SUBLIME: Selecting Subgraph Matching Algorithms via Machine Learning

GENRYU KURAYA, The University of Osaka, Japan
KONSTANTINOS SKITSAS, Aarhus University, Denmark
DAVIDE MOTTIN, Aarhus University, Denmark
PANAGIOTIS KARRAS, University of Copenhagen, Denmark
DAICHI AMAGATA, The University of Osaka, Japan
YUYA SASAKI, The University of Osaka, Japan

Subgraph matching is a fundamental problem in graph analysis that seeks all instances (or *embeddings*) of a *query subgraph* within a larger *data graph*. Numerous subgraph matching algorithms have been developed for efficient processing. However, the best-performing algorithm differs across query and data graphs. A previous study proposed manually designed rule-based models for selecting the algorithm to use depending on query and data characteristics. However, this rule-based model is often ineffective even on in-distribution data, let alone on graphs having the same distribution. In this paper, we propose SUBLIME, a machine-learning-based framework that selects a subgraph matching algorithm to use for the sake of efficiency, based on the query and the data. SUBLIME learns from observations of the performance of subgraph matching algorithms on different data. It comprises two key components: a *labeling strategy*, which adds labels to experimental results, and a *featurizer*, which extracts handcrafted features from datasets and queries. Our experiments in subgraph matching and subgraph coverage problems show that SUBLIME selects high-performing algorithms and improves embeddings per second by up to 36.0% and coverage by up to 46.3% over baselines.

CCS Concepts: • **Computing methodologies** → **Machine learning algorithms**; • **Information systems** → **Query optimization**.

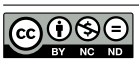
Additional Key Words and Phrases: Subgraph matching, Machine learning, Algorithm selection, Graph analysis, Subgraph coverage

ACM Reference Format:
Genryu Kuraya, Konstantinos Skitsas, Davide Mottin, Panagiotis Karras, Daichi Amagata, and Yuya Sasaki. 2026. SUBLIME: Selecting Subgraph Matching Algorithms via Machine Learning. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 238 (June 2026), 25 pages. https://doi.org/10.1145/3802115

1 Introduction

Subgraph matching seeks occurrences (i.e., *embeddings*) of a given *query graph* within a much larger *data graph*. This task has a wide variety of applications such as fraud detection [26], social network analysis [7], bioinformatics [4], and knowledge graph analysis [15, 25]. However, it is computationally challenging and known to be **NP**-hard [12]. Numerous algorithms have thus been proposed over decades to accelerate finding subgraphs [3, 4, 14, 36]. However, despite those efforts,

Authors' Contact Information: Genryu Kuraya, The University of Osaka, Japan, kuraya.genryu@ist.osaka-u.ac.jp; Konstantinos Skitsas, Aarhus University, Aarhus, Denmark, skitsas@cs.au.dk; Davide Mottin, Aarhus University, Aarhus, Denmark, davide@cs.au.dk; Panagiotis Karras, University of Copenhagen, Copenhagen, Denmark, piekarras@gmail.com; Daichi Amagata, The University of Osaka, Japan, amagata.daichi@ist.osaka-u.ac.jp; Yuya Sasaki, The University of Osaka, Japan, sasaki@ist.osaka-u.ac.jp.



ACM 2836-6573/2026/6-ART238
https://doi.org/10.1145/3802115

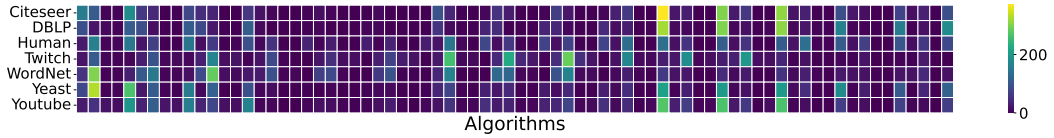


Fig. 1. Heatmap showing how often each algorithm achieves the highest EPS per dataset.

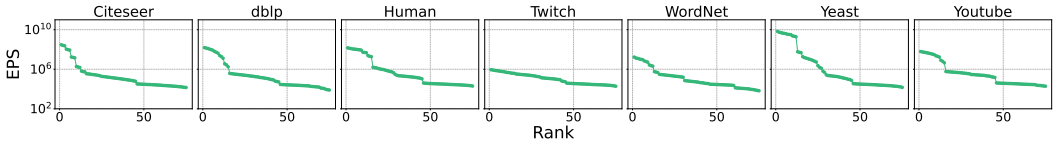


Fig. 2. Average EPS per rank; we obtain EPS at rank i by selecting the algorithm ranked i for each query.

there is no single *algorithm of choice* that performs reliably well across problem instances. Each algorithm has pros and cons depending on the query and data graphs. For instance, GQL [13] performs well on dense graphs but poorly on sparse ones, whereas RI [4] presents the opposite behavior. Previous benchmarking studies have empirically compared existing algorithms across various data graphs and concluded that there is no clear winner [19, 34, 42].

Figure 1 shows a heatmap of the number of times each algorithm achieves the best throughput in terms of *embeddings per second* (EPS) [42] over sixty algorithms on seven real-world datasets and query size in $\{4, 8, 16, 32, 64, 96, 128\}$. Figure 2 plots the average EPS for each algorithm rank on each data set. These results show that the best-performing algorithm is highly data-dependent and efficiency decreases drastically when selecting an unsuitable algorithm for given query and data.

Given this predicament, there is a need for a way to select the best subgraph matching algorithm, given a query and a data graph. Prior studies [34, 42] have proposed manual, rule-based models to select algorithms based on features of the query and data graphs according to empirical performance data. However, these models often fail to select the best algorithm even when query and data graphs remain within the training distribution, let alone when applied to out-of-distribution queries and data. There is thus a need to robustly select the algorithm to use for a given query and data.

In this paper, we propose Sublime, a novel framework that employs machine learning to automatically select and generate the subgraph matching algorithm to employ for a given problem instance, out of a realm of possibilities. We train a machine learning model on empirical performance data, which includes quadruples of a *data graph*, a *query graph*, an *algorithm*, and the associated *performance*. The key components of our framework are a *labeling strategy*, which builds training data by adding labels to empirical performance data, and a *featurizer*, which extracts well-designed features from query and data graphs. We test three labeling strategies for assigning target labels, TopX, IncY, and Weighted, defined in terms of how they select the algorithm to which they assign a positive label. The TopX strategy selects the top-X performing algorithms for each query and data graph pair. The IncY strategy selects the algorithms whose performance is within a Y ratio of the best-performing algorithms. The Weighted strategy selects algorithms based on their performance compared to the top-performing one. The featurizer extracts 13 features from query and data graphs, such as density and diameter. Put together, the labeling strategy and featurizer enable training a machine learning model on a classification task that predicts the algorithm to employ, as a *class*, for given query and data features. In the inference phase, the trained model selects the algorithm predicted to perform best on a given query and data graph, and executes subgraph matching using the selected algorithm.

In our experimental study, we evaluate 75 subgraph matching algorithms, formed by combining solutions for filtering, ordering, and enumeration, on seven real-world data graphs. Our selection

framework improves performance by up to 35.96% in EPS over baselines, and 42.75% in mean reciprocal rank (MRR) in terms of selecting the best performer. We also establish the robustness of our framework even when training on out-of-distribution (OOD) data graphs and query types. Further, we apply our framework to the related problem of subgraph coverage.

We outline our contributions as follows:

- **Novel problem:** We leverage machine learning to select the subgraph matching algorithm to employ; to our knowledge, no prior work has addressed this task.
- **Novel framework:** We propose a comprehensive framework that selects a subgraph matching algorithm via machine learning, using a labeling strategy and a feature extraction module.
- **Empirical validation:** Extensive experiments on seven real-world datasets demonstrate that our framework outperforms baselines in various scenarios.

2 Preliminaries

Here we present core definitions and related work.

2.1 Graph and subgraph matching

We define $g = (V(g), E(g), L_g)$ to be a labeled undirected graph, where $V(g)$ is a set of vertices and $E(g)$ is a set of edges; an edge $e(v, v') \in E(g)$ indicates that $v, v' \in V(g)$ are connected in g; L_g is a mapping from each vertex to the set of vertex labels Σ.

