Mix & Match: Subgraph Matching for Absolute Coverage

Konstantinos Skitsas
Aarhus University
skitsas@cs.au.dk

Yuya Sasaki
University of Osaka
sasaki@ist.osaka-u.ac.jp

Davide Mottin
Aarhus University
davide@cs.au.dk

Panagiotis Karras
U. of Copenhagen & Aarhus U.
piekarras@gmail.com

ABSTRACT

The **NP**-hard problem of *subgraph matching* calls to detect all matchings of a smaller *query graph* within a larger *data graph*. The problem is fundamental in graph analysis and query answering, as it facilitates the understanding and analysis of the larger graph. Nevertheless, existing subgraph matching methods return results from one location of the graph before moving to another location, while the total results may be in the order of billions or even trillions; under these circumstances, existing methods may only present a portion of the results within reasonable time or space, which is not representative of the totality of results. This predicament leads to a biased representation of the data graph. In this paper, we study the problem of *coverage* in subgraph matching and propose Mix & Match (M&M) an algorithm that quickly returns results that are representative of the whole data graph. M&M achieves higher coverage employing a combination of *global exploration*, which prioritizes the exploration of nodes at the first level of backtracking that may enlarge coverage, and *local exploration*, which improves backtracking efficiency by pruning exploration paths that do not increase coverage. Our experimental study shows that M&M finds on average twice as many unique nodes as state-of-the-art algorithms in the same time.

PVLDB Reference Format:
Konstantinos Skitsas, Yuya Sasaki, Davide Mottin, and Panagiotis Karras. Mix & Match: Subgraph Matching for Absolute Coverage. PVLDB, 18(13): 5610 - 5622, 2025.
doi:10.14778/3773731.3773737

PVLDB Artifact Availability:
Source code, data, and other artifacts have been made available at https://github.com/constantinosskitsas/Mix-Match.

1 INTRODUCTION

Subgraph matching is essential to graph analysis in chemistry [28, 29], bioinformatics [5, 46], social network analysis [23, 34], computer vision [8, 35], pattern recognition [11, 40], and elsewhere [6, 7, 25, 31, 44]. The problem is **NP**-hard with worst-case time complexity exponential to the query graph size. State-of-the-art subgraph matching algorithms terminate upon reaching a reasonable number of matchings or a predefined runtime threshold, and utilize *backtracking* to return hundreds of thousands of matchings per second [37, 47] by leveraging locality through depth-first exploration, finding all matchings that contain one candidate node before moving on to another, which may be from the same neighborhood, leading to overlapping results [27] that lack wide coverage of the underlying graph. The problem of *top-k diversified subgraph matching* [43] seeks k matchings that achieve high diversity. However, the cardinality constraint k poses a hindrance, as (1) it is hard to set an appropriate value of k in advance, (2) choosing a most diverse set of k known matchings remains **NP**-hard, and (3) refining the top-k set by comparing new to previously found matchings hinders exploring new parts of the graph.

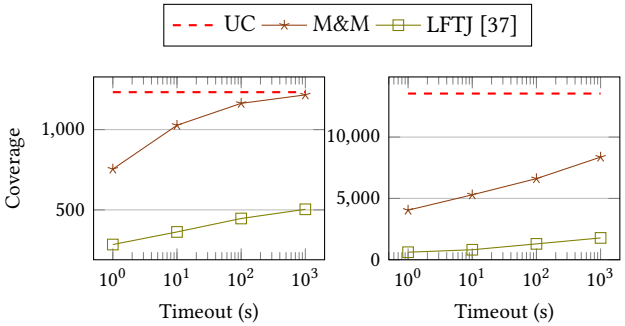


Figure 1: Avg coverage, query size 16**;** DBLP **(L) &** EU **data (R).**

In this paper, we propose Mix & Match (M&M), a subgraph matching algorithm that aims to maximize the amount of distinct nodes in the answer set it returns within a time limit. We extend M&M to return a set of top-k matches that achieve high coverage, while ensuring that k does not affect the subgraph matching runtime. Given enough time, M&M returns all candidate nodes that belong to a matching. Figure 1 presents the number of distinct nodes in the matchings M&M returns vs. LFTJ [37], for queries of size 16 on the DBLP and EU dataset within 1000 seconds. For the sake of completeness, the figure also shows a line standing for the average of UC, which is the number of unique nodes across all matchings, if the algorithm terminates within the time limit, otherwise the number of unique candidate nodes passed to the verification stage. Notably, M&M returns matchings with more distinct nodes within one second than LFTJ does within 1,000 seconds. This outcome indicates that a more diverse representation does not arise by simply letting an algorithm run longer.

A straightforward way to enhance coverage could be to raise the matching limit. Unfortunately, this solution does not unequivocally improve coverage. Figure 2 reports the coverage, as unique nodes

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773737

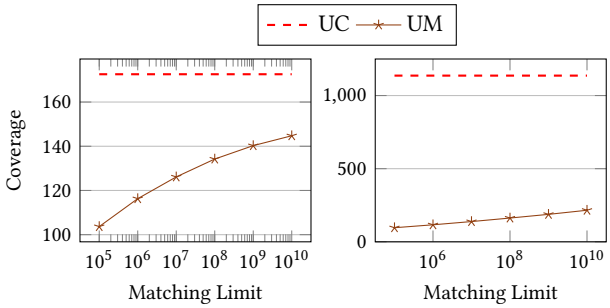


Figure 2: Average coverage of matchings for queries with limit in $[10^5, 10^{10}]$ for Yeast (left) and DBLP (right) data.

participating in returned matchings (UM), while varying the limit on returned matchings from 10^5 to 10^{10} across 400 queries, as well as the average number of unique candidates (UC) passed to the verification phase (including *false positives*). The left side of the figure shows results for a small dataset, on which only 50 queries yield more than 10^{10} matchings (the rest yielding fewer matchings), while the right side shows results for a larger dataset, on which only 5 queries yield fewer than 10^{10} matchings (the rest yielding more). As expected, coverage grows as the limit on returned matchings increases to 10^{10}. On a small dataset with small answer sets, LFTJ captures most matchings by increasing the limit. Still, on a larger dataset, such as DBLP, which yields trillions of matchings (LFTJ needs approximately 12 hours per query to return 10^{10} matchings), increasing the limit does not significantly enhance coverage.

Applications. Subgraph matching maximizing coverage is useful in recommender systems, where a user may require a set of products results satisfying compatibility constraints and wishes to see an overview of products related to the request [9]. Similarly, a manager navigating a collaboration network may be interested in building a team of people combining different skills with a purview of all possible participants [10, 43]. A related application pertains to increasing the job satisfaction of drivers in a public transport system by maximizing the number of locations along driven routes [12]. Lastly, as subgraph matching underpins the forensic analysis of social interactions to detect criminal activities, a coverage objective brings more suspects in scope [23].

We summarize our contributions as follows.

1. We study the problem of coverage in subgraph matching in depth to decrease bias in query answering.
2. We propose M&M, a fast subgraph matching algorithm tailored to coverage considerations.
3. We propose a locality improvement that eliminates redundant node evaluations by capturing symmetries based on shared neighbors among candidate nodes.
4. We additionally augment M&M to return a selection of top-k matchings that cover the most distinct nodes.

2 RELATED WORK

Subgraph matching algorithms usually comprise *filtering*, which removes candidates that cannot match a query vertex, and *enumeration*, which explores remaining candidates by isomorphism checks. There are three broad classes of such algorithms.

Filter and Verification methods [15] leverage pre-indexed structures to remove false positives at query time [32, 41, 45].

Direct Enumeration methods [20] traverse the data graph to identify subgraphs isomorphic to the query by backtracking, using sophisticated ordering and basic filtering to improve efficiency.

Preprocessing enumeration methods [37] construct an auxiliary structure containing edges between candidate nodes at query time, enabling filtering and verification without scanning the entire data graph; these are state-of-art. GQL [16] filters candidates based on neighborhood signatures of nodes and labels and a pseudo-subgraph isomorphism check that verifies the neighborhood of a query node is contained in that of a candidate; PILOS [33] enhances filtering using spectral rules and extends neighborhood-based pruning to two-hop neighborhoods in the candidate space. In terms of ordering, CFL [3] eschews materializing Cartesian products among sets of candidate nodes by a core-forest-leaf decomposition strategy; DAF [13] employs a dynamic order of traversal, considering the expected size of the candidate space and matching paths at each backtracking step, and prunes unpromising paths using the knowledge of previously explored ones by the failing sets method [13]. In the same vein, GuP [1] maintains *guards* for each vertex to anticipate conflicting matches on partial embeddings; VEQ [19] defines dynamic *equivalence classes* as candidate nodes of a query node that share the same neighborhood and extends the concept to *equivalent trees* to prune nodes that lead to symmetric failures or matchings.

The problem of *diversified top-k subgraph matching* [43] retrieves k matchings of a query targeting the maximization of *relative coverage*, i.e., the number of unique nodes across the returned k matchings. A multi-level disjoint algorithm, DSQL [43], iteratively relaxes the overlap constraint until it finds k matches, enabling early termination. However, it performs efficiently only for small queries (up to 10 nodes) and small values of k values (up to 50). By contrast, we target the maximization of *absolute coverage within a time limit*, i.e., the number of unique nodes across *all* matchings, as well as scalability to large queries. A related problem, *diversified top-k graph pattern matching* [10], considers matchings that allow one-to-many pairings from the query to the data graph.

3 PRELIMINARIES

Table 1 lists the main notations we employ. A *labelled graph* is a triple $G : (V, E, L)$ on a set of nodes V, a set of edges $E \subseteq V \times V$, and a labelling function $L : V \rightarrow \Sigma$ that assigns to each node a label from a label set Σ. The *neighborhood* of a node u, $\mathcal{N}(u) = \{v \in V | (u, v) \in E \vee (v, u) \in E\}$, is the set of nodes adjacent to u by incoming or outgoing edges. The *degree* of a node $u \in V$ is the size of its neighborhood, $d(u) = |\mathcal{N}(u)|$. An *induced subgraph* of a graph G over a set of nodes S is the graph $G[S] = (S, E_S, L)$ where E_S is the set of edges in G that have both endpoints in S, $E_S = \{(u, v) \in E | u, v \in S\}$. A graph $G_1 : (V_1, E_1, L_1)$ is **isomorphic** to $G_2 : (V_2, E_2, L_2)$, i.e., $G_1 \equiv G_2$, if there exists a *bijective mapping function* $f : V_1 \rightarrow V_2$ such that for each node $v \in V_1$ there exists a node $f(v) \in V_2$ with $L_1(v) = L_2(f(v))$, and for each edge $(u, v) \in E_1$ there exists an edge $(f(u), f(v)) \in E_2$. A graph $G_1 : (V_1, E_1, L_1)$ is **subgraph isomorphic** to $G : (V, E, L)$, i.e., $G_1 \sqsubseteq G$, if there exists a subgraph of G isomorphic to G_1. A *subgraph matching query* on a graph G, expressed by a labelled graph $Q : (V_q, E_q, L_q)$

with set of nodes V_q, set of edges $E_q \subseteq V_q \times V_q$ and labelling function $L_q : V_q \rightarrow \Sigma$, seeks subgraphs of G isomorphic to Q, with answers $\mathcal{A}(Q) = \{G' \sqsubseteq G | Q \equiv G'\}$.

