernicke and F. Rasche. FANMOD: a tool for fast network motif detection. *Bioinform.*, 22(9):1152–1153, 2006.

[59] M. Xie, S. S. Bhowmick, G. Cong, and Q. Wang. Panda: toward partial topology-based search on large networks in a single machine. *The VLDB Journal*, 26:203–228, 2017.

[60] R. Yang, Z. Zhang, W. Zheng, and J. X. Yu. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proc. ACM Manag. Data*, 1(1):15:1–15:26, 2023.

[61] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang. HUGE: an efficient and scalable subgraph enumeration system. In *SIGMOD*, pages 2049–2062. ACM, 2021.

[62] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Very Large Data*

Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings, pages 82–94. IEEE Computer Society, 1981.

[63] H. Zhang, J. X. Yu, Y. Zhang, K. Zhao, and H. Cheng. Distributed subgraph counting: A general approach. *Proc. VLDB Endow.*, 13(11):2493–2507, 2020.

[64] Z. Zhang, Y. Lu, W. Zheng, and X. Lin. A comprehensive survey and experimental study of subgraph matching: Trends, unbiasedness, and interaction. *Proc. ACM Manag. Data*, 2(1):60:1–60:29, 2024.

[65] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.*, 3(1-2):340–351, 2010.

Appendix

A PROOF AND COMPLEXITY ANALYSIS

Proof of Lemma 4.1. We prove this lemma by contradiction. Suppose the optimal attribute tree T_A^* on τ is not shared on the longest common prefix. Let $\pi = \Lambda_{\tau_i \in \tau} \pi^{(i)}$ be the longest common prefix on τ, then T_A^* shared on π' must be $\pi' \prec \pi$. Let $|\pi'| = t$ and $|\pi| = l > t$, then the unshared part of π has $l - t$ attributes, i.e, $unshared = \{\pi_{t+1}, ..., \pi_l\}$. We can construct a new attribute tree T_A' by merging the unshared part of π. Due to the fact that each τ_i has a fixed attribute order, then (1) for each unshared attribute π_i, it will appear at least twice in T_A^* and only once in T_A'. (2) merging the unshared part of π does not affect the cost of the remaining node on T_A^*. Let the set of nodes $V_{unshared}$ be the nodes for unshared attributes in T_A^* and $c(\pi_i)$ be the number of times π_i appears, we have:

$$\begin{aligned}
\text{cost}(T_A^*) &= \sum_{\substack{u \in T_A^* \\ u \notin V_{unshared}}} \text{cost}(u) + \sum_{u \in V_{unshared}} \text{cost}(u) \\
&= \sum_{\substack{u \in T_A^* \\ u \notin V_{unshared}}} \text{cost}(u) + \sum_{\pi_i \in unshared} c(\pi_i) \cdot \text{cost}(\pi_i) \\
&> \sum_{\substack{u \in T_A^* \\ u \notin V_{unshared}}} \text{cost}(u) + \sum_{\pi_i \in unshared} \text{cost}(\pi_i) \\
&= \text{cost}(T_A')
\end{aligned}$$

Thus, T_A' is a better attribute tree than T_A^*, which contradicts the assumption.

The Complexity of DPTree: For a given tree decomposition T on pattern graph p, there are k bags, $\tau = \{\tau_1, \tau_2, \cdots, \tau_k\}$ and the bag size is bounded by $tw(T) + 1$. For a state in DPTree, (S, π), has $S \subseteq \tau$ and $\pi \in \text{Perm}(\cap_{\tau_i \in S} \tau_i)$. Here, the number of subsets of τ is 2^k and the number of possible longest common prefixes for each subset S can be roughly bounded by $2 \cdot (\text{tw}(T) + 1)!$. Thus, the total number of states is:

$$\sum_{S \subseteq \tau} |\bigcup_{A \subseteq \bigcap_{\tau_i \in S} \tau_i} \text{Perm}(A)| \leq \sum_{S \subseteq \tau} 2 \cdot (\text{tw}(T) + 1)! = 2^{k+1} \cdot (\text{tw}(T) + 1)!$$

For each state (S, π), there are $2^{|S|} - 1$ enumeration of division of S into two subsets, π can be directly transferred from the optimized transition using the state transition optimization in Eq. (12). The total number of transitions is:

$$\sum_{j=2}^{k} \sum_{S \subseteq \tau, |S|=j} (2^j - 1) \cdot |\bigcup_{A \subseteq \bigcap_{\tau_i \in S} \tau_i} \text{Perm}(A)|$$