Given a query graph $q = (V(q), E(q), L_q)$ and a data graph $G = (V(G), E(G), L_G)$, an *embedding* of q in G is a mapping $M : V(q) \rightarrow V(G)$ such that: (i) M is *injective*, i.e., if $u \neq u' \in V(q)$, then $M(u) \neq M(u')$, (ii) $L_q(u) = L_G(M(u)), \forall u \in V(q)$, and (iii) $e(M(u), M(u')) \in E(G), \forall e(u, u') \in E(q)$. If there exists an embedding of q in G, then q is *subgraph isomorphic* to G. The *subgraph matching* problem calls to find *all* embeddings of query graph q in data graph G. In most cases, the data graph G is significantly larger than the query graph q.

2.2 Subgraph coverage

Similarly to subgraph matching, the problem of *subgraph coverage* aims to maximize the unique vertices appearing in embeddings found within a time limit, which we call *coverage* [30, 32].

2.3 Related Work

Subgraph Matching Algorithms. Subgraph matching algorithms feature three stages: filtering, ordering, and enumeration [34, 42]. *Filtering* narrows the search space down to the vertices and edges in the data graph that could potentially match the query graph based on structural and label information. *Ordering* determines a guiding sequence by which vertices in the query graph are efficiently mapped to the data graph. *Enumeration* outputs all embeddings of the query graph.

Numerous subgraph matching algorithms have been proposed. LDF [36] filters unnecessary vertices in data graphs based on vertex labels and degrees. NLF [43] enhances filtering by evaluating neighborhood labels. GQL [13] determines an efficient matching order based on candidate size, and LFTJ [34, 38] employs a worst-case optimal join algorithm based on hybrid (merge-based or galloping) set intersections. RI [4] uses an ordering technique that selects vertices most connected to the already ordered vertices set. DPISO [10] converts the query graph into a directed acyclic graph (DAG) for dynamic filtering and utilizes *failing sets*, i.e., unmatchable subsets of the query, to prune the search space. RM [35] treats subgraph matching as a problem on relational database tables. VEQ [14] determines a matching order by vertex groups and dynamically prunes the search space using equivalences. KSS [41] reduces the search space by decomposing the query graph into a

kernel and a *shell*, confining the main computation to the kernel. Pilos [31] adopts a strong spectral filtering technique based on Laplacian matrices and the *interlacing theorem* as algebraic constraints.

Subgraph Matching with Machine Learning. Several studies [9, 17, 18, 21, 22, 39] use machine learning for subgraph matching. NeuroMatch [22] performs binary classification to determine whether a query graph exists in the data graph. Sub-GMN [18] and AED-Net [17] identify a single embedding of the query graph in the data graph. HFrame [9] applies machine learning to compute vertex embeddings and predict matchings. QVO [39] uses reinforcement learning in the ordering step, while RSM [21] does so to determine the candidate generation order in the enumeration step. NeuSO [40] determines ordering using a graph neural network and multitask framework that estimates subquery cardinality and execution cost. It has also been proposed to use active learning for selecting query graph nodes in the filtering stage [8].

All these works use machine learning to perform subgraph matching directly. To our knowledge, no prior work uses machine learning to *choose* the most appropriate among existing algorithmic options for each step of the subgraph matching process, as we do.

Selecting Algorithms via Machine Learning. While unrelated to subgraph matching, there is a prior work, Ease [23], that selects a graph partitioning algorithm from a pool of existing methods utilizing machine learning, highlighting the importance of structural features like degree skewness; XGBoost [6] performs best for supervised machine learning on the task.

Subgraph Coverage Algorithms. Algorithms for the coverage problem use techniques that efficiently expand the set of covered unique vertices along with existing filtering and ordering methods. MatCo [30] uses a *local candidate space* data structure and effective pruning strategies, while Mix&Match [32] applies an effective enumeration method that searches a wider range of data graph.

Benchmarking. Several benchmarking studies evaluate the performance of algorithms to understand their characteristics [19, 34, 42]. Such studies have established that the best-performing algorithm depends strongly on the query and data graphs. Lee et al. [19] re-implemented existing algorithms under a common environment and evaluated each algorithm. Sun et al. [34] proposed the decomposition of algorithms into filtering, ordering, enumeration, and optimization steps and analyze the impact of each step on the performance. Further, Zhang et al. [42] generate new algorithms by recombining techniques for each stage of existing methods and show that some new recombined algorithms outperform the original ones. In this work, we study how to select recombined algorithms based on the query and data characteristics.

3 Motivation and Problem Definition

As we discussed, Figure 1 presents the top-performance distribution of seventy five algorithms in seven datasets by the throughput of identified embeddings per second (EPS) [42], when running 2 800 queries for each dataset (Section 5 provides more details). We observe that the best-performing algorithm varies significantly across datasets. For example, DBLP and WordNet present largely deviating results on the best performer. On DBLP, the right-hand-side algorithms in Figure 1 tend to perform best, whereas on WordNet the left-hand-side algorithms have an advantage. On the other hand, some data, for example, Yeast and Youtube, exhibit similar trends.

Besides, Figure 2 shows that EPS does not grow linearly vs. rank. For example, in Yeast, the EPS difference between the top-ranked and the lower-ranked algorithms reaches several orders of magnitude, whereas in Twitch that difference is more modest. In addition, EPS differs highly even among top-ranked algorithms, for instance the algorithms ranked third and fourth on Citeseer. These results suggest that it may not always be essential to single out the top-performing algorithm, yet there are cases where a lower-ranked contender can fall noticeably short of the best. In effect, predicting the most efficient algorithm given a query and a data graph is a challenging task. We

thus conjecture that it may be practical to select algorithms based on *empirical performance data* obtained from experimental results, defined as follows:

DEFINITION 1 (EMPIRICAL PERFORMANCE DATA). *Given a set $\mathcal{A}$ of algorithms, empirical performance data includes a set of quadruples <G, q, algorithm, performance measure> for any algorithm in $\mathcal{A}$, where G identifies a data graph and q a query.*

While we use EPS [42] and coverage [30, 32] as performance measures, any other measures, such as runtime for a task, would be applicable. We assume that a performance measurement is available for each data graph, query, and algorithm in $\mathcal{A}$, hence we know the best-performing algorithm for a given q on G. We define the problem we address as follows:

PROBLEM 1 (SUBGRAPH MATCHING ALGORITHM SELECTION). *Given query graph q, a data graph G, a set $\mathcal{A}$ of subgraph matching algorithms, the* subgraph matching algorithm selection *problem calls to select a single best-performing algorithm from $\mathcal{A}$ for q and G.*

A straightforward solution to the problem may select an algorithm that achieves high performance based on the empirical performance data. For example, one may select the subgraph matching algorithm that often achieves the best performance for either all data graphs or the same data graph. However, such an approach may not select the best algorithm for each query, as it does not consider the query graph. Another solution might be to cache the best-performing algorithm for each query and use the cached knowledge, as a workload may evaluate recurring or structurally similar queries. However, this solution is not practicable either, due to the vast number of possible queries. We thus propose to train a machine learning model on historical queries using structural features, to enable reliable predictions for similar, unseen queries. For instance, if training data show that algorithm X performs well on small, dense queries for dataset G, the model would infer that X likely performs well on another dense query Q with similar characteristics. This learning-based selection allows us to quickly predict an appropriate algorithm with minimal computational overhead, effectively leveraging past experience without the need for exhaustive caching.

4 Our Framework

Here we present our framework that selects a subgraph matching algorithm with the highest predicted performance among a given set of algorithms via machine learning and performs subgraph matching with the selected algorithm. Figure 3 presents an overview. We perform a classification task, in which each class label represents an algorithm, the inputs are features of the query and data graphs. The model learns to predict the best-performing algorithm given the features of the query and data graphs.

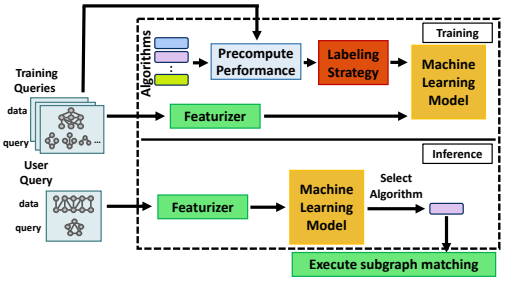


Fig. 3. SUBLIME framework.

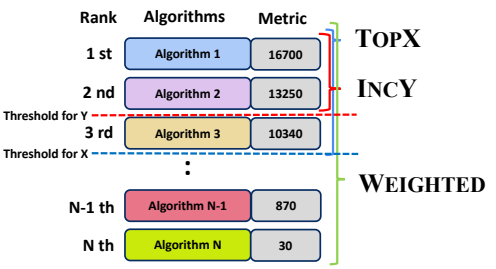


Fig. 4. Labeling strategies.