Table 1: Notations.

Symbol	Meaning
$G = (V, E, L)$	Graph, V nodes, E edges, L labelling function
$Q = (V_q, E_q, L_q)$	Query, V_q nodes, E_q edges, L_q labelling function
$\mathcal{N}(u)$	Neighbors of node u
$\mathcal{N}^+(u)$	Forward Neighbors of node u
$\mathcal{N}^-(u)$	Backward Neighbors of node u
CS /ADS	Candidate Space/Auxiliary Data Structure
$C(q)$	Candidate Set of q
(u_{ID}, v_{ID})	Candidate node
ϕ	matching order
M	(partial) matching/embedding
M_s^*	symmetric (partial) matching/embedding to M
$\mathcal{T}_M(u,v)/\mathcal{T}_M^*(u,v)$	set of matchings/subset of matchings
$\mathcal{T}_M(u, v_i)$	set of (partial) matchings of M extended at (u, v_i)
$\mathcal{T}_v^*$	coverage (unique nodes) for $\mathcal{T}_M^*(u,v)$
$\pi(u,v)/\pi^*(u,v)$	equivalence cell/forward equivalence cell
$\pi_M^-(u,v)$	negative cell
$\pi_M^+(u,v)$	positive cell
$\pi_M(u, v_i)$	equivalence cell at $M(u, v_i)$

Problem 1 (Subgraph Matching). *Given a* data graph $G : (V, E, L)$ *and a* query graph $Q : (V_q, E_q, L_q)$, *return all subgraphs* $G' = (V', E', L')$ *isomorphic to* Q.

In the following, we assume G to be *undirected*, i.e., that $(u, v) \in E$ implies $(v, u) \in E$; however, our discussion readily adapts to the directed case, as well as to *edge-labelled* graphs.

Filtering. A *candidate vertex* for a query graph vertex (i.e., *query vertex*) u_i is a data graph vertex (i.e., *data vertex*) v_j that might be a mapping of u_i. We denote the set of candidate nodes for u_i as $C(u_i) \subseteq V$. Subgraph matching algorithms [37] reduce candidates for a query vertex via *filtering rules* based on the query and data vertex degrees, labels, and neighborhood information.

Auxiliary Data Structure (*ADS*). Candidate vertices are organized in an (*ADS*), also called *candidate space* (CS), where each edge in the query graph and the data graph is represented.

Definition 1. *An ADS is a graph* $CS = (V_q \times V, (V_q \times V) \times (V_q \times V))$ *having a node* (u, v) *for each query node* $u \in V_q$ *and each candidate node* $v \in C(u)$ *and an edge* $((u, v)), (u', v'))$ *if and only if there exists an edge* $(u, u') \in E_q$ *in the query graph and a corresponding edge* $(v, v') \in E$ *between the candidate node* $v \in C(u)$ *and* $v' \in C(u')$ *in the data graph.*

With a slight abuse of notation, we denote the set of neighbors of the query node $u \in V_q$ and candidate node $v \in C(u)$ as $\mathcal{N}((u, v)) = \{(u', v') \in V_q \times C(u) | (u, u') \in E_q \wedge (v, v') \in E\}$. We say that two candidate nodes $v \in C(u), v' \in C(u), v \neq v'$ for query node u *share neighbors* if their neighborhoods in the candidate space are the same, i.e., $\mathcal{N}((u, v)) = \mathcal{N}((u, v'))$.

An *equivalence cell* $\pi(u, v) = \{v' \in C(u) | \mathcal{N}((u, v)) = \mathcal{N}((u, v'))\}$ is a set of candidate nodes $v' \in C(u)$ for the query node u that share neighbors with candidate node $v \in C(u)$.

Matching process. A **partial matching** M is an isomorphic mapping from an induced subgraph of the query Q to a subgraph of the data graph G. The *matching order* ϕ is a permutation of the query nodes V_q by which the algorithm matches them, typically obtained by traversing the query in depth-first search (DFS) or breadth-first search (BFS) fashion, inducing a *traversal subtree* over query edges.

Let $\mathcal{T}_M(u, v_i)$ denote a set of partial matchings containing partial matching M and expanded over a traversal subtree rooted at $\{(u, v_i)\}$. Figure 3 shows an example with the partial matching $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_7)\}$ expanded to a set of partial matchings over a traversal subtree rooted at $\{(u_4, v_8)\}$.

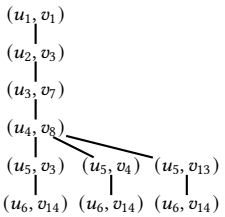


Figure 3: Subtree rooted at $\{(u_4, v_8)\}$.

Definition 2 (Symmetric partial matchings[19]). *Given a partial matching* M^* *expanding partial matching* M *over a traversal subtree rooted at* $M \cup \{(u, v_i)\}$, *another partial matching* M_s^* *is* symmetric *to* M^* *with respect to data node* v_j *provided that:*

(1) if v_j *is not mapped in* M^*, *then* $M_s^* = M^* \setminus \{(u, v_i)\} \cup \{(u, v_j)\}$,
(2) if v_j *is mapped to* u' *in* M^*, *then* $M_s^* = M^* \setminus \{(u, v_i), (u', v_j)\} \cup \{(u, v_j), (u', v_i)\}$.

We say that two traversal subtrees are *equivalent* iff every partial matching found in the one is also symmetrically found in the other.

Definition 3 (Equivalent traversal subtrees [19]). *Two traversal subtrees expanding partial matching* M *and rooted at* $\{(u, v_i)\}$ *and* $\{(u, v_j)\}$ *are* equivalent *provided that:*

(1) if $\mathcal{T}_M(u, v_i) = \emptyset$ *then* $\mathcal{T}_M(u, v_j) = \emptyset$.
(2) if $\mathcal{T}_M(u, v_i) \neq \emptyset$ *then for all* $M^* \in \mathcal{T}_M(u, v_i)$ *there exists partial matching* $M_s^* \in \mathcal{T}_M(u, v_j)$ *symmetric to* M^* *at* v_j, *and vice versa.*

While finding matching nodes, we may encounter *conflict nodes* v_i already mapped to M. For example, in Figure 3, node v_3 is matched in both (u_2, v_3) and (u_5, v_3), entailing in a conflict. Let $I_M(u, v_i)$ be the set of all mappings (u', v_i) that conflict with (u, v_i) at v_i in the subtree rooted at $M \cup (u, v_i)$. Similarly, let $O_M(u, v_i)$ be the set of all mappings (u', v') visited by the algorithm in the subtree rooted at $M \cup (u, v_i)$ such that $v_i \notin \pi(u', v')$. The two equivalence cases in Definition 3 steer the exploration away from a subtree based on the presence of matches. To facilitate this operation, we distinguish two types of equivalence cells $\pi(u, v)$: *negative cells* $\pi_M^-(u, v)$, which track symmetric nodes of a subtree without any match, and *positive cells* $\pi_M^+(u, v)$, which track symmetric nodes of a subtree with at least one match. A negative cell $\pi_M^-(u, v)$ is initially equivalent to the shared neighbors $\pi(u, v)$ and gets refined during exploration. If a subtree rooted at (u, v) is fully evaluated without a match or conflict, then $\pi_M^-(u, v)$ remains unchanged. However, in case a conflict arises, we exclude from $\pi_M^-(u, v)$ nodes that break equivalence symmetry and retain only those that have symmetric counterparts in $\pi(u', v)$ at lower levels of the tree, enabling the detection of any matches entailed by broken symmetries.

Definition 4. *[19] A negative equivalence cell is* $\pi_M^-(u, v_i) = \pi(u, v_i)$ *if there is no conflict in the subtree rooted at* $M \cup (u, v_i)$. *If a conflict arises at* v_i *in the subtree rooted at* $M \cup (u, v_i)$, *the negative cell becomes* $\pi_M^-(u, v_i) = \pi(u, v_i) \cap \left(\bigcap_{(u', v_i) \in I_M(u, v_i)} \pi(u', v_i)\right)$.

A positive equivalence cell extends a negative cell by also tracking whether a node discovered during matching (or its symmetric counterpart) belongs to $\pi^-_M(u, v_i)$. In such cases, a conflict arises, hence we should not report a symmetric match. For instance, assume $\pi^-_M(u_3, v_7) = \{v_7, v_4\}$ in Figure 3, and the algorithm finds a match $M = \{(u_1, v_1), (u_2, v_3), (u_4, v_8), (u_5, v_4), (u_6, v_{14})\}$. Then we cannot report a symmetric match on (u_3, v_4) as a conflict arises between (u_3, v_4) and (u_5, v_4).

Definition 5. *[19] A positive equivalence cell $\pi^+_M(u, v_i)$ is $\pi^-_M(u, v_i) \backslash \delta_M(u, v_i)$ where $\delta_M(u, v_i) = \bigcup_{(u', v') \in O_M(u, v_i)} \pi(u', v')$.*

If the evaluation of the subtree rooted at $M \cup (u, v_i)$ does not yield any match, then we prune equivalent nodes in $\pi_M(u, v_i)$, otherwise, if it does, we report their symmetric counterparts by means of $\pi^+_M(u, v_i)$.

Definition 6. *[19] The* equivalence cell $\pi_M(u, v_i)$ *on M is $\pi^-_M(u, v_i)$ if $\mathcal{T}_M(u, v_i) = \emptyset$, otherwise $\pi^+_M(u, v_i)$.*

4 COVERAGE-ORIENTED MATCHING

For a query Q and a data graph G, subgraph matching returns all the matchings $\mathcal{T}_{M(u,v)}$ of the query in the data graph. The set of solutions contain data vertices $\mathcal{T}_v$ of V that belong in the matchings $\mathcal{T}_{M(u,v)}$. In diversity we maximize the coverage defined as follows:

Definition 7 (Absolute Coverage [43]). *Given a set of matchings $\mathcal{T}^*_{M(u,v)}$ of subgraphs of G, the coverage of $\mathcal{T}^*_{M(u,v)}$, denoted as $\mathcal{T}^*_v$, is the set of all unique vertices in $\mathcal{T}^*_{M(u,v)}$.*

When the algorithm returns all matchings, it maximizes the extent of coverage $|\mathcal{T}_v|$; however, such complete coverage is rare. Current subgraph matching algorithms [42] return a subset of matchings, $\mathcal{T}^*_M(u, v)$, under a limit on their number (e.g., 10^5) or on elapsed time. $\mathcal{T}^*_M(u, v)$ is a subset of $\mathcal{T}_M(u, v)$.

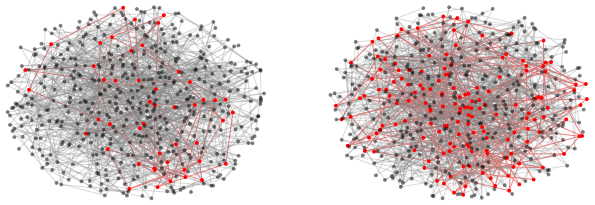


Figure 4: Transaction network with 500 nodes; results for LFTJ (left) and M&M-I (right) for size 16 and 10 matchings; nodes participating in matching colored in red.