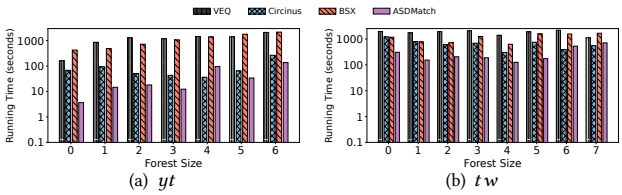
$$\leq \sum_{j=2}^{k} \sum_{S \subseteq \tau, |S|=j} (2^j - 1) \cdot 2 \cdot (\text{tw}(T) + 1)! \leq 2 \cdot 3^k \cdot (\text{tw}(T) + 1)!$$

Each transition takes $O(k)$ time to record up to k orders of $|\mathcal{S}|$ bags (lines 12-16). Each state updates **cost** and $\mathbf{T}_A$ which takes $|\pi|$ time (lines 17-20). $|\pi|$ is bounded by $\text{tw}(T) + 1$. Therefore, the time complexity of DPTree is $O(k \cdot 3^k \cdot (tw(T) + 1)! + 2^k \cdot (\text{tw}(T) + 1) \cdot (tw(T) + 1)!)$, and the space complexity is $O(k \cdot 3^k \cdot (tw(T) + 1)!$.

In practice, the sharing size is not as large as $\text{tw}(T)$, and becomes smaller when there are more bags. The number of bags is also not large. Therefore, DPTree is highly efficient. We find that for a pattern graph $p = (V_p, E_p)$ with $|V_p| \leq 16$, the time to find the optimal attribute tree is much shorter than the time to find subgraph matches. Therefore, it is worth finding the optimal attribute tree.

B ADDITIONAL EXPERIMENTS

Brief explanations of the baselines. All the methods compared follow the filtering-ordering-enumerating framework. where the bottleneck is in enumeration. For the enumeration, RapidMatch [51] adopts the classic worst-case optimal join and proposes a new data layout and accelerates set intersection computations. VEQ [29], GuP [3] and BICE [9] develop pruning techniques to avoid unnecessary search branches. VEQ maintains equivalence sets of data nodes where these data nodes have identical neighbors in the candidate space index. Once matches for a node within the equivalence set are enumerated, the corresponding matches for the other nodes in the set can be directly inferred. BICE maintains a bipartite graph during the backtracking process and prunes a search branch if the size of the maximum matching in the bipartite graph is smaller than the number of unmapped query vertices. GuP encodes failures in the backtracking process in a structure called "guard" and prunes redundant search branches when a similar failure happens again. IVE [24] and Circinus [26] make use of the vertex cover and the independent set of the pattern graph nodes. When a vertex cover of p is matched, the remaining nodes are independent and can be processed efficiently by Cartesian product and injectivity checking. BSX [35] matches a pattern graph node to a batch of data graph nodes at a time rather than just one data graph node to reduce the search space.



(a) yt (b) tw

Figure 13: Performance under different forest-structure sizes

Varying forest-structure sizes. Fig. 13 shows the performance of four algorithms under different forest sizes. The pattern graph size is fixed to $|V_p| = 16$. Here, the forest-structure is induced by the edges of p that are not in the 2-core of p, and therefore does not have cycles. For such forest-structure, the fractional hypertree decomposition produces bags with a single edge. Their matches are the edge tables in the candidate set data structure. Therefore, the tree decomposition is not effective for processing such forest-structure. BSX and our ASDMatch do not have specific optimizations for the forest-structure. VEQ and Circinus are suitable for processing forest-structures. VEQ can efficiently prune invalid search branches and produce similar matching results for data graph nodes with the same neighbors in the candidate space. Circinus allows for processing disconnected vertex cover in the query graph, which is effective

for processing the forest-structure in a query. Fig. 13 shows the performance of four algorithms under different forest sizes. The pattern graph size is fixed to $|V_p| = 16$. We can see that when the size of the forest-structure increases, the gap between BSX and ASDMatch is relatively stable. In contrast, the gap between VEQ (Circinus) and ASDMatch decreases. For example, in the tw graph, when there is no forest-structure, ASDMatch outperforms Circinus by 3.89×. When the forest-structure size is 7, Circinus outperforms ASDMatch by 1.29×.

More discussions about Fig. 9. BSX performs well in the *db* dataset. Our ASDMatch performs the second best. BSX matches a pattern graph node to a batch of data graph nodes at a time, rather than just one data graph node. A batch of data graph nodes should have identical neighbors in a subspace of the candidate sets. In the *db* graph, data graph nodes with the same label tend to have similar neighbors and is likely to form identical batches when searching inside a subspace of candidates during the backtracking process. The average batch size is 1.69. Considering the product of batch sizes of query nodes, the product is 43.5. Both the size and the product are the largest among all data graphs. Therefore, BSX becomes highly efficient in *db*. However, in other data graphs, the batches are smaller and the performance of BSX is not as good as *db*.