In the training phase, we extract features from the query and data graphs and assign labels corresponding to algorithm performance according to our labeling strategies. In the inference phase,

we extract features from the given query and data graphs and input them into the trained model. The model infers an algorithm expected to achieve the highest performance, so our framework executes subgraph matching using the selected algorithm.

4.1 Labeling Strategies

To render empirical performance data usable for model training, we assign labels to them. When labeling algorithms for a given query, we assign labels to multiple algorithms with high performance instead of only selecting the single algorithm with the highest performance, based on our empirical analysis in Section 3. Mathematically, we aim to learn a performance function $f: G \times q \rightarrow \mathbb{R}^{|\mathcal{A}|}$. Thus, the size of the training data is the number of queries.

We propose three labeling strategies as follows:

- TopX: Assign positive labels to the top-X performing algorithms.
- IncY: Assign positive labels to algorithms performing at least a ratio Y of the best performance.
- Weighted: Assign real-valued positive labels to algorithms based on weights computed as their relative performance.

Figure 4 illustrates how our labeling strategies work with an example. By the TopX strategy with $X = 3$, we assign label 1 to the top-3 algorithms and 0 to others. By the IncY strategy with $Y = 0.7$, we assign label 1 to the algorithms whose EPS are higher than $0.7 \cdot 16,700 = 11,690$ and 0 to others. By the Weighted strategy, we assign labels 1.0, 0.79, 0.62, 0.05, and 0.002 to the 1st, 2nd, 3rd, $(N-1)$th, and Nth algorithms, respectively.

4.2 Featurizer

Featurizer extracts 13 structural features from query and data graphs that represent their characteristics. These features include the number of vertices in the query and data graphs (i.e., $|V(q)|$ and $|V(G)|$), the number of edges in the query graph (i.e., $|E(q)|$), the *average degree* of the query graph, the *diameter* of the query graph, *degeneracy* (i.e., maximum core number [29]) of query and data graphs, *density* of query and data graphs (calculated as $\frac{2|E(q)|}{|V(q)|(|V(q)|-1)}$ and $\frac{2|E(G)|}{|V(G)|(|V(G)|-1)}$, respectively), *tree width* [27], *label set size* (i.e., $|\Sigma|$), and *candidate size* of the data graph, and the *label ratio* between query and data graphs. We define the last two measures, which are not standard measures in graph analysis.

Candidate size is the average number of candidate vertices involved in subgraph matching, as computed by the LDF filtering algorithm [36]: $\frac{1}{|V(q)|} \sum_{u \in V(q)} |C(u)|$ where $|C(u)|$ is the number of candidates vertices in G for query vertex u. We calculate candidate size using LDF, a lightweight algorithm based on label and degree constraints. *Label ratio*, which we propose, is the dot product of the label distributions in the query graph and the data graph: $\frac{\sum_{\ell \in \Sigma} |V(q)_\ell| \cdot |V(G)_\ell|}{|V(q)| \cdot |V(G)|}$, where $V(q)_\ell$ and $V(G)_\ell$ denote the sets of vertices with label ℓ in q and G, respectively.

Four of these features, namely the density, label set size of the data graph, candidate size, and tree width, appear in the rules proposed by prior work [42]. We add the nine extra features as sufficient for a thorough graph description for our purposes. We tried to add even more elaborate features, yet they did not bring any benefit to the featurizer.

4.3 Model Training

Here we explain how we train the machine learning model. We use the standard cross-entropy loss [28] as follows:

$$\mathcal{L} = -\frac{1}{|Q|} \sum_{q \in Q} \sum_{i=1}^{|\mathcal{A}|} y_{q,i} \log \frac{\exp(\hat{y}_{q,i})}{\sum_{j=1}^{|\mathcal{A}|} \exp(\hat{y}_{q,j})} \tag{1}$$

where $y_{q,c}$ is the ground-truth label and $\hat{y}_{q,c}$ is the output by the machine learning model for class $c \in [1, |\mathcal{A}|]$ in query q. Q denotes the set of queries in training and $|Q|$ denotes their number.

We update the model's parameters using backpropagation targeting this loss function, for a predefined number of epochs. After each epoch, we evaluate the model's performance on a separate validation set to monitor its generalization capability. To avoid overfitting, we employ an early stopping strategy: if the monitored loss on the validation set does not decrease over a certain number of epochs, we terminate the training. We select the model that achieved the best validation loss during training as the final model.

5 EXPERIMENTS

We conduct experiments to evaluate where Sublime achieves good performance in subgraph matching and coverage, is robust to out-of-distribution queries and datasets and to assess the cost of training and preparation, the importance of different features.

5.1 Setup

Algorithms and experimental environment. For our experiments on subgraph matching, we consider the methods listed in Table 1 for each stage and produce 75 algorithms as their $5 \cdot 3 \cdot 5$ combinations. Pilos [31] is a recently proposed filtering method, while other algorithms have been recommended in a prior benchmarking study [42]. We use the appellation 'LFTJ' to refer to the backtracking-based method that calculates local candidates via hybrid set intersection [38, 42]. We use QFilter [11] for examining intersection mechanisms. Still, the core distinction between enumeration algorithms lies in the use of the candidate space, the backtracking order, and the utilization of local candidates. We use a publicly available source code[1] [42] and share[2] our own code. Note that the enumeration method of RM does not work for some queries, so we remove the RM option from the enumeration phase.

Table 1. Subgraph matching: options for each stage.

Stages	Methods
Filtering	LDF [36], NLF [43], DPiso [10], VEQ [14], Pilos [31]
Ordering	RI [4], RM [35], GQL [13]
Enumeration	EXPLORE [36], LFTJ [38], QFilter [11], KSS [41], VEQ [14]

For subgraph coverage, we consider the methods listed in Table 2. To filter a vertex, MatCo [30] checks whether the vertex label in the data graph match that in the query graph, and orders vertices by their degrees. We call MatCo's native filtering technique LAB and its ordering technique DEG. We implement these MatCo techniques in the framework we use with Mix&Match, as well as the core MatCo algorithm for enumeration. We also consider VEQ [14] for filtering and CFL [3] for ordering in subgraph coverage, as these are used in Mix&Match [32]. We thus produce 12 algorithms as $2 \cdot 2 \cdot 3$ combinations.

Datasets. We use the seven data graphs presented in Table 3; the same datasets have been widely used in previous studies (e.g., [34, 42]). Since Twitch originally has no labels, we assign labels uniformly at random following existing works [35].

[1] https://github.com/GenryuK/Sublime-Selecting-Subgraph-Matching-Algorithms-via-Machine-Learning.git
[2] https://anonymous.4open.science/r/Selecting-Subgraph-Matching-Algorithm-via-Machine-Learning-7C86

Table 2. Subgraph coverage: options for each stage.

Stages	Methods
Filtering	LAB [30], VEQ [14]
Ordering	DEG [30], CFL [3]
Enumeration	MatCo [30], Mix&Match [32], Mix&Match-I [32]

Table 3. Data statistics; k: degeneracy (max core number).

Dataset	Name	$\|V\|$	$\|E\|$	k	$\|\Sigma\|$	Type
Citeseer[2]	cs	3,264	4,536	7	6	Citation
DBLP[3]	db	317,080	1,049,866	113	14	Collab
Human[3]	hm	4,674	86,282	148	43	Protein
Twitch[4]	tw	168,114	6,797,557	149	45	Social
WordNet[3]	wn	76,853	120,399	31	4	Lexical
Yeast[3]	ys	3,112	12,519	10	70	Protein
Youtube[3]	yt	1,134,890	2,987,624	51	23	Social

Query sets. We generate two types of query sets; induced graphs Q_I and star graphs Q_S. To generate a Q_I query graph, we start from one vertex in the data graph, extend edges to its connected neighbors, and then move to one of those neighbor vertices and repeat this process until we reach the desired number of vertices. Thereafter, we extract all edges that connect the extracted vertices. To generate a Q_S query graph, we start from one vertex in the data graph, extending edges to its connected neighbors, and then move to a neighbor vertex that *has not been extracted* and repeat this process until we reach the desired number of vertices. We extract queries of seven different sizes, where size $i \in \{4, 8, 16, 32, 64, 96, 128\}$ is the number of vertices; we indicate as Q_i the set of queries of size i.