Unfortunately, such early termination biases the solution. Figure 4 shows the results of 10 matches with a cycle query of size 16, found by LFTJ algorithm and our proposed M&M-I, colored in red, in a transaction network where nodes are bank accounts and edges represent money transfers. A cycle in such a graph may indicate a money laundering transaction, hence such query help data analysts identify accounts engaged in illicit activities. Figure 4 shows that a classic subgraph matching solution returns many several matchings with repeated fraudulent accounts and fails to capture other accounts participating in illicit operation. On the other hand, a coverage-oriented algorithm captures more bank accounts. While backtracking-based approaches can return millions of matchings in a small time limit and scale to graphs of more than 10 nodes by exploiting locality, they do so by recombining the same nodes, leading to matchings with high overlap and an incomplete view of query results. If we were to eschew backtracking and try to minimize overleap, we would discover fewer matches with fewer unique nodes in total. Our solution lies between these two extremes: it uses backtracking to find matchings, yet sidesteps overlapping ones and favors regions where matchings have less overlap.

Obtaining high coverage on subgraph matching queries is challenging, as it remains unclear how many matches would cover all distinct regions of the graph. The problem is exacerbated by the depth-first-search nature of backtracking algorithms, which generate all answers from one search-space subtree before moving on to another. Grounded on these observations, we propose a solution, M&M, that strives to maximize coverage within a time limit, i.e., the number of unique nodes $|\mathcal{T}_v|$ of partial matchings $\mathcal{T}^*_{M(u,v)}$ under a time limit T, leading to the following problem.

Problem 2 (Coverage-oriented matching). *Given a query Q and a graph G, return the largest set $\mathcal{T}_v$ under a time limit T:*

$$\max_{\mathcal{T}^*_{M(u,v)} \subseteq \mathcal{T}_{M(u,v)}} \left| \mathcal{T}_v = \bigcup_{\mathcal{M} \in \mathcal{T}^*_{M(u,v)}} \{v | (\cdot, v) \in \mathcal{M}\} \right| \text{ subject to } \texttt{time}\left(\mathcal{T}^*_{M(u,v)}\right) \leq T \tag{1}$$

The solution to Problem 2 leads to an unbiased representation of query results.

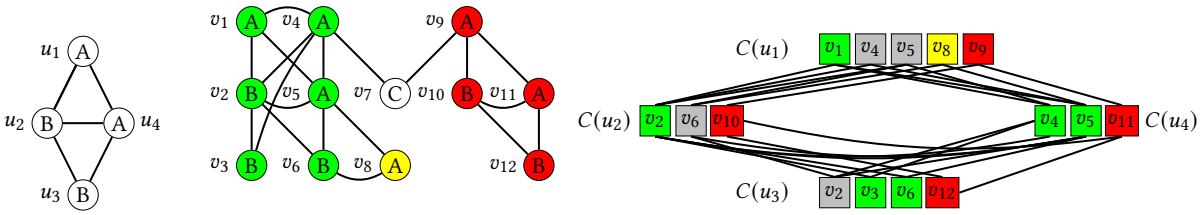
4.1 M&M: Algorithm Outline

We introduce Mix & Match (M&M), a subgraph matching algorithm that enlarges coverage by exploring more graph regions than conventional subgraph matching algorithms within the same time. While the attained coverage is orthogonal to filtering, it is vital for time-efficiency to eschew evaluating candidates that do not lead to a match via effective filtering. To that end, we apply the VEQ [19] filtering method, but any state-of-the-art filtering technique may also be used. To ensure extensibility, we build M&M on top of a hybrid Leapfrog Triejoin (LFTJ)-like enumeration method, which selects among set intersection algorithms based on cardinality skew [37], and use failing sets [13] to prune paths that do not lead to a matching. We employ the Core-Forest-Leaf decomposition (CFL) [3] to order nodes and the *index-based subgraph matching algorithm with general symmetries* (ISMAGS) [17], as in [47], to eliminate automorphic results.

To enhance coverage, we revamp both *global* and *local exploration*. In *global exploration*, we prioritize previously unexplored areas of the data graph when selecting each starting node v_i for the first query node $\phi(0)$. In *local exploration*, which focuses on the area surrounding a data node, we prioritize matching subgraphs that contain unique nodes. If a branch of the matching tree is not due to yield new unique nodes, we abandon its exploration and shift to the next branch that may yield new unique nodes.

4.2 Global Exploration

To prioritize previously unexplored areas of the data graph, we reorder the exploration of candidates v for the first query node $\phi(0)$. We illustrate this prioritization with an example: Figure 5 shows



(a) Query Q. **(b) Graph G.** **(c) Candidate space.**

Figure 5: An example of a query graph, a data graphs and its candidate space

a query graph Q (a), a data graph G (b), and the corresponding candidate space (c). Assume that the backtracking order is $\phi : \langle u_1, u_2, u_3, u_4 \rangle$. The selection of a candidate for query node $\phi(0) = u_1$ determines the area of the data graph to be evaluated. This selection is critical, as in large-size or time-limited queries the algorithm may fail to evaluate all candidate nodes for $\phi(0)$. The enumeration algorithm starts backtracking from (u_1, v_1) in the candidate space and evaluates all matchings that contain v_1 as a mapping for u_1. After exploring all subtrees starting from (u_1, v_1), it selects the next candidate node for u_1, in this case (u_1, v_4). Selecting a next candidate node from the same graph area is likely to yield matchings overlapping those already reported, while choosing from different region enhances coverage. For instance, when evaluating (u_1, v_1), we find the matching $\{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4)\}$, and $\{(u_1, v_1), (u_2, v_2), (u_3, v_6), (u_4, v_5)\}$, hence discover the nodes $v_1, v_2, v_3, v_4, v_5, v_6$, colored green in Figure 5. If the next choice were (u_1, v_4), it would not reveal any new nodes, as u_1 determines which graph region we explore. However, if we select another candidate, e.g., (u_1, v_9), we identify the red-colored matches, enhancing coverage. After exploring the subtree rooted at (u_1, v_9), selecting (u_1, v_8) next uncovers the remaining unique nodes forming matchings (in yellow). Conventional subgraph matching algorithms disregard the order in which they process candidate vertices for $\phi(0)$, as reshuffling that order does not yield additional matchings. The following definition guides our exploration.

Definition 8 (Strongly Uncovered Node). *A candidate node v_i is* strongly uncovered *if matchings including $(u_{\phi(0)}, v_i)$ do not overlap any previously discovered matching.*

Determining whether a candidate node is *strongly uncovered* requires evaluating all its matchings, compromising efficiency. We eschew this costly operation and simply drive the exploration to (weakly) *uncovered* candidate nodes, i.e., nodes that do not overlap with any already discovered matching. Overall, we consider four criteria to select such nodes with small computational effort:

- **Uncovered (U)** selects nodes that do not belong to an already discovered matching, i.e., in $\mathcal{T}_v^*$.
- **Minimum covered Neighbors (mN)** selects uncovered nodes with the fewest covered neighbors.
- **Maximum uncovered Neighbors (MN)** selects uncovered nodes with the most uncovered neighbors, hence towards higher-degree nodes than mN does.
- **Hybrid (HN)** selects nodes by combining mN and MN.

To apply these criteria, we create a list of *unique* candidate nodes UL to be evaluated for $\phi(0)$ that are not part of any already discovered matching. Any of those nodes satisfies the U criterion. mN also accounts for already covered neighbors, which indicate proximity to already discovered matchings. To apply the mN criterion, we iterate over UL to find and return the node with the fewest *covered* neighbors, $\arg\min_{v \in UL} |\mathcal{N}(v) \cap \mathcal{T}^*|$, resolving ties arbitrarily. For instance, after exploring (u_1, v_1), UL contains (u_1, v_8), (u_1, v_9), as v_4 and v_5 already exist in the solution set of (u_1, v_1). Since (u_1, v_8) has two covered neighbors (v_5, v_6), while (u_1, v_9) has none, we return (u_1, v_9) by mN. An extension of this criterion to k^{th}-order neighborhoods would more perspicaciously capture each candidate's position in relation to already discovered matchings, yet would also increase computational cost; thus, we restrict the view to each node's immediate neighborhood. To apply MN, we iterate over UL to find and return the candidate node with the most *uncovered* neighbors, $\arg\max_{v \in UL} |\mathcal{N}(v) \setminus \mathcal{T}^*|$, resolving ties arbitrarily. For instance, (u_1, v_8) has no unique neighbor, while (u_1, v_9) has v_{10} and v_{11}, hence we select (u_1, v_9) by MN.

The HN criterion combines mN and MN as Algorithm 1 shows. We iterate (Lines 2–3) over the list UL of candidate nodes that do not belong to previously found matchings, counting the unique (U) and previously seen duplicate (D) adjacent neighbors of each (Line 4). We return a candidate with no adjacent duplicates (Lines 10–11), if such exists, which reliably leads to an unexplored region, otherwise the one with the most unique adjacent candidates (Lines 12–14).

Algorithm 1 *HN*

```
1:  procedure HN(T, CS)
2:      UL ← UNIQUE(T, CS(u_π(0)))                ▷ UL is the list of unique nodes
3:      for i = 0, ..., |UL| do                   ▷ Iterate over all elements of UL
4:          U = 0, D = 0                          ▷ U: unique candidates, D: duplicates
5:          for j = 0, ..., N(UL[i]) do           ▷ Visit all neighbors of UL[i]
6:              if j ∈ T then                     ▷ Check if the node exists in the set of matchings
7:                  D++
8:              else
9:                  U++
10:         if D = 0 then
11:             return UL[i]                      ▷ if all neighbors are unique, return node
12:         if U > max then                       ▷ keep track of the node with the most unique neighboors
13:             max = U, maxL = i
14:     return UL [maxL ]                         ▷ return the node with the most unique neighboors
```

While we focus on selecting candidate nodes for $\phi(0)$, in scenarios with very large query sizes where only one node may be evaluated for $\phi(0)$, we may apply our criteria in lower levels of the backtracking tree, when selecting for $\phi(1)$, $\phi(2)$, etc. Overall, global exploration is attractive in cases where there are many candidate nodes for $\phi(0)$ and the backtracking tree for $(u_{\phi(0)}, v_i)$ allows for evaluating many of those candidates.

Complexity analysis. Each lookup to the hashing table $\mathcal{T}_v^*$ takes $O(1)$ time and $O(|G|)$ memory. Evaluating whether each $C(\phi(0))$ candidate's neighbor belongs to a matching takes $O\left(|C(\phi(0))| \cdot \deg(C(\phi(0)))\right)$ repeated at most $|C(\phi(0))| - 1$ times.

4.3 Initialization with low-overlap matchings

We propose the global exploration procedure as a low-cost solution for any query size. When the query graph is small, which is likely to yield more matching results, we initialize the exploration by identifying a set of disjoint or minimally overlapping matchings at a small cost and then run the DFS-based backtracking algorithm to identify harder matchings. To that end, we add two rules that guarantee we find matchings with at most one overlapping node. First, upon detecting a matching, we report it and do not explore alternative extensions. Instead, we choose a new candidate node for $\phi(1)$. We repeat this process to evaluate all candidate nodes for $\phi(0)$. Second, when selecting nodes after $\phi(0)$, we choose only nodes not belonging to a previously discovered matching $\mathcal{T}_v^*$.