Table 5: The number of subgraph matches of test cases

query size	8	10	12	14	16
yeast	5.1×10^3	5.6×10^2	1.3×10^4	5.3×10^4	8.8×10^4
dblp	1.1×10^7	1.7×10^7	5.3×10^8	1.9×10^{10}	2.0×10^{13}
youtube	1.3×10^5	5.6×10^3	2.1×10^6	6.9×10^7	9.4×10^8
patents	6.8×10^3	6.7×10^4	3.6×10^6	4.0×10^7	1.7×10^5
human	1.4×10^9	1.3×10^{11}	1.9×10^{10}	1.1×10^{10}	3.9×10^{10}
eu2005	2.2×10^9	3.5×10^{10}	9.8×10^{12}	1.5×10^{16}	1.1×10^{18}
stanford	4.2×10^7	2.1×10^9	3.2×10^{10}	1.3×10^{12}	6.9×10^{13}
twitch	4.5×10^5	8.3×10^6	2.2×10^8	5.5×10^9	3.5×10^{10}
WordNet1	2.8×10^9	1.0×10^8	2.3×10^8	4.3×10^9	2.0×10^7
WordNet2	1.4×10^4	2.8×10^4	2.1×10^5	3.3×10^5	6.6×10^5

Number of results. We report the number of results of each test case in Table 5. Since the number of results can vary by tens of magnitudes, to avoid the number being dominated by the largest one, we use the geometric mean of the results.

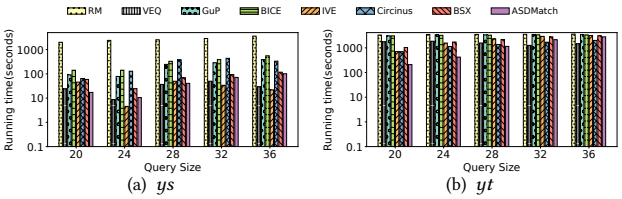


Figure 14: Performance of large query graphs

Performance for large queries. We show the results for large queries in Fig. 14. The query graph sizes tested range from 20 to 36. For each case, we generated 200 queries to test and report the average. The query graphs are sampled from the data graphs by random walks. When the query size is large, it is more likely to produce query graphs with a large forest-structure [5]. Here, the forest-structure is induced by the edges of p that are not in the 2-core of p, and therefore does not have cycles. For query planning, we merge the optimal orders of bags into an attribute tree with sharing, because the cost of finding the optimal attribute tree for a query of >20 nodes is high.

Table 6: Comparing with Emptyheaded (seconds)

	pattern graph	p_1	p_2	p_3	p_4	p_5	p_6
Emptyheaded	preprocessing	36.36	15.81	386.26	323.76	24.73	54.86
	evalutation	1.77	7.26	1.95	7.30	10.37	13.36
	total	38.10	23.07	382.21	331.06	35.10	73.22
ASDMatch	preprocessing	0.078	0.069	0.070	0.073	0.081	0.083
	evalutation	0.043	0.007	0.020	0.109	0.026	0.049
	total	0.121	0.076	0.090	0.182	0.107	0.132

Among the eight approaches tested, as shown in Fig. 14, our approach performs best when the query sizes are 20 and 28, and performs the 3rd best in other cases in *ys*; and our approach performs best when the query sizes are 20, 24, and 28, and performs the 3rd best in other cases in *yt*. VEQ/IVE performs the best in the cases where our approach does not perform the best.

In Fig. 9 in the paper, our approach outperforms VEQ by 40.2×, and outperforms IVE by 8.37× on 50 test cases on average when the query size is ≤16. But our approach does not perform well for queries with large forest-structures (e.g., cycle-free) [5], when the query size ≥20, because the fractional hypertree decomposition is used to deal with queries with complex cycles. This confirms the finding in the survey [49] such that the failing set pruning used in VEQ/IVE can significantly improve the performance on large queries.

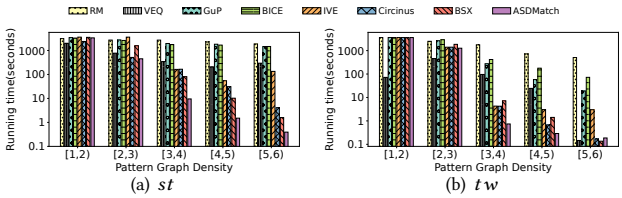


Figure 15: Performance under different query graph densities

Varing query graph densities. Fig. 15 shows the results for query graphs of 12 nodes with varying densities. The density for a query graph $p = (V_p, E_p)$ is calculated as $\frac{2\times|E_p|}{|V_p|}$ following [35]. For each density range and dataset, we generate 50 queries. As the density of the query graphs increases, the number of results decreases, which results in a corresponding reduction in running time. Our ASDMatch performs better for query graphs with densities ≥3, consistently outperforms other algorithms. For the case of density ≥ 5 in the tw graph, most algorithms perform well because the number of results is small. Our ASDMatch does not have the advantage in processing the tree-like query graphs. When the density is <2, the queries are all trees. VEQ can efficiently prune invalid search branches and produce similar matching results for the data graph nodes with the same neighbors in the candidate space. Circinus allows processing a disconnected vertex cover in the query graph, which is effective for processing the forest-structure in a query.