We prepare 200 queries for each of Q_{I_i} and Q_{S_i} from each dataset (i.e., 2, 800 across all datasets), yield a total of 19, 600 (i.e., 2,800 × 7) query-data pairs. We remove queries for which all algorithms fail to find any embeddings (i.e., totally 19, 151 queries). In the experiments on subgraph coverage, we examine query size $i \in \{16, 32, 48, 64, 80, 96, 112, 128\}$. We omit smaller sizes, for which the problem is easy. To ensure the reliability of our results and mitigate the effects of random initialization, we conducted all experiments using five different seeds and report average performance values.

All algorithms are implemented in C++, whereas we developed all machine learning components in Python3. We compiled the C++ code using CMake version 3.22.1 and g++ version 11.4.0. All experiments are conducted on a server with two Intel Xeon E5-2640 v4 @ 2.40GHz CPUs (totaling 20 physical cores and 40 logical cores) and 1 TB of RAM running Ubuntu 22.04 LTS.

Collecting empirical performance data. To collect empirical performance data, we run subgraph matching on all queries using all algorithms in advance with a 10-second time limit for enumeration. We then use a part of the collected data for training and other parts for validation and testing, with a random 6 : 2 : 2 split for training, validation, and testing on each data graph. As we collect such data once only, we exclude the collection time from the reported results. Figure 5 shows the cumulative cost of subgraph matching for each algorithm and the total embeddings each algorithm finds within the preprocessing time and the 10-second time limit for enumeration.

ML model and hyperparameters. We use a 2-layer multilayer perceptron (MLP) as our machine learning model, valued for its simplicity and rapid training capability, and train it with the Adam

[2] https://networkrepository.com/citeseer.php
[3] https://github.com/RapidsAtHKUST/SubgraphMatching
[4] https://snap.stanford.edu/data/twitch_gamers.html

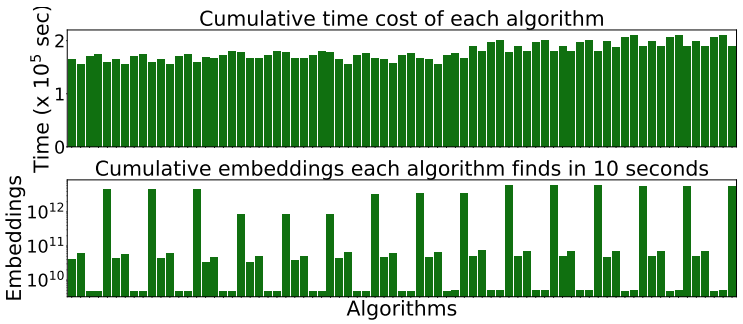


Fig. 5. Cumulative runtime, # embeddings, all queries.

optimizer [16] using ReLU as the activation function in the hidden layers and softmax in the output layer. We set the maximum number of training epochs to 10 000, and apply early stopping if the validation loss does not improve within 20 consecutive epochs. We tune hyperparameters in [5e-4, 1e-3, 5e-3, 1e-2, 5e-2] for learning rate, [0.0, 0.1, 0.2, 0.3, 0.4, 0.5] for drop out rate, [0.0, 1e-5, 1e-4, 1e-3, 1e-2] for weight decay, and [16, 32, 64, 128] for unit size in hidden layer [2, 24, 33]. Each labeling strategy has its own unique hyper-parameters: for TopX, we choose the value of X from the set [1, 2, 3, 4, 5, 6, 7]; for IncY, we choose the value of Y is chosen from [1.0, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6, 0.55, 0.5]. We use Optuna [1] to find the hyperparameters minimizing the validation loss.

For each experiment, we perform hyperparameter search on the training data and select the combination that yields the minimum loss on the validation set from the search ranges specified above.

Baselines. We compare our methods against the following:

- **VOTED-D**, which selects the algorithm that most frequently performs the best on a given dataset.
- **VOTED**, which selects the algorithm that most frequently performs the best across all datasets.
- **RECALGO** [42], which selects methods for filtering, ordering, and enumeration by a manually designed rule-based model based on the characteristics of the graph.

Measures. We report the average EPS (embedding per second) [42] for the subgraph matching problem. This metric indicates how many embeddings are processed per second within a specified timeout, as opposed to measuring the time required to find a fixed number of embeddings. Furthermore, since the order of exploration is an intrinsic algorithmic decision, we do not enforce that the embeddings discovered within the timeout be identical across methods. The runtime includes filtering, ordering, and enumeration, and, in addition, for our methods and RECALGO, the time to extract features from query graphs and inference time. We set the timeout of enumeration to 10 seconds. For the coverage problem, we report average coverage, i.e., the number of unique vertices in embeddings found within the time limit. Since the EPS and coverage values fluctuate across queries, we normalize each measured value through division by the maximum observed value for each query and average across queries. We also report the average MRR (mean reciprocal rank) among test query sets over five random splits, which evaluates how close a method comes to selecting the true best-performing algorithms, calculated as $MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}$, where $rank_q$ denotes the true rank of the algorithm that a method selects for query q. If we select the best algorithm for each query, we get MRR 1. For our methods, the labels are calculated based on EPS for subgraph matching (Secs. 5.2–5.9) and coverage for subgraph coverage (Sec. 5.11.)

We also use another common measure; the time to find k embeddings, denoted as *time-to-k* (see Section 5.10). This measure can reflect search behavior when a limited number of results is required, yet it favors algorithms that discover easy embeddings early, is highly sensitive to the distribution of matches, and may be dominated by timeouts for queries with few matches. Moreover, it is difficult to select an appropriate value of k, which depends on dataset characteristics that are often unknown in advance. In contrast, EPS is a throughput-oriented metric that captures steady-state enumeration performance without relying on an arbitrary cutoff. Nevertheless, EPS can be biased by easy queries or short time limits. To mitigate these effects, we normalize performance across queries and adopt longer timeouts than prior work, so that EPS reflects sustained rather than short-term enumeration performance.

Table 4. MRR across datasets.

MRR	cs	db	hm	tw	wn	ys	yt
VOTED-D	0.2793	0.2912	0.1565	0.2540	0.2183	0.2269	0.2461
VOTED	0.2793	0.3004	0.1528	0.0545	0.0717	0.2111	0.2461
RECALGO	0.0812	0.0644	0.0970	0.1183	0.0929	0.0675	0.0614
Ours TOPX	0.3396	**0.3570**	**0.2234**	**0.2765**	0.3045	0.2929	**0.3043**
Ours INCY	**0.3415**	0.3527	0.2189	0.2734	**0.3099**	**0.2961**	0.2933
Ours WEIGHTED	0.3191	0.3442	0.2106	0.2520	0.2973	0.2886	0.2889

Table 5. EPS across datasets.

EPS	cs	db	hm	tw	wn	ys	yt
VOTED-D	**0.6066**	0.5566	0.4015	0.6273	0.4838	0.4207	0.5432
VOTED	**0.6066**	0.5687	0.4996	0.1946	0.1353	**0.5364**	0.5432
RECALGO	0.2656	0.1866	0.3490	0.2852	0.3034	0.2012	0.1947
Ours TOPX	0.5611	**0.6405**	0.5656	0.6818	0.6376	0.3399	**0.6667**
Ours INCY	0.5196	0.6402	0.5616	0.6774	0.6360	0.3324	0.6566
Ours WEIGHTED	0.5665	0.6323	**0.5853**	**0.6887**	**0.6578**	0.4370	0.6659

5.2 Comparison with baselines

We compare the performance of our methods vs. baselines. Tables 4 and 5 show the average MRR and average normalized EPS for each method, respectively. Our methods outperform baselines on all datasets in MRR, achieving an MRR of 0.2945 on average across all datasets, which indicates they select about the third-best algorithm on average, while baselines achieve an MRR of 0.2134 on average. Moreover, our methods achieve higher EPS than baselines on datasets where baselines falter. On hm, our methods perform strongly while VOTED-D, which selects the algorithm that most frequently performs the best, faces a difficulty as the algorithms that achieve high EPS are widely diversified (see Figure 1). On tw, our methods still perform well, while VOTED does not, as the algorithms that achieve high EPS on this data set differ markedly from those with other datasets. Further, RECALGO underperforms across all datasets because it does not consider PILOS [31] in its options, which achieves the best EPS on several queries.

Table 6. Average runtime.