Complexity analysis. In the worst case, we face the subgraph matching time complexity of $O(|G|^{|V_q|})$.

4.4 Local Exploration

After we select a candidate node for $\phi(0)$, we explore matchings *locally* in its neighborhood delimited by the query size. As exploring all backtracking branches starting from the $(u_{\phi(0)}, v_i)$ assignment is exponential in query size, we explore only a few such assignments to enlarge coverage on large query graphs. To favor exploration that leads to new unique nodes, we avoid exploring *symmetric* branches by Definition 2. Given an embedding M^* rooted at $M \cup \{(u, v_i)\}$ and a symmetric embedding M_s^* rooted at $M \cup \{(u, v_j)\}$, the matchings in $\mathcal{T}_{M^*}$ will be the same as those in $\mathcal{T}_{M_s^*}$, except for substituting node v_i for v_j, hence only v_j may be a new node. Besides, if v_j has already been reported in a matching in $\mathcal{T}_{M^*}$, then the extension of M_s^* yields no new nodes. VEQ [19] leverages symmetries to avoid exploring equivalent paths at runtime. In local exploration, we capture larger equivalent subtrees than VEQ and eschew reporting symmetric matches that do not yield new unique nodes.

4.4.1 Enhanced Equivalences. VEQ [19] separates equivalence cells into positive π_M^+ and negative ones π_M^- (Definition 6), taking in consideration conflicts and overlapping matchings, pruning trees that do not lead to matchings and reporting matchings in trees that do so, without exploring them. In effect, it removes nodes from the initial equivalence cell π_M that violate equivalence symmetry. Equivalence symmetry starts by identifying candidate nodes that share neighboors, hence we initially populate the equivalence cell $\pi_M(u_i, v_j)$ with all nodes having the same neighbors as (u_i, v_j) *in the candidate space.*

We define the *forward neighbors* of u_i, $\mathcal{N}^+(u_i)$, as the neighbors that we explore after u_i, and the *backward neighbors* of u_i, $\mathcal{N}^-(u_i)$, as those that we visit before u_i, in the backtracking order.

Previous work VEQ [19] defines equivalences via symmetry in both forward and backward neighbors. However, we only need to check for symmetry in the tree that comes after M, thus do not need to check backward neighbors. We denote nodes as equivalent in terms of their forward neighbors using π^*. Regarding backward neighbors, it suffices to know the partial matching M being extended and eliminate candidates having different previous neighbors. To that end, we intersect the local candidates (LC) in $M \cup \{(u_{i-1}, v_l)\}$ with those in $\pi^*(u_i, v_j)$ to get the equivalent candidates for (u_i, v_j). Besides, we need not calculate equivalences for the last query node $u_{|V_q|}$, as there are no forward neighbors to consider; the final equivalence set simply comprises the candidate nodes for $u_{|V_q|}$. Figure 6 presents the candidate space for a query and a data graph and their equivalence sets. The table on the left shows equivalences, sorted by cell size, when considering all neighbors to calculate cell π, while the one on the right shows equivalences when considering only forward neighbors to calculate cell π^*. Equivalent nodes are assigned the same ID during the check. For instance, the equivalence cell $\pi(u_5, v_4)$ (ID 12) contains $\{v_4, v_{13}\}$, as these two nodes have the same backward candidate-space neighbors, $\mathcal{N}^-(u_5, v_4) = \mathcal{N}^-(u_5, v_{13}) = \{(u_3, v_7), (u_3, v_8), (u_3, v_9)\}$, and forward candidate-space neighbors, $\mathcal{N}^+(u_5, v_4) = \mathcal{N}^+(u_5, v_{13}) = \{(u_6, v_{14}), (u_6, v_{15})\}$. On the other hand, the corresponding forward equivalence cell is $\pi^*(u_5, v_4) = \{v_3, v_4, v_{13}\}$, as $\mathcal{N}^+(u_5, v_4) = \mathcal{N}^+(u_5, v_3) = \mathcal{N}^+(u_5, v_{13}) = \{(u_6, v_{14}), (u_6, v_{15})\}$. Notably, the forward equivalence encompasses v_3, while the regular equivalence $\pi(u_5, v_4)$ misses it, because (u_5, v_3) lacks backward neighbor (u_3, v_9) which (u_5, v_4) has, even though they share backward neighbors (u_3, v_7) and (u_3, v_8). In effect, candidate space node (u_5, v_3) yields a subtree equivalent to those of nodes (u_5, v_4) and (u_5, v_{13}) when extending a partial matching having chosen the match (u_3, v_7) or (u_3, v_8). When choosing the match (u_3, v_9), the equivalence cell $\pi^*(u_5, v_4)$ will superfluously encompass (u_5, v_3), yet as we intersect each equivalence cell π^* with the constructed partial matching M, in this case featuring (u_3, v_9), the candidate (u_5, v_3) is rendered unavailable in M and hence not included in π^*. Overall, by using the current partial matching M and forward neighbors, we capture more equivalent subtrees and improve performance.

4.4.2 Catering to Coverage. The latter part of the algorithm caters to coverage by adapting the rules for equivalent subtrees (Definitions 5 and 6). Given a partial matching $M^* = M \cup \{(u, v_i)\}$ and its symmetric counterpart M_s^* rooted at $M \cup \{(u, v_j)\}$, the latter, which pertains to an already reported subtree, yields v_j as a new unique node iff v_j does not participate in a matching in the former. We add this result to the answer set without exploring M_s^*.

As we eschew counting symmetric matches that yield no new unique nodes, we remove $\delta_M(u, v_i)$ from Definition 5 (π^+-equivalence), which prevents recounting nodes equivalent to v_i. Thus $\pi_M^+(u, v_i) = \pi_M^-(u, v_i) \setminus \delta_M(u, v_i)$ becomes $\pi_M^+(u, v_i) = \pi_M^-(u, v_i)$, rendering positive and negative cells identical, hence we only need to keep track of one forward equivalence cell $\pi_M^*(u, v_i)$ in both scenarios:

Rule 1. *If there is no mapping conflict at v_i, then the forward equivalence cell $\pi_M^*(u, v_i)$ regarding M is the same as $\pi^*(u, v_i)$. If there is at least one mapping conflict at v_i, then $\pi_M^*(u, v_i) = \pi^*(u, v_i) \cap \{\cap_{(u', v_i) \in I_M(u, v_i)} \pi^*(u', v_i)\}$.*

In effect, we modify the equivalence cell only in a conflict scenario. For instance, if we follow the backtracking path $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_7), (u_4, v_8), (u_5, v_3)\}$, a conflict arises between (u_2, v_3) and (u_5, v_3). Their forward equivalence cells are $\pi^*(u_2, v_3) = \{v_3, v_4\}$ and $\pi^*(u_5, v_3) = \{v_3, v_4\}$, hence $\pi_M^*(u_2, v_3) = \pi^*(u_2, v_3) \cap \pi^*(u_5, v_3) = \{v_3, v_4\}$. In effect, path $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_7), (u_4, v_8), (u_5, v_4)\}$ is symmetric to $M^* = \{(u_1, v_1), (u_2, v_4), (u_3, v_7), (u_4, v_8), (u_5, v_3)\}$, which we therefore eschew exploring.

On the other hand, if we follow the backtracking path $M = \{(u_1, v_1), (u_2, v_3), (u_3, v_8), (u_4, v_8)\}$, a conflict arises between (u_3, v_8)

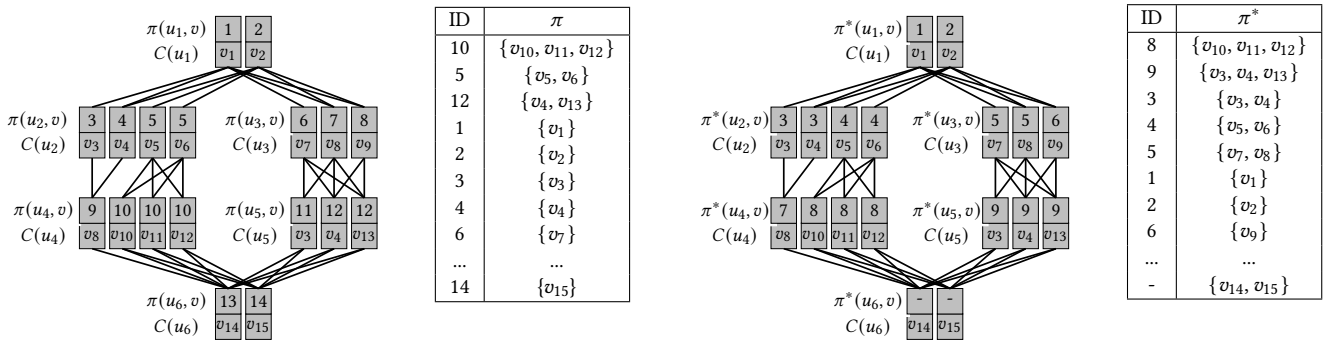
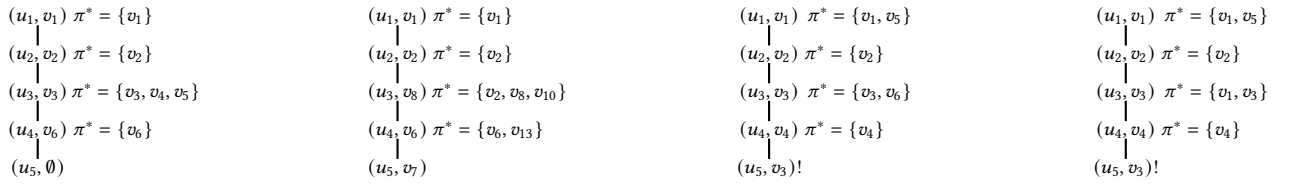


Figure 6: Candidate space with equivalence sets considering all neighboors (π, left) and only forward neighboors (π^*, right).



(a) No matching **(b) Symmetric matchings** **(c) Last level conflict** **(d) Last level conflict**

Figure 7: Matchings for a subtree M; last level conflicts by Rule 2 (c) and Rule 1 (d).

and (u_4, v_8). Their forward equivalence cells are $\pi^*(u_3, v_8) = \{v_7, v_8\}$ and $\pi^*(u_4, v_8) = \{v_8\}$, hence $\pi^*_M(u_3, v_8) = v_8$, excluding v_7, as the path $M^* = (u_1, v_1), (u_2, v_3), (u_3, v_7), (u_4, v_8)$ is not symmetric to M and thus has to be evaluated.