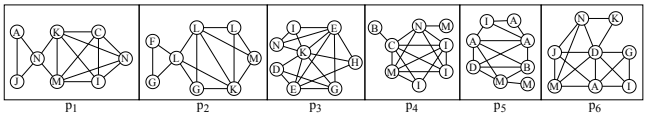


Figure 16: Queries compared with Emptyheaded

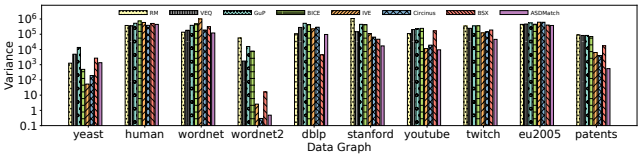


Figure 17: Variance of the algorithms in different data graphs

Comparing with Emptyheaded. We conducted experimental studies to compare EmptyHeaded. The implementation of Emptyheaded has several issues. First, it has a long prepossessing time for computing the FHD and encoding the data structures. Second, it occurs a high memory usage since it needs to materialize the bag relations. As a result, Emptyheaded can not process most of the queries in the experiments due to running out of 1h time or out of 256GB memory. In the *db* graph, when the query size is 8, it can compute 155 queries. For these queries, the average total times of Emptyheaded and ASDMatch are 115.9 and 0.131, respectively. We select 6 query graphs $\{p_1, p_2, p_3, p_4, p_5, p_6\}$ as shown in Fig. 16. The results are presented in Table 6. ASDMatch outperforms Emptyheaded by more than two orders of magnitude in terms of the total time and by more than one order in terms of the evaluation time. On top of the implementations, our ASDMatch improves Emptyheaded by better attribute orders, computation sharing and adaptive materialization.

Analysis for hard queries. The hard queries are the queries with complex cycles that are dense with a large vertex cover. Recent approaches [24, 26, 35, 64] find a vertex cover of the query graph so that pattern graph nodes that are not in the vertex cover form an independent set and can be processed efficiently. However, when the vertex cover size is still large, the existing approaches cannot process them efficiently. We analyze the queries with $|V_p| = 16$ that the state-of-the-art algorithm BSX can not process within the time limit while our ASDMatch can process efficiently. We find that such queries have a larger vertex cover. For example, in the *eu* (*sf*) graph, the average vertex cover size is 9.37 (8.5), while the average vertex cover size of the queries that can be computed within the time limit is 8.39 (7.58). Our ASDMatch is a tree-decomposition-based approach which is not to deal with vertex covers per se. For a given query (e.g., a pattern graph p), we make p as small sub-queries $\{p_i\}$ by tree-decomposition in order to reduce the backtracking costs, and compute p as a cycle-free tree by the results of $\{p_i\}$ with new optimization techniques. The tree-decomposition-based approach has better theoretical complexity and can reduce the number of set intersections in practice.

Variances of running times. Fig. 17 shows the variance of the algorithms in different data graphs. ASDMatch has the smallest variance in the 5 largest graphs *sf*, *yt*, *tw*, *eu* and *pt*, and the variance is not too large in other data graphs. Our ASDMatch performs more stably than the baselines.

The issue of disconnected bags. In Table 4, increasing memory from 4M to 32M results in slower performance in *yt*. We explain as follows. As we discussed in Section 5 and Figure 5 in the paper, the enumeration costs of a bag depend on the union of the prefix of previous bags and the attributes in the current bag. We implemented an optimization technique to use the prefix of previous bags to prune the search space of the current bag. Some details are given below. Suppose that u is in the prefix but not in the current bag, u' is in the current bag but not in the prefix, and $(u, u') \in E_p$, if a node v is not connected with $f(u')$ in the data graph, we do not match v to u'. In most cases, even with this pruning, reducing the prefix is still better. But, there are exceptions for bags that are disconnected subgraphs. Disconnected subgraphs incur many matches. In *yt*, there is a query with a bag τ with a disconnected subgraph. When the budget is 32M, its enumeration cost is $\#n(\text{prefix}_1 \cup \tau)$, and the running time is 703 seconds, as there is no edges between prefix_1 and τ. When the budget is 4M, its enumeration is $\#n(\text{prefix}_2 \cup \tau)$, where $\text{prefix}_1 \subset \text{prefix}_2$ but there are edges between prefix_2 and τ. With the pruning effect, the running time is 2.5 seconds only.