Datasets	cs	db	hm	tw	wn	ys	yt
Running time	4.52	7.96	8.14	9.61	11.40	2.98	9.42

On EPS, our methods outperform baselines on five datasets (i.e., db, hm, tw, wn, and yt). In particular, on wn, WEIGHTED achieves about 1.4 times higher EPS than VOTED-D, which achieves the highest performance among baselines, despite the inference cost overhead. On cs and ys, our methods fail to achieve higher EPS than baselines even while they achieve higher MRR score, a

result we attribute to inference time cost. Table 6 shows the average runtime including filtering, ordering, and enumeration time for each dataset. We observe that the runtime on cs and ys is lower than on other data, rendering inference time an overhead; as ys enables the highest average EPS values among all datasets, it magnifies performance differences among algorithms (see Figure 2). If we exclude the inference cost, the algorithms SUBLIME selects achieve higher EPS than baselines on all datasets, as Table 7 shows.

Table 7. EPS excluding inference time cost.

EPS	cs	db	hm	tw	wn	ys	yt
Ours TOPX	0.6863	0.7020	0.6221	0.7374	0.6968	0.5640	0.7336
Ours INCY	0.6849	0.6703	0.5999	0.6885	0.6477	0.5607	0.6693
Ours WEIGHTED	0.6778	0.6625	0.6203	0.6995	0.6696	0.6279	0.6789

Among our methods, TOPX trumps INCY and WEIGHTED on MRR, while WEIGHTED outpaces others on EPS. That is so because WEIGHTED distributes weight to all algorithms that achieve high EPS value, whereas TOPX strictly distributes weight to a few methods that achieve high EPS value. Since WEIGHTED aims at high EPS, it outperforms TOPX on datasets that exhibit large gaps in EPS between algorithms, such as ys (see Figure 2).

We also evaluate performance using only one dataset for training and testing. Since the training data contains results for a single dataset, VOTED is equivalent to VOTED-D. Thus, we only report the results for VOTED-D. Tables 8 and 9 show the average MRR and average normalized EPS, respectively. SUBLIME retains its performance and outperforms baselines with minor differences from the results in Tables 4 and 5.

Table 8. MRR using only one dataset.

MRR	cs	db	hm	tw	wn	ys	yt
VOTED-D	0.2793	0.2912	0.1565	0.2540	0.2183	0.2269	0.2461
RECALGO	0.0812	0.0644	0.0970	0.1183	0.0929	0.0675	0.0614
Ours TOPX	**0.3428**	**0.3592**	**0.2250**	**0.2734**	**0.3155**	**0.3106**	**0.3011**
Ours INCY	0.3423	0.3500	0.2166	0.2652	0.3131	0.2991	0.3003
Ours WEIGHTED	0.3218	0.3535	0.2187	0.2547	0.3014	0.3020	0.2907

Table 9. EPS using only one dataset.

EPS	cs	db	hm	tw	wn	ys	yt
VOTED-D	**0.6066**	0.5566	0.4015	0.6273	0.4838	0.4207	0.5432
RECALGO	0.2656	0.1866	0.3490	0.2852	0.3034	0.2012	0.1947
Ours TOPX	0.5645	**0.6386**	0.5738	**0.6862**	0.6496	0.3487	0.6501
Ours INCY	0.5546	0.6336	0.5778	0.6726	0.6495	0.3331	0.6583
Ours WEIGHTED	0.5712	0.6311	**0.5998**	0.6857	**0.6599**	**0.4365**	**0.6701**

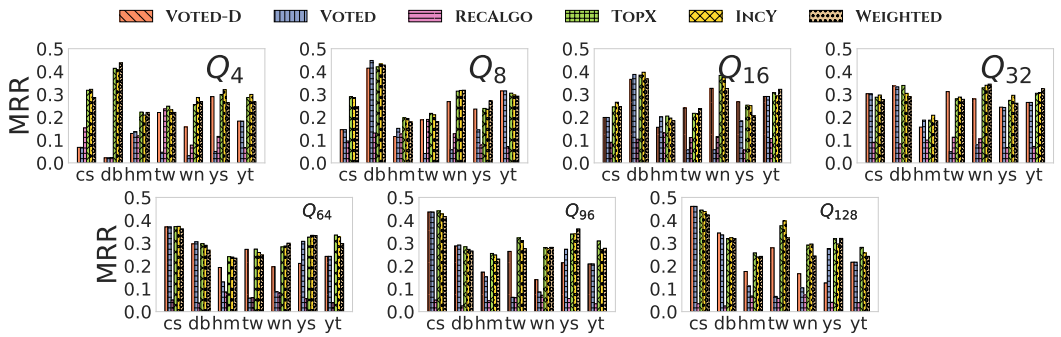
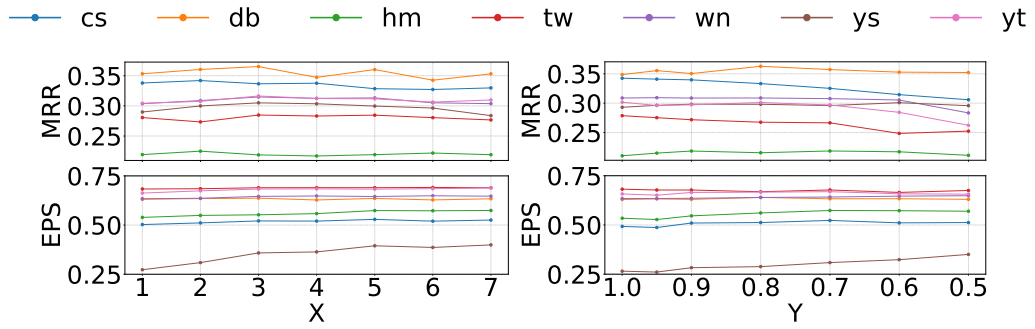


Fig. 6. MRR vs. query size.

Figure 6 shows the average MRR for each query size per dataset. Our methods outperform baselines on more than five datasets across query sizes, and on all datasets with small queries (e.g., Q_4). Besides, they maintain MRR score around 0.3 across a wide range of query sizes, while baselines are sensitive to query size. For instance, on cs, baselines falter with small query size.

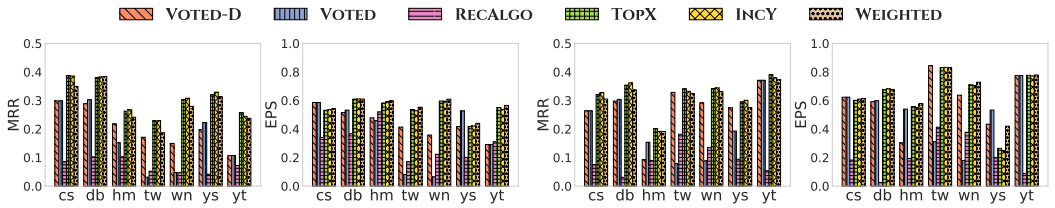
Figure 7 plots the average MRR and normalized EPS vs. hyperparameter values (i.e., X and Y) for TopX and IncY. Recall that the number of true labels increases with X and falls with Y. In most datasets, TopX and IncY are largely unaffected by hyperparameters. However, in ys, while MRR is fairly stable, EPS is highly sensitive to hyperparameters. As ys presents a large drop in EPS across algorithms (see Figures 1 and 2), labeling a small share of those algorithms as TRUE causes fluctuations in ranking and EPS. On the other hand, in cs, tw, wn, and yt, small Y values in IncY cause MRR to fall, indicating that the top-4 algorithms achieve similar EPS, given their MRR in the range [0.25, 0.35].



(a) TopX (b) IncY

Fig. 7. MRR and EPS vs. hyperparameters, TopX and IncY.

Figure 8 plots the average MRR and average normalized EPS for each query type. Sublime methods outperform baselines on all datasets on MRR. On EPS, Sublime outperforms baselines on five datasets (i.e., db, hm, tw, wn, and yt) when testing on Q_I and on four datasets (i.e., db, hm, wn, and yt) when testing on Q_S. This difference is attributed to the difficulty in processing each query set. Since queries in Q_S have a simple tree structure, running time is shorter (average running time 9.1433 for Q_I and 8.9471 for Q_S), accentuating the inference time overhead. Sublime methods incorporate such cost, hence their EPS on tw for Q_S is close to that of baselines, within 1% of the maximum value. Thus, Sublime methods fare well in this case too.