Handling last-level conflicts. We also improve the algorithm when conflicts arise at the last backtracking level. If a node v_j conflicted between an ancestor node and a last-level node is equivalent to a node v_o outside the current matching, then there is at least one matching that uses v_o in the ancestor position and v_j in the last level. Assume $M = \{(u_1, v_i), (u_2, v_j), (u_3, v_k), (u_4, v_l), (u_5, v_j)\}$ with a conflict between (u_2, v_j) and (u_5, v_j), where $\pi^*(u_2, v_j) = \{v_j, v_o\}$ and $\pi^*(u_5, v_j) = \{v_j\}$. A regular iteration would treat the conflict with the rules defined in Definition 4 and would produce the forward equivalence cell $\pi^*_M(u_2, v_j) = \{v_j\}$, evicting v_o, which is not in $\pi^*(u_5, v_j)$, as there may be a different matching anchored on (u_2, v_o). Indeed such a matching exists, using (u_2, v_o) and (u_5, v_j), and the algorithm may discover it processing $M = \{(u_1, v_i), (u_2, v_o)\}$. Being at the last level, we promptly report this equivalence-breaking matching as $M^* = \{(u_1, v_i), (u_2, v_o), (u_3, v_k), (u_4, v_l), (u_5, v_j)\}$ and retain v_o in equivalence cell $\pi^*_M(u_2, v_j)$. Besides M^*, we report all combinations of symmetric matches for equivalent nodes in M by adding to the solution their corresponding equivalents at each level of M^*. The equivalence remains unchanged, since symmetry-breaking matches can only occur at or after the level in which a conflict occurs. At the last level, no other conflict exists beyond the one already reported one, in this case (u_5, v_j).

Rule 2. *If v_j is a conflict node between u_i and u_n, where n is the last level of the backtracking tree and $\{x \in (\pi^*_M(u_i, v_j)) \mid x \notin M\} \neq \emptyset$, then there is at least one matching using (u_i, v_x) and (u_n, v_j).*

On the other hand, there may exist a matching anchored on (u_n, v_j) and symmetric nodes inside M, $\{x \in (\pi^*_M(u_i, v_j)) \mid x \in M\}$. In this case, we have to check all swaps of such nodes with equivalent ones. Thus, if we cannot find a match by Rule 2, we apply Rule 1 for regular levels as usual. If, on the other hand, we find a matching by Rule 2, then we do not need to apply Rule 1, as the application of Rule 2 reports all matchings that would be discovered by the symmetries of M for node v_j in the last level of M.

Example 1. Figure 7 shows four examples of matchings returned by M&M. Figure 7a presents a case where a node in M, here u_5, yields no matching; thus, the equivalent nodes of $(u_3, v_3) \in M$, v_3, v_4, v_5 lead to no match in M. Figure 7b shows a case where M contains symmetric matchings $\{v_2, v_{10}\}$ for $\pi^*_M(u_3, v_8)$ and $\{v_{13}\}$ for $\pi^*_M(u_4, v_6)$. Here, we return the symmetric matchings as alternative solutions to the matching $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_8), (u_4, v_6), (u_5, v_7)\}$ if they include additional unique nodes. For instance, if the algorithm returns $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_8), (u_4, v_6), (u_5, v_7)\}$ and $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_{10}), (u_4, v_{13}), (u_5, v_7)\}$ it does not also return $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_8), (u_4, v_{13}), (u_5, v_7)\}$, which provides no extra coverage. Although node v_2 belongs to $\pi^*_M(u_3, v_8)$, it is excluded from the matchings of u_3 as it is already mapped in $\pi^*_M(u_2, v_2)$. Figure 7c and Figure 7d show examples of last level conflicts between matchings (u_3, v_3) and (u_5, v_3). By Rule 2, the conflict in Figure 7c results in at least one matching $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_6), (u_4, v_4), (u_5, v_3)\}$ using a symmetric node in $\pi^*_M(u_3, v_3)$. We also append to the solution the matching $M = \{(u_1, v_5), (u_2, v_2), (u_3, v_6), (u_4, v_4), (u_5, v_3)\}$ that includes the unique node v_5. Similarly, by Rule 1, the conflict between (u_1, v_1) and (u_3, v_1) in Figure 7d results in the removal of v_1 from $\pi^*_M(u_3, v_3)$ and of v_5 from $\pi^*_M(u_1, v_1)$, and forces a backtracking iteration.

Our algorithm may extend any existing backtracking-based algorithm. Still, we recommend using a candidate space to find the candidate neighbors of candidate nodes and a *static* backtracking order like GQL [16] or CFL [3], so that forward equivalence cell of each candidate remains the same once calculated on the fly. An *adaptive* backtracking order would allow forward neighbors to change, thus we would have to recalculate forward equivalence. If

we remove the time limit, M&M returns the maximum coverage, as it explores all paths than may lead to unique nodes.

Complexity analysis. Computing a cell $\pi(u_i, v_i)$ takes $O(|C(u_i)| \cdot \deg(v_i))$, where $\deg(v_i)$ is the number of forward neighbors. In practice, we stop early upon detecting an edge mismatch and exclude already considered nodes in $C(u_i)$, reducing the time to $O(|C(u_i)|)$. Candidate cells are computed on demand during backtracking rather than during candidate space construction. The final equivalence cell $\pi^*(u_i, v_i)$ in M arises by intersecting $M \cup (u_{i-1}, v_l)$ with a sorted list of node IDs, with complexity $O(|C(u_{i-1})|)$.

Algorithm 2 M&M

```
1: function M&M(Q,CS,φ)
2:     depth ← 0, u ← φ(0)
3:     idx[depth] ← 0, LC[u] ← |C(u)|
4:     while true do
5:         while idx[depth] < |LC[u]| do
6:             if depth = 0 then
7:                 LC ← GlobalExploration(CS, LC, T)                ▷ Global exploration
8:             u ← φ[depth]                                          ▷ Get u by φ
9:             v ← LC[u][idx[depth]]                                 ▷ Get v by LC and v and idx
10:            π*(u, v) ←ForwardEquiv(Q,CS,φ, u, v)                  ▷ Calculate Forward equivalences
11:            if depth ≠ 0 then
12:                π*(u, v) ← intersect_cand(π*(u, v), LC(u))        ▷ Final equivalences
13:            if v ∈ M then                                         ▷ Conflict case
14:                if depth ≠ max_depth − 1 then
15:                    HandleConflict(Q, π*, M)                      ▷ Update π*(u, v) by Rule 1
16:                else
17:                    HandleLastLevelConflict(Q, π*, M, φ, T)
18:                idx[depth] + +; continue                          ▷ Move to next node in LC(u)
19:            idx[depth] + +
20:            if cur_depth = max_depth − 1 then
21:                T ←ReportMatch(T, M, π*)
22:            else
23:                Add (u,v) to M                                    ▷ Add v to matching
24:                depth + +, idx[depth] ← 0                         ▷ Move to next level
25:                LC(u, M) ← ComputeLC(CS,Q, M, depth, u, v, φ)
26:        depth − −                                                 ▷ Backtrack to previous level
27:        Remove (u,v) from M                                       ▷ Remove v from matching of current depth
28:        if depth < 0 then                                         ▷ Terminate
29:            break
30:        PruneSymmetricNodes(LC, π*, depth, idx)                   ▷ Prune equivalent nodes
31:    return T
```

Algorithm 2 outlines M&M, omitting failing sets [13] and automorphism pruning [17]. In Line 7, we iteratively select a node in $\phi(0)$ and call global exploration to reorder the Local Candidates (LC). We then set the values of u and v based on the current M, $\phi(0)$ and LC (Lines 8–9). Next, we compute forward equivalences for (u, v) (Line 10), obtaining their final set by intersecting $\pi^*(u, v)$ with LC(u), except at the first level, where no backward neighbors exist (Line 12). We handle conflicts by Rule 1 and Rule 2, as applicable (Lines 15–17) and move to the next node in LC(u) (Line 18). If no conflict exists at the last level, we report M and its equivalent matchings using π^* (Line 21). Otherwise, we add v to M (Line 23), extend to the next level (Line 24) and calculate local candidates by set intersections [37] (Line 25). After visiting all local candidates at the current level, we backtrack to the previous level (Line 26), remove v from M (Line 27), prune nodes symmetric to v (Line 30), and proceed to the next node for extension (Line 9).

5 ADAPTIVE TOP-k MATCHING

We extend M&M to operate under a limit k to select a set of top-k matches that achieve high coverage *adaptively* during backtracking and return the top-k results within a given time limit.

Problem 3 (top-k matching). *Given a query Q, a graph G, and an integer k, return the largest set $\mathcal{T}_v^*$ under a time limit T and cardinality limit k:*

$$\max_{\mathcal{T}^*_{M(u,v)} \subseteq \mathcal{T}_{M(u,v)}} \left| \mathcal{T}_v^* = \bigcup_{\mathcal{M} \in \mathcal{T}^*_{M(u,v)}} \{v | (\cdot, v) \in \mathcal{M}\} \right| \quad s.\,t. \quad \texttt{time}\left(\mathcal{T}^*_{M(u,v)}\right) \leq T \;\; and \;\; |\mathcal{T}^*_{M(u,v)}| \leq k \tag{2}$$

We extend M&M to solve Problem 3 by a greedy approach that returns matches *adaptively*. This solution scales to large queries and k, overcoming the runtime bottleneck of the algorithm in [43].

5.1 Simple Greedy

In our simple greedy approach, we maintain the top-k matchings in a min-heap sorted by the count of unique nodes they contribute. Upon discovering a matching, we probe a lookup table of $\mathcal{T}_v^*$ to count the new unique nodes it brings; if these exceed the heap's top, we add the matching to the heap. If the heap exceeds size k, we discard the top entry to ensure we retain the top-k matchings. In effect, as the order in which we discover matchings affects the top-k evaluation, while a diversified exploration enhances quality.

Example 2. Figure 6 illustrates the greedy approach with an example where $k = 2$. Initially, $\mathcal{T}_v^*$={}; assuming the first matching is M={(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{10}), (u_5, v_3), (u_6, v_{14})}, we update $\mathcal{T}_v^* = \{v_2, v_5, v_8, v_{10}, v_3, v_{14}\}$, report the symmetric matching $M = \{(u_1, v_2), (u_2, v_6), (u_3, v_8), (u_4, v_{10}), (u_5, v_3), (u_6, v_{14})\}$ due to the equivalence cell $\pi^*(u_2, v_5)$={v_5, v_6}, and add the newly discovered unique node v_6 to $\mathcal{T}_v^*$. Next, we find M={(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{11}), (u_5, v_3), (u_6, v_{14})} and M={(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{12}), (u_5, v_3), (u_6, v_{14})}, which are due to cell $\pi^*(u_4, v_{10})$={v_{10}, v_{11}, v_{12}}, and bring the new unique nodes v_{11} and v_{12}, respectively. Since bringing one new unique node does not exceed the last matching in the solution, we do not add these matchings to the top-k solution. The same reasoning applies to M={(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{10}), (u_5, v_4), (u_6, v_{14})} and $M = \{(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{10}), (u_5, v_{13}), (u_6, v_{14})\}$, which are due to equivalence cell $\pi^*(u_5, v_3)$={v_3, v_4, v_{13}}.

5.2 Lazy Symmetric Greedy

Simple greedy returns symmetric matches immediately, often yielding at most one unique node per match. To enhance top-k coverage, we propose *Lazy Symmetric Greedy*, which leverages *symmetric paths* of local exploration. Given a matching M, if the node v_j in pair $(u_i, v_j) \in M$ is in $\mathcal{T}_v^*$, we replace it with a symmetric node from $\pi_M(u_i, v_j)$ which is not in $\mathcal{T}_v^*$ and do likewise for each node in M to utilize symmetric nodes. Moreover, we *lazily* return all matchings with symmetric nodes only when we visit the last matching of $M \cup (u_{\phi(|V_q|)}, v_i)$, where $\phi(|V_q|)$ is the last candidate node that does not conflict on M. This procedure allows us to identify matchings that bring more unique nodes during backtracking.