(a) Q_I (b) Q_S

Fig. 8. MRR and EPS for each query type.

Overall, Sublime methods achieve better MRR than baselines, as they almost always make better choices, attain higher EPS on many datasets, and excel with more costly queries. Besides, they afford good performance with all query sizes, and even when queries are inclined towards one particular type.

5.3 Evaluation on OOD datasets

For the sake of completeness, we also evaluate the performance of our methods on out-of-distribution (OOD) datasets to assess their power to generalize beyond training data. We train and validate on queries drawn from six out of the seven datasets, and test on queries from the seventh dataset. We omit VOTED-D, as our training data by design excludes results for the test datasets. Results of VOTED differ from those in Table 5, since the training sets are now different from those in previous experiments. Figure 9 presents the measured average MRR and normalized EPS of each method. On six datasets (i.e., cs, db, hm, tw, ys, and yt), our methods fail to outperform baselines on MRR. While baselines keep their performance on par with that in Tables 4 and 5. That result is explicable by the fact that dataset features play a more significant role, as we also discuss later in Section 5.8, hence it is challenging for SUBLIME methods to select an algorithm on OOD datasets. Nevertheless, SUBLIME outperforms baselines on wn, for which baseline MRR values in Table 4 are low, due to the divergence of algorithms that achieve high EPS in relation to other datasets (see Figure 1). On the tw dataset RECALGO fares better than other methods on MRR because NLF, its usual filtering choice, performs well on tw, as Figure 1 shows. Overall, these results indicate that SUBLIME methods, and baselines too, face a challenge on OOD datasets.

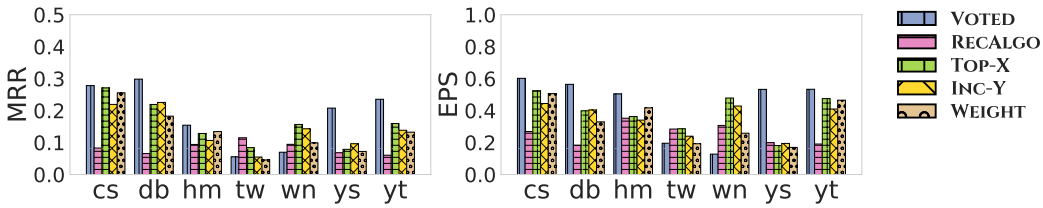


Fig. 9. MRR and EPS on OOD data.

While OOD datasets pose a challenge, the target data is typically available for training in advance in settings where the training and test graph distributions diverge substantially, such as social networks (tw, cs) and biological databases (ys, hm). SUBLIME remains highly effective in such scenarios where, as evidences by the results in Tables 8 and 9. Enhancing its generalizability to entirely unseen datasets is a promising avenue for future work. Section 5.12 outlines preliminary steps toward addressing the OOD challenge, including few-shot fine-tuning using a limited number of target-domain queries.

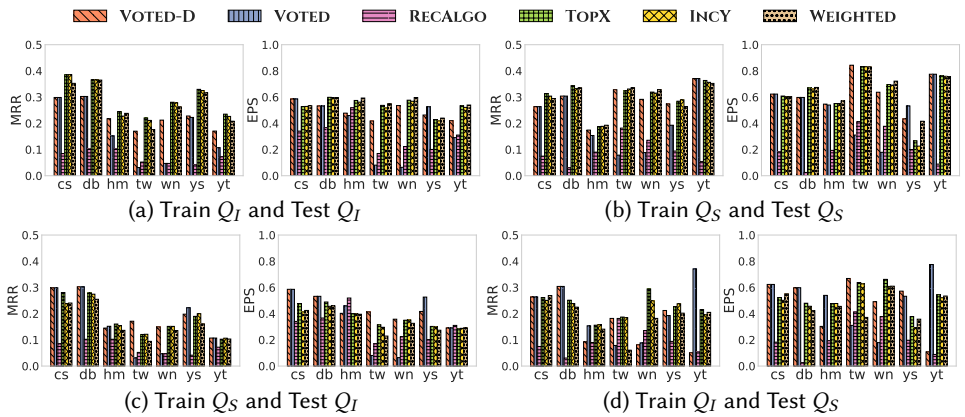
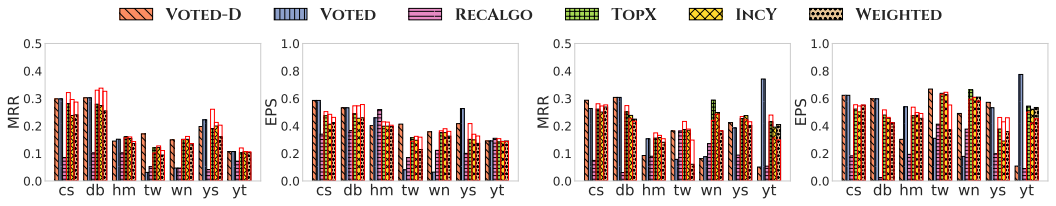


Fig. 10. Performance vs. train and test query patterns.

5.4 Effect of training query set

Next, we evaluate the effect of the training query set. Figure 10 shows the average MRR and normalized EPS for different patterns of training and test query sets for induced subgraphs Q_I and star graphs Q_S. Figures 10(a) and (b) present results for consistent training regimes, whereas Figures 10(c) and (d) show the results for inconsistent (i.e., OOD) training regimes. In consistent regimes, SUBLIME methods outperform baselines on MRR on almost all datasets except for yt. On EPS with Q_I queries, our methods still outperform baselines on five datasets (i.e., db, hm, tw, wn, and yt). On the other hand, with Q_S queries our methods outperform baselines on three datasets (i.e., db, hm, and wn); with these queries, it is hard to outperform baselines as the inference cost becomes an overhead and there is less feature information as tree width is always 1 with Q_S. The inconsistent regime is more challenging, with SUBLIME methods presenting a drop of around 0.1 in MRR score after switching the training set from Q_I to Q_S or vice versa, while baselines maintain similar scores. Training on queries that are structurally similar to those in the test set is therefore critical, and also realistic when past query workloads are available.



(a) Train Q_S + Q_α and Test Q_I (b) Train Q_I + Q_α and Test Q_S

Fig. 11. Performance with additional training set Q_α; red bars indicate the effect of adding Q_α to training.

We further investigate whether incorporating diverse query types (denoted as Q_α) into the training set improves performance, adding queries of six types to the training set: Cycle, Path, Tall Narrow (diameter $>$ 2 and average degree $\leq$ 2), Tall Wide (diameter $>$ 2 and average degree $>$ 2), Flat Narrow (diameter $\leq$ 2 and average degree $\leq$ 2), and Flat Wide (diameter $\leq$ 2 and average degree $>$ 2) [31]. For each dataset, we generate 10 query graphs consisting of 16 vertices for each query type; we use 7 and 3 query graphs for the training and validation sets, respectively. Figure 11 shows the improvement of average MRR and normalized EPS. When testing on Q_I, improvements appear across all datasets. In particular, for cs, db, and ys, SUBLIME improves on both metrics. When testing on Q_S, performance improves on cs, db, hm, tw, ys, and tw, while it degrades on wn and yt, yet never becomes inferior to that of baselines where SUBLIME outperforms them before adding training queries. The cases where SUBLIME variants outperform baselines only grow. Overall, incorporating diverse query graphs into the training set can improve performance in OOD queries.

5.5 Effect of OOD query size

We evaluate robustness to OOD query sizes. We remove queries of a chosen size from the training data and then evaluate on the removed queries. We test on queries of size 4, 8, 16, 64, 96, and 128. Figure 12 presents the average MRR and normalized EPS results. On MRR, SUBLIME methods outperform baselines on more than five datasets with larger OOD query sizes (i.e., Q_{64}, Q_{96}, Q_{128}) and on more than four datasets with smaller OOD query sizes (i.e., Q_4, Q_8, Q_{16}). With such smaller OOD queries, VOTED-D and VOTED tend to perform worse than with larger OOD queries. Smaller queries are often easier to process, hence lightweight algorithms (e.g., using NLF for filtering) tend to achieve higher EPS. In effect, the algorithms that frequently achieve higher EPS often differ from those for larger queries. On EPS, our methods also perform better than baselines, especially on larger query sizes, as the relative inference time overhead shrinks as the runtime grows.

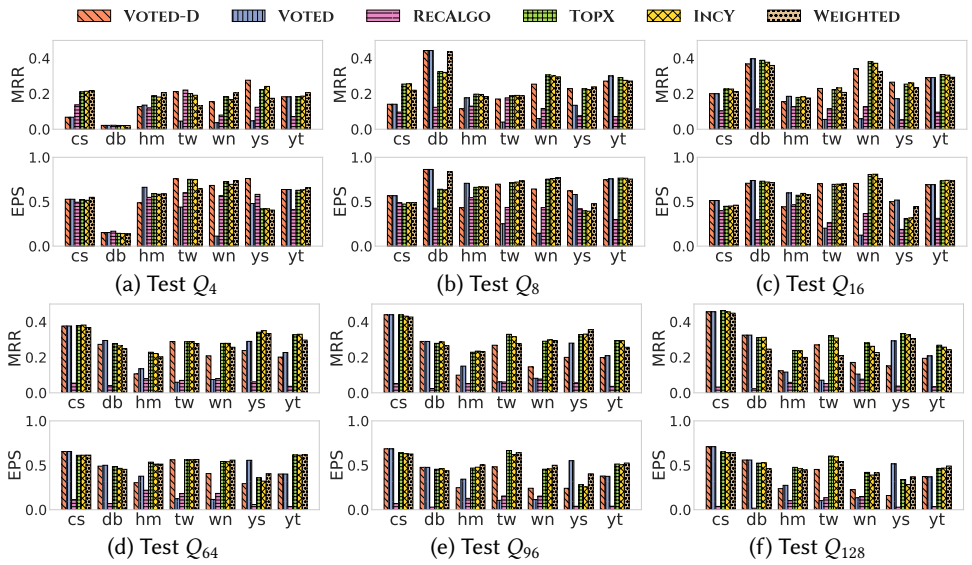


Fig. 12. MRR and EPS on OOD query sizes when removing a query size from training.

Overall, SUBLIME methods are robust to OOD query sizes. On the other hand, as we have seen, when the test set only contains queries from OOD datasets or with OOD structures, SUBLIME methods are challenged in choosing the best algorithms.

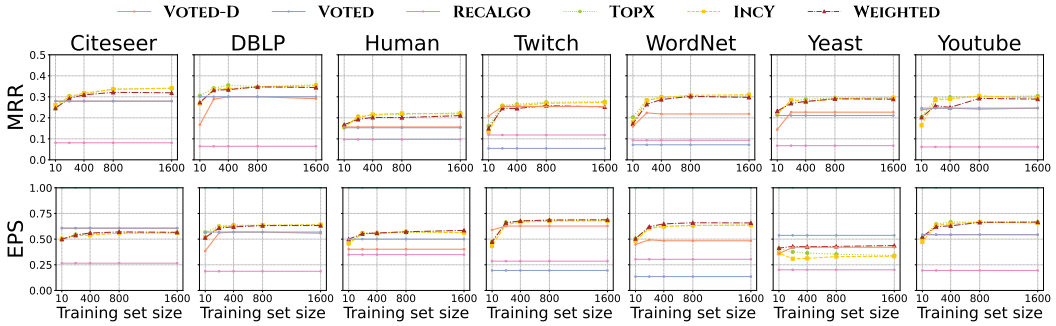


Fig. 13. Effect of training data size on MRR and EPS.

5.6 Effect of training data size

We now examine the performance of SUBLIME as the training data size decreases. By default, the training set contains about 1600 queries per dataset. We reduce this number to 800, 400, 200 and 10, while keeping the same test data. Figure 13 plots the average MRR and normalized EPS vs. training data size. The values of VOTED-D and VOTED also change depending on training set size because they decide an algorithm based on training set, while RECALGO does not change as it decides algorithm by pre-defined rules.

On MRR, the performance of SUBLIME does not change significantly even as we reduce the training set size to 200, indicating that we could potentially reduce the cost of collecting training data; we revisit this matter in Section 5.9. However, when we reduce the training set size to 10, performance degrades across all datasets, yet our methods still outperform the baselines on three

datasets (i.e., hm, wn, and ys). On cs and db, the difference from baselines is at most only 0.07%. EPS presents a similar effect of training set size. On ys, the EPS values of TopX and IncY increase as we reduce the training set size to 10 even though the MRR score decreases. This is because increasing the training set makes it richer for our methods to learn and this leads our methods to select higher rank algorithms for each dataset. As the results in Table 5 show, algorithms that often yield high EPS on ys do not necessarily yield high average EPS. In effect, Voted-D performs worse than Voted on ys on EPS and an increased training set leads to higher MRR but lower EPS. Voted outperforms Voted-D on the hm dataset too. However, in that case, the algorithms that often yield high EPS are distributed more widely than on ys (see Figure 1). Therefore, changing the training set size on hm does not incur as dramatic consequences as on ys. Sublime performs well even with a limited training set size.

5.7 Effect of ML model

We evaluate three distinct machine learning models aside of the MLP: Decision tree [5], XGBoost [6], and Transformer [37]. Tables 10 and 11 present the average MRR and normalized EPS for each model. On MRR, MLP outperforms other ML models on cs, db, tw, and yt, while XGBoost and Transfomer perform well on hm, wn, and ys. On EPS, MLP and Decision Tree work better than XGBoost and Transformer because the inference times of the latter two incur overhead. Overall, MLP achieves the best performance in terms of balancing prediction accuracy and inference time.

Table 10. MRR across ML models.

	MRR	cs	db	hm	tw	wn	ys	yt
MLP	Ours TopX	0.3384	**0.3659**	0.2226	**0.2870**	0.3109	0.2923	**0.3270**
	Ours IncY	**0.3415**	0.3527	0.2189	0.2734	0.3099	0.2961	0.2933
	Ours Weighted	0.3191	0.3442	0.2106	0.2520	0.2973	0.2886	0.2889
Decision Tree	Ours TopX	0.3090	0.3605	0.2053	0.2658	0.3049	0.2692	0.3034
	Ours IncY	0.3057	0.3501	0.2062	0.2466	0.2880	0.2714	0.2733
	Ours Weighted	0.2956	0.3396	0.2125	0.2550	0.3063	0.2562	0.2800
XGBoost	Ours TopX	0.3388	0.3646	**0.2257**	0.2686	0.3261	0.2895	0.3083
	Ours IncY	0.3085	0.3451	0.2170	0.2418	0.2775	0.2910	0.2664
	Ours Weighted	0.3157	0.3410	0.2212	0.2644	0.3090	0.2893	0.2972
Transformer	Ours TopX	0.3289	0.3609	0.2244	0.2693	**0.3279**	**0.3084**	0.3000
	Ours IncY	0.3175	0.3574	0.2186	0.2707	0.3122	0.2890	0.3051
	Ours Weighted	0.3152	0.3451	0.2147	0.2670	0.3045	0.2970	0.2940

Table 11. EPS across ML models.

	EPS	cs	db	hm	tw	wn	ys	yt
MLP	Ours TopX	0.5201	**0.6715**	0.5681	**0.7275**	**0.6844**	0.3028	**0.7201**
	Ours IncY	0.5181	0.6363	0.5442	0.6770	0.6349	0.2884	0.6561
	Ours Weighted	0.5254	0.6285	0.5684	0.6884	0.6569	0.3930	0.6654
Decision Tree	Ours TopX	0.5201	0.6429	0.5627	0.6830	0.6374	0.4349	0.6774
	Ours IncY	0.5285	0.6302	**0.5771**	0.6669	0.6284	0.2744	0.6439
	Ours Weighted	**0.5335**	0.5982	0.5642	0.6901	0.6391	**0.4490**	0.6407
XGBoost	Ours TopX	0.4369	0.6035	0.5538	0.6828	0.6643	0.3396	0.6718
	Ours IncY	0.4217	0.5857	0.5306	0.6485	0.6136	0.2632	0.6286
	Ours Weighted	0.4374	0.5838	0.5446	0.6833	0.6541	0.3297	0.6574
Transformer	Ours TopX	0.2633	0.4031	0.3973	0.5433	0.5310	0.1707	0.5156
	Ours IncY	0.2549	0.4024	0.3932	0.5327	0.5209	0.1118	0.5097
	Ours Weighted	0.2716	0.4085	0.4166	0.5677	0.5436	0.1748	0.5330

5.8 Effect of features

We reduce the number of features used in training from 13 to 4, as in RECALGO, to assess the effect of individual features. Figure 14 shows the performance gain of all-feature vs. 4-feature versions, calculated as $(MRR_{all} - MRR_{rec})/MRR_{rec}$, where MRR_{rec} is the MRR of the 4-feature version and MRR_{all} that of the 13-feature version. On almost all datasets except for tw, performance improves when using all features. On tw, the performance of TOPX and INCY improves while that of WEIGHTED degrades slightly. These results suggest that the features used in RECALGO are insufficient and it is hard to manually build rule-based models. There is however room for improving performance by feature selection.