Example 3. Figure 6 illustrates the lazy symmetric greedy with an example. We first report matching M={(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{10}), (u_5, v_3), (u_6, v_{14})}. Conventionally, the next matching would be $M = \{(u_1, v_2), (u_2, v_5), (u_3, v_8), (u_4, v_{10}), (u_5, v_3), (u_6, v_{15})\}$, bringing only one a new unique node, v_{15}. Instead, we report equivalent matches for each node that is already in $\mathcal{T}_v^*$, aiming to enlarge coverage. As the equivalence cell of $\pi^*(u_1, v_2)$ is v_2, we consider $M = \{(u_1, v_2)\}$. On the other hand, considering $\pi^*(u_2, v_5) = \{v_5, v_6\}$, given that $v_5 \in \mathcal{T}_v^*$, we choose $v_6 \notin \mathcal{T}_v^*$. With regard

to $\pi^*(u_3, v_8) = \{v_7, v_8\}$, we cannot extend v_7 as a valid backward neighbor, hence select v_8 resulting in M=$\{(u_1, v_2), (u_2, v_6), (u_3, v_8)\}$. The same logic applied on $u_3, \cdots, u_n$, yields $M = \{(u_1, v_2), (u_2, v_6), (u_3, v_8), (u_4, v_{11}), (u_5, v_4), (u_6, v_{15})\}$, with four new unique nodes, v_6, v_{11}, v_4, v_{15}.

6 EXPERIMENTS

We ran experiments on an Intel Core i9-10940X machine @3.30GHz with 378GB RAM running Linux 4.15.0, with all algorithms implemented[1] in C++, extending the testing framework of [47].

Datasets. We use 7 experimental datasets exhibiting diverse characteristics in terms of number of nodes, labels, and average degree. Table 2 summarizes data characteristics.

- Yeast (YE) [2], a protein interaction network (Figure 2).
- Wordnet (WN) [24], a graph of word semantic relationships.
- DBLP (DB) [42], a co-authorship collaboration network.
- Amazon (AM) [21], Amazon product co-purchasing network
- eu2005 (EU) [4], a crawl of linked .eu pages.
- YouTube (YT) [42], a network among YouTube users.
- US Patents (UP) [22], a large patent-citation network.

Table 2: Data characteristics; number of nodes $|V|$; number of edges $|E|$; number of labels $|\Sigma|$; average degree avg d.

Dataset	Abbr.	$\|V\|$	$\|E\|$	$\|\Sigma\|$	avg d
Yeast	YE	3 112	12 519	71	8
WordNet	WN	76 853	120 399	5	3.1
DBLP	DB	317 080	1 049 866	15	6.62
Amazon	AM	401 692	2 198 486	6	10.94
eu2005	EU	862 664	16 138 468	40	37.4
Youtube	YT	1 132 890	2 987 624	25	5.26
US Patents	UP	3 774 768	16 518 947	20	8.75

Enhancing DSQL [43]. As we will see, our experiments show that DSQL, designed for top-k matching, is not competitive. Still, its disjoint orientation offers an interesting approach, which we build upon to create DSQL+. As in M&M, we use VEQ [19] filtering and LFTJ [37] enumeration with CFL [3] ordering. We also refine DSQL's subroutine [43, pg. 1172] that extends a partial matching, by accounting for the neighbors of a matched node that are already *matched* before considering any other neighbors. This adjustment constrains the other candidates to be examined, hence advances efficiency. DSQL relaxes the overlap constraint in each round, denoted as i-overlap, where i is the number of duplicate elements a matching M is allowed have. For each i-overlap subset [43, Algorithm 3], DSQL evaluates possible *partial matchings* separately, found by extending nodes from obtained matches of previous i-overlaps, leading to redundant backtracking. Contrarily, we evaluate *all* partial matchings for each i-overlap subset in one DFS traversal, shifting the algorithm towards a DFS orientation. While this shift alters the order of discovered matches, the quality of discovered matchings remains unchanged, even while attaining enhanced scalability.

Baselines. We compare M&M and M&M-I to DSQL+, VEQ [19], RM [38], LFTJ [37] and DSQL [43]. In LFTJ we use VEQ filtering and CFL ordering. In DSQL we use VEQ filtering and the LFTJ candidate space. We obtained implementations from the experimental framework in [47], and added the coverage measure for evaluation. We enable the extension from ISMAGS [17], which detects and removes *automorphic* subgraphs of results to improve performance. M&M uses the *mN* approach for global exploration as default. M&M-I uses initialization with low-overlap matchings instead of global exploration. Notably, LFTJ is equivalent to M&M with both global and local exploration removed.

[1]Code: https://github.com/constantinosskitsas/Mix-Match.

Query generation. We generate queries of size {8, 16, 24, 32, 64}; as in previous works [14, 18, 30, 37, 39], we select a node at random and perform a random walk therefrom to collect the desired amount of nodes; if the walk does not reach new nodes in ten consecutive steps, we restart from a node among the visited ones; if we does not discover new nodes in thirty consecutive steps, we return to the initial node and choose an unvisited neighbor to continue. After we collect the desired number of nodes, we add all edges between them to extract an *induced* connected subgraph of the data graph. We generate 100 queries for each size.

Evaluation. We evaluate all algorithms with a time limit of enumeration up to 1 000 seconds and query size up to 64 nodes. We report absolute coverage as the number of unique nodes returned. In the top-k matching experiment, we use k of up to 10 000 matches and a time limit of 10 seconds.

6.1 Effectiveness of Global Exploration

Before our main evaluation, we test our design choices, assessing M&M on four global exploration approaches, *U*, *mN*, *MN*, and *HN*, and two local approaches: *plain*, which selects the next node by ID, and *random*, which selects it randomly. We conduct experiments on 5 datasets with query size of 16 and 24 under a time limit of up to 1 000 seconds. Notably, the plain strategy is widely used in existing subgraph matching algorithms. We expect diminishing returns from global exploration as the query size grows, since the number of candidates in $\phi(0)$ decreases, and the backtracking depth increases, with query size.

Figure 8 plots coverage vs. time limit T. As expected, smaller queries gain higher coverage, as they afford evaluating more candidate nodes for $\phi(0)$. The best performance emerges with the minimum covered neighbors criterion *mN*, closely followed by the hybrid *HN*, both selecting nodes in $\phi(0)$ whose neighborhood remains unvisited. Selecting nodes merely based on their own status by *U* yields fewer unique nodes. *MN* often performs worse than random; this result suggests that is ineffective to disregard how many neighboring nodes have been discovered; it may direct the exploration toward regions with *potential* unique candidate nodes, yet those do not materialize into matching results. For query size 24, the advantage of global exploration is less prominent, due to the reduction in candidates in $\phi(0)$, yet we note improvements with the *mN* and *HN* criteria on the DB and UP datasets, and, to a lesser extent, on YT, EU and AM. Overall, *mN* and *HN* are the criteria of choice, exhibiting the best performance across all datasets.

6.2 Coverage Evaluation

Here, we compare algorithms on the coverage they achieve within a time limit. Figure 9 plots coverage in up to 1 000 seconds and with query size up to 64. M&M and M&M-I generally outperform other methods by virtue of their backtracking enhancements, even while, in rare instances, this advantage is not expressed at the early stages. For some queries of size 8 and 16, which return all matchings within the time limit, extra time does not yield higher coverage. In these

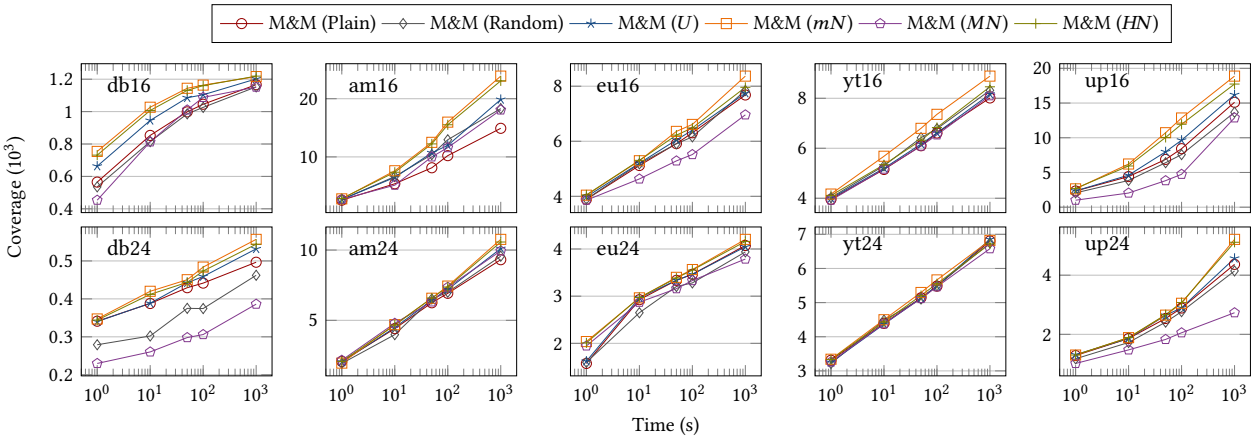


Figure 8: Coverage for M&M variants; query size 16 and 24; time limit T up to 1 000 seconds.

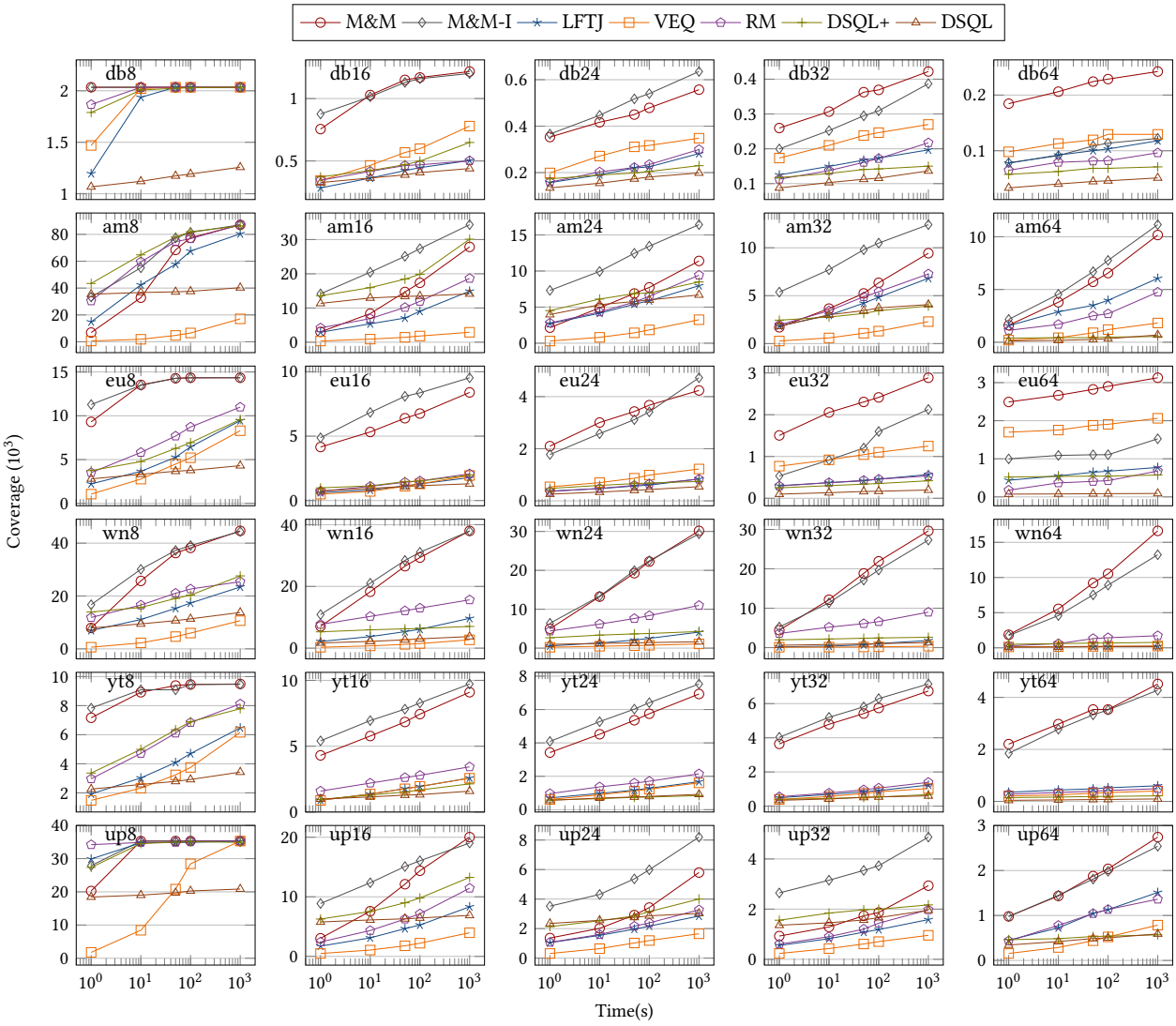


Figure 9: Coverage with query size 8–64; time limit up to 1 000 seconds.

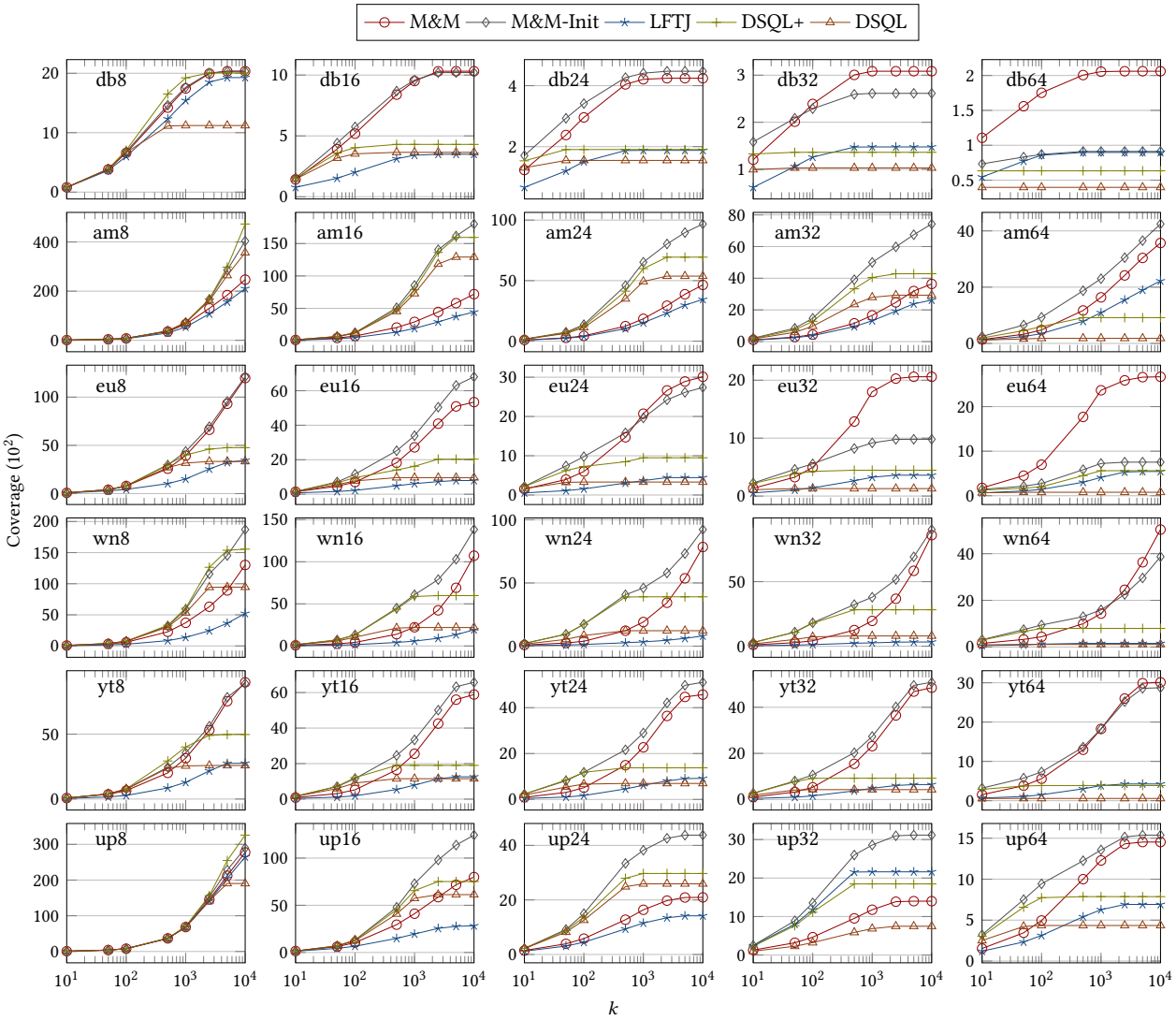


Figure 10: Average unique nodes; query size up to 64 nodes; k up to 10 000; time limit of 10 seconds.

cases, our algorithms discover all unique nodes faster than others, except with the UP dataset and query size 8.

DSQL performs worse for average to large query sizes. As query size increases, its strategy of expanding previous matches struggles to scale. It is the worst performer on DB and EU, second worst on WN, and achieves average performance on AM and UP. Still, our enhanced variant of DSQL, DSQL+ ranks second only to M&M algorithms on small query sizes (8–16) in datasets with many candidates (AM, UP) by quickly identifying matchings with low overlap. However, on larger queries (32-64), where low-overlap matchings are harder to detect, this advantage diminishes and DSQL+ hardly scales, despite our enhancements. After the initial node collection, it struggles to increase the number of unique nodes, even with extended time limits. Consequently, basic subgraph matching algorithms outperform DSQL+ given enough time.

Subgraph Matching Algorithms lack a coverage orientation, yet their coverage grows with time; an exception is VEQ, which performs poorly on coverage, as it prioritizes the output of symmetric matchings which do not advance coverage. These algorithms outperform DSQL+ on large queries and with extended time limit. VEQ surpasses LFTJ and RM on DB and EU, while RM outperforms both on WN. RM is marginally better than LFTJ on small data, and LFTJ slightly better for larger queries.

M&M and M&M-I. Our algorithms consistently outperform previous methods, often doubling coverage. This gain stems from backtracking enhancements in local exploration and becomes more pronounced with larger time limits. On small queries and datasets with abundant matches (AM, UP), the initialization with nearly disjoint matchings gives M&M-I an edge over M&M at shorter time limits. However, M&M catches up given more time. On larger queries, enforcing near-disjoint matchings may hinder performance, as observed with DB and EU. We recommend M&M-I on small queries; on medium-sized queries, we advise selecting between M&M and M&M-I based on candidate count—using M&M-I when candidates are abundant, otherwise M&M. On large queries, M&M is preferred as it avoids complications arising from disjoint matchings.

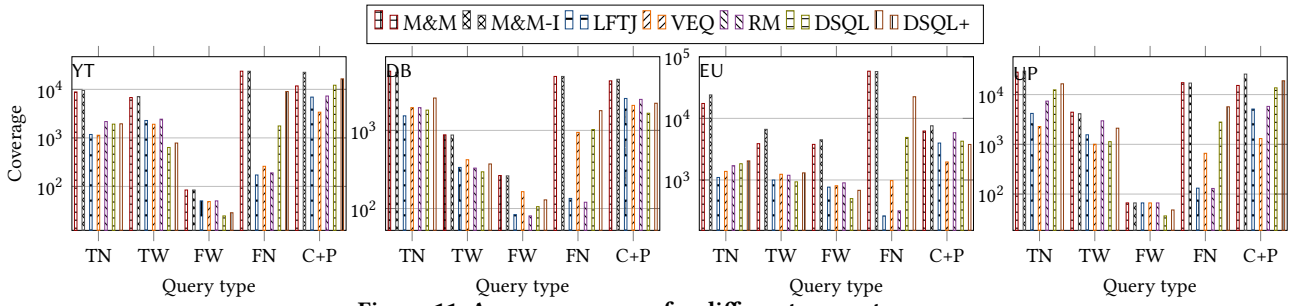


Figure 11: Average coverage for different query types.

6.3 Evaluation on top-k Matchings

We now we compare our algorithms on top-k matchings to the basic baseline, LFTJ, equipped with the greedy top-k matching method as well as DSQL and its enhanced variant, DSQL+. Figure 10 plots coverage vs. k under a 10-second time limit. M&M and M&M-I consistently outperform other methods in this setting. As k grows, the number of unique nodes rises and then plateaus, revealing the minimum k required to detect all unique nodes. An effective top-k algorithm should thus achieve full coverage with a small k. A greedy algorithm finds most unique nodes within relatively few matchings ($\simeq$ 10 000), which allows an analyst to work with thousands rather than hundreds of millions of results and enables efficient post-processing. Within 10 seconds, M&M and M&M-I return on average 93% and 95% of all unique nodes found in the 10 000 top matchings. Besides, for small query sizes and low k, M&M-I finds nearly as many disjoint matchings as DSQL, even though it not explicitly optimized for top-k matching. This highlights the effectiveness of the nearly disjoint initialization in M&M-I.

LFTJ. Unsurprisingly, LFTJ, which we include for the sake of completeness, performs the worst in top-k matching as it does not exploit disjoint matchings, symmetric matchings, or coverage enhancements. Nevertheless, LFTJ performs well with query size 8 and $k = 10$ on the DB, AM, EU, and UP datasets. For k in 100–1 000, it reveals its weakness as it cannot find solutions with low overlap. Yet as the query size and k grow, LFTJ matches the performance of DSQL+ on the DB, EU, and YT datasets, even with the small time limit of 10 sec, due to the scalability bottleneck of an i-overlap disjoint extension method against a backtracking-based approach.

DSQL. The results reconfirm that DSQL provides good results only for small query size and small top-k. On the other hand, **DSQL+** performs well on the top-k problem for small to average k (up to 100) and small to average queries, which yield many disjoint matches, as with the AM, WN, and UP datasets, thanks to its disjoint approach. Still, it faces a scalability bottleneck on large query sizes, even with our enhancements, as the scarcity of low-overlap matches forces it to recompute search trees at greater depths.

M&M and M&M-I. Both these approaches perform the best on top-k evaluation on most instances, while the best choice among the two depends on query size, k and dataset. The key difference is that, while M&M-I starts by finding all the disjoint matches, M&M directly applies global exploration with the enhanced backtracking process to enlarge coverage. In effect, M&M-I performs better in most scenarios, as obtaining disjoint matches attains a compact representation. It is advisable to use M&M-I when attainable. However, it is hard to apply M&M-I with large query sizes, with which the cost of obtaining disjoint matches dominates runtime, while most matchings that contribute new unique nodes arise from overlapping matches. Both are relatively scalable, as they exploit our enhancements in backtracking to explore a larger graph area than other algorithms in the same time. On the DB dataset, M&M and M&M-I perform similarly with small and average size queries and outperform competitors. For such query sizes, M&M-I achieves a diverse representation based on a few matchings. With larger query sizes (i.e., 32 and 64), M&M emerges as the algorithm of choice, as M&M-I incurs a high cost to obtain nearly disjoint matches. The same challenge of obtaining nearly disjoint matchings with large queries appears on the EU dataset. On the YT dataset, with a large k, both algorithms perform equally well, whilst with smaller k M&M-I attains more expansive top-k solutions. On datasets yielding many candidates, such as AM, WN, and UP, M&M-I benefits from easily obtaining disjoint matchings and becomes the algorithm of choice. Unlike DSQL+, it presents no scalability issue after establishing disjoint matchings and easily finds more matchings thereafter, hence remains the best option across k values with small queries. With large queries, on which it is hard to obtain nearly disjoint matches, M&M is the algorithm of choice. Overall, our algorithms provide up to tens of thousands of representative query answers.

6.4 Evaluation on Special Query Types

Here, we evaluate five distinct query types of size 16, namely Cycle (C), Path (P), Tall Narrow (TN), Tall Wide (TW), Flat Wide (FW), and Flat Narrow (FN). We classify a query graph as *Tall Narrow* if diameter > 2 and average degree ≤ 2, *Tall Wide* if diameter > 2 and average degree > 2, *Flat Wide* if diameter ≤ 2 and average degree ≤ 2, and *Flat Narrow* if diameter ≤ 2 and average degree > 2. Figure 11 reports the average coverage for 20 queries each query type across four datasets, with a time limit of 50 seconds. Due to their similarity, we aggregate the results for C and P queries. As with results for random walk queries of size 16, our algorithms outperform baselines, showcasing robustness across query types.

7 CONCLUSION

We studied the problem of attaining high coverage in subgraph matching within a time limit, and proposed the M&M and M&M-I algorithms that exploit enhanced global and local exploration in the enumeration phase. A greedy variant returns k results with high coverage. Our proposals outperform the state of the art in the amount of unique nodes covered across datasets and query sizes. In the future, we will examine subgraph matching with diversity awareness, drawing from diversity in graph anonymization [26, 36].

REFERENCES

[1] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *Proc. ACM Manag. Data* 1, 2 (2023), 167:1–167:26. https://doi.org/10.1145/3589312

[2] Saurabh Asthana, Oliver D. King, Francis D. Gibbons, and Frederick P. Roth. 2004. Predicting protein complex membership using probabilistic network reliability. *Genome Res.* 14 (2004), 1170–1175. https://doi.org/10.1101/gr.2203804

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*. 1199–1214. https://doi.org/10.1145/2882903.2915236

[4] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*. 595–602. https://doi.org/10.1145/988672.988752

[5] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* 14, 7 (2013), 1–13. https://doi.org/10.1186/1471-2105-14-S7-S13

[6] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA*. 129–143. https://doi.org/10.1007/11790754_8

[7] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-Performance Dynamic Pattern Matching over Disordered Streams. *PVLDB* 3, 1 (2010), 220–231. https://vldb.org/pvldb/vol3/R19.pdf

[8] Guillaume Damiand, Christine Solnon, Colin de la Higuera, Jean-Christophe Janodet, and Émilie Samuel. 2011. Polynomial algorithms for subisomorphism of nD open combinatorial maps. *Comput. Vis. Image Underst.* 115, 7 (2011), 996–1010. https://doi.org/10.1016/j.cviu.2010.12.013

[9] Marina Drosou and Evaggelia Pitoura. 2012. DisC diversity: result diversification based on dissimilarity and coverage. *PVLDB* 6, 1 (2012), 13–24. https://doi.org/10.14778/2428536.2428538

[10] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified Top-k Graph Pattern Matching. *PVLDB* 6, 13 (2013), 1510–1521. http://www.vldb.org/pvldb/vol6/p1510-fan.pdf

[11] Pasquale Foggia, Gennaro Percannella, and Mario Vento. 2014. Graph Matching and Learning in Pattern Recognition in the Last 10 Years. *Int. J. Pattern Recognit. Artif. Intell.* 28, 1 (2014). https://doi.org/10.1142/S0218001414500013

[12] Fedor V. Fomin, Petr A. Golovach, Lars Jaffke, Geevarghese Philip, and Danil Sagunov. 2024. Diverse Pairs of Matchings. *Algorithmica* 86, 6 (2024), 2026–2040. https://doi.org/10.1007/s00453-024-01214-7

[13] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*. 1429–1446. https://doi.org/10.1145/3299869.3319880

[14] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*. 337–348. https://doi.org/10.1145/2463676.2465300

[15] Wook-Shin Han, Jinsoo Lee, Minh-Duc Pham, and Jeffrey Xu Yu. 2010. iGraph: A Framework for Comparisons of Disk-Based Graph Indexing Techniques. *PVLDB* 3, 1 (2010), 449–459. https://doi.org/10.14778/1920841.1920901

[16] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*. 405–418. https://doi.org/10.1145/1376616.1376660

[17] Maarten Houbraken, Sofie Demeyer, Tom Michoel, Pieter Audenaert, Didier Colle, and Mario Pickavet. 2014. The Index-Based Subgraph Matching Algorithm with General Symmetries (ISMAGS): Exploiting Symmetry for Faster Subgraph Enumeration. *PLOS ONE* 9, 5 (2014), 1–15. https://doi.org/10.1371/journal.pone.0097896

[18] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms. In *EDBT*. 25–36. https://doi.org/10.5441/002/edbt.2017.04

[19] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *SIGMOD*. 925–937. https://doi.org/10.1145/3448016.3457265

[20] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *PVLDB* 6, 2 (2012), 133–144. https://doi.org/10.14778/2535568.2448946

[21] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The dynamics of viral marketing. *ACM Trans. Web* 1, 1 (2007), 5. https://doi.org/10.1145/1232722.1232727

[22] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*. 177–187. https://doi.org/10.1145/1081870.1081893

[23] Tinghuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. 2018. A Comparative Study of Subgraph Matching Isomorphic Methods in Social Networks. *IEEE Access* 6 (2018), 66621–66631. https://doi.org/10.1109/ACCESS.2018.2875262

[24] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41. https://doi.org/10.1145/219717.219748

[25] Alastair Colin Murray and Björn Franke. 2012. Compiling for automatically generated instruction set extensions. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO*. 13–22. https://doi.org/10.1145/2259016.2259019

[26] Sadegh Nobari, Panagiotis Karras, HweeHwa Pang, and Stéphane Bressan. 2014. L-opacity: Linkage-Aware Graph Anonymization. In *EDBT*. 583–594. https://doi.org/10.5441/002/edbt.2014.52

[27] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: on Compression and Computation. *PVLDB* 11, 2 (2017), 176–188. http://www.vldb.org/pvldb/vol11/p176-qiao.pdf

[28] Hemaka A Rajapakse. 2007. Small molecule inhibitors of the XIAP protein-protein interaction. *Current topics in medicinal chemistry* 7, 10 (2007), 966–971. https://doi.org/10.2174/156802607780906816

[29] John W. Raymond, Eleanor J. Gardiner, and Peter Willett. 2002. Heuristics for Similarity Searching of Chemical Graphs Using a Maximum Common Edge Subgraph Algorithm. *J. Chem. Inf. Comput. Sci.* 42, 2 (2002), 305–316. https://doi.org/10.1021/ci010381f

[30] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *PVLDB* 8, 5 (2015), 617–628. http://www.vldb.org/pvldb/vol8/p617-ren.pdf

[31] Michele Sevegnani and Muffy Calder. 2015. Bigraphs with sharing. *Theor. Comput. Sci.* 577 (2015), 43–73. https://doi.org/10.1016/j.tcs.2015.02.011

[32] Dennis E. Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. 2002. Algorithmics and Applications of Tree and Graph Searching. In *PODS*. 39–52. https://doi.org/10.1145/543613.543620

[33] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. Pilos: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *ICDE*. 1180–1193. https://doi.org/10.1109/ICDE65448.2025.00093

[34] Tom A. B. Snijders, Philippa E. Pattison, Garry L. Robins, and Mark S. Handcock. 2006. New Specifications for Exponential Random Graph Models. *Sociol. Method.* 36, 1 (2006), 99–153. https://doi.org/10.1111/j.1467-9531.2006.00176.x

[35] Christine Solnon, Guillaume Damiand, Colin de la Higuera, and Jean-Christophe Janodet. 2015. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognit.* 48, 2 (2015), 302–316. https://doi.org/10.1016/j.patcog.2014.05.019

[36] Yi Song, Panagiotis Karras, Qian Xiao, and Stéphane Bressan. 2012. Sensitive Label Privacy Protection on Social Network Data. In *SSDBM*. 562–571. https://doi.org/10.1007/978-3-642-31235-9_37

[37] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *SIGMOD*. 1083–1098. https://doi.org/10.1145/3318464.3380581

[38] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *PVLDB* 14, 2 (2020), 176–188. http://www.vldb.org/pvldb/vol14/p176-sun.pdf

[39] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB* 5, 9 (2012), 788–799. http://vldb.org/pvldb/vol5/p788_zhaosun_vldb2012.pdf

[40] Christof Vömel, Flavio de Lorenzi, Samuel Beer, and Erwin Fuchs. 2017. The Secret Life of Keys: On the Calculation of Mechanical Lock Systems. *SIAM Rev.* 59, 2 (2017), 393–422. https://doi.org/10.1137/15M1030054

[41] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD*. 335–346. https://doi.org/10.1145/1007568.1007607

[42] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213. https://doi.org/10.1007/s10115-013-0693-z

[43] Zhengwei Yang, Ada Wai-Chee Fu, and Ruifeng Liu. 2016. Diversified Top-k Subgraph Querying in a Large Graph. In *SIGMOD*. 1167–1182. https://doi.org/10.1145/2882903.2915216

[44] Ye Yuan, Delong Ma, Zhenyu Wen, Zhiwei Zhang, and Guoren Wang. 2021. Subgraph Matching over Graph Federation. *PVLDB* 15, 3 (2021), 437–450. http://www.vldb.org/pvldb/vol15/p437-yuan.pdf

[45] Shijie Zhang, Meng Hu, and Jiong Yang. 2007. TreePi: A Novel Graph Indexing Method. In *ICDE*. 966–975. https://doi.org/10.1109/ICDE.2007.368955

[46] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *EDBT*, Vol. 360. 192–203. https://doi.org/10.1145/1516360.1516384

[47] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proc. ACM Manag. Data* 2, 1 (2024), 60:1–60:29. https://doi.org/10.1145/3639315