Fig. 14. Effect of features.

Figure 15 shows Shapley values, which represent each feature's contribution to model predictions. As establishing baseline values for calculating Shapley value is computationally expensive, we selected 100 background samples from our training set of approximately 10,000 queries and completed the analysis in a few hours. We find that *label ratio*, a feature of our own design, has the highest impact, indicating its importance for data and query graphs. Besides, the features of the data graph (e.g., core number, vertices, label set size, and density) tend to contribute more than those of query graph, with the exception of query density, suggesting that the data graph has a stronger effect on algorithm performance.

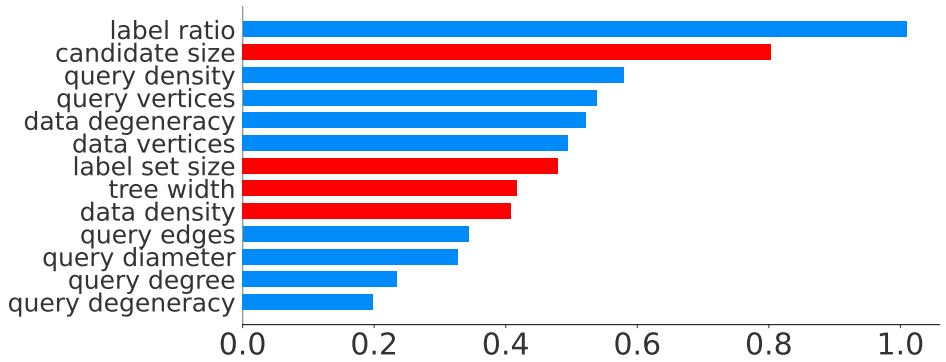


Fig. 15. Average Shapley values; red bars for RECALGO features; blue for those added in SUBLIME.

Table 12. Training time and number of epochs.

Ours	training time (sec)	epochs
TOPX	$14.27_{\pm 4.83}$	$255.80_{\pm 92.61}$
INCY	$23.31_{\pm 5.31}$	$359.00_{\pm 82.88}$
WEIGHTED	$96.15_{\pm 140.39}$	$1629.40_{\pm 2473.25}$

5.9 Cost of training and advance preparation

We also study the cost of training the machine learning model and collecting the subgraph matching data mentioned in Figure 5. Table 12 shows the average training time and number of epochs. Our methods complete training in at most a few hundred seconds. The training time sometimes takes longer than average, as we use different learning rates chosen from hyper-parameter tuning, leading to high variance. As WEIGHTED often employs lower learning rates on average, it requires higher training time and number of epochs. Still, our framework trains machine learning models efficiently.

The cumulative time for collecting subgraph matching data is 20 086 165 sec $\approx$ 232 days. Since we use a 40-thread machine, it cost $\frac{232}{40} = 5.8$ days to collect the data. As Figure 13 shows, competitive performance is achievable with reduced training data, making our methods practical for deployment. Besides, other baselines also require performance data to make their choices. Since the best-performing algorithm differs across datasets and queries, the need to collect empirical performance data arises in real-world cases even when not using machine learning.

5.10 Time-to-k measure

We have used the EPS measure for the sake of compatibility with prior work [42]. We here use time-to-k. We train and evaluate SUBLIME on subgraph matching under this measure with k set to 10^5 and a time limit of 10 seconds. Tables 13 and 14 present our results on average MRR and time-to-k, respectively. SUBLIME methods outperform baselines on all datasets on MRR and six datasets on EPS. These results show that SUBLIME successfully selects high-performance algorithms for the time-to-k measure. Nevertheless, while EPS and time-to-k are effective when computing all embeddings is infeasible, their scope is limited to the regime until a subset of embeddings is found. Developing measures for more comprehensive and fair evaluation is therefore an important direction for future work.

Table 13. MRR for time-to-k across datasets.

MRR	cs	db	hm	tw	wn	ys	yt
VOTED-D	0.2285	0.3845	0.2057	0.3319	0.2191	0.3266	0.3058
VOTED	0.1974	0.3845	0.1838	0.0365	0.0749	0.3266	0.3128
RECALGO	0.0810	0.1133	0.1064	0.0758	0.0347	0.0537	0.1647
Ours TOPX	**0.3244**	0.4437	**0.2861**	**0.3504**	0.3304	0.3729	0.3869
Ours INCY	0.3223	0.4444	0.2801	0.3483	**0.3402**	0.3724	**0.3918**
Ours WEIGHTED	0.3046	**0.4543**	0.2714	0.3485	0.3279	**0.3753**	0.3703

Table 14. Time-to-k (sec) across datasets.

time-to-k	cs	db	hm	tw	wn	ys	yt
VOTED-D	0.8044	1.7518	2.1448	1.8876	**1.4430**	1.0245	2.8632
VOTED	2.3519	1.7518	2.5296	4.1338	6.5318	1.0245	2.8602
RECALGO	5.6119	8.3950	7.8695	8.2705	9.8941	4.6032	8.8010
Ours TOPX	0.8545	1.5187	1.4220	1.9614	2.0222	0.4853	1.5462
Ours INCY	0.8579	1.5272	1.4410	1.9766	2.0323	0.5177	1.5090
Ours WEIGHTED	**0.7681**	**1.2534**	**1.2806**	**1.6918**	1.5427	**0.3677**	**1.3121**

5.11 Experiments on subgraph coverage

Here, we apply SUBLIME to the subgraph coverage problem, for which no prior work recommends algorithms. For the record, Figure 16 reports the cumulative cost of subgraph coverage for each algorithm and the total coverage each algorithm achieves within a 10-second time limit after preprocessing.

To motivate the application of SUBLIME to subgraph coverage, we study how often each of the twelve algorithms achieves the top coverage on seven datasets; Figure 17 shows the resulting

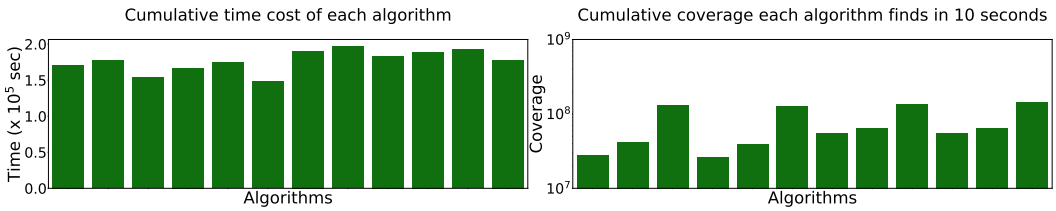


Fig. 16. Cumulative runtime and coverage.

heatmap, in a similar spirit to Figure 1. As multiple algorithms may return all results within the 10-second time limit after preprocessing, there are ties on the top rank, leading to a high count of top performers. Still, there are also differences in the distribution of top performers across data sets, e.g., between cs and tw. Figure 18 presents the average coverage per rank; as with EPS (see Figure 2), average coverage does not fall linearly vs. rank and the difference can be large; for instance, on the wn dataset, the top algorithm yields more than 10× the coverage of the lowest-rank algorithm.

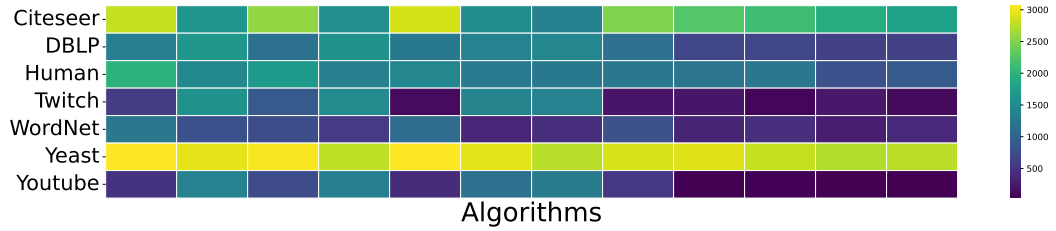


Fig. 17. Heatmap of how often algorithm achieves best coverage; in Yeast, several algorithms do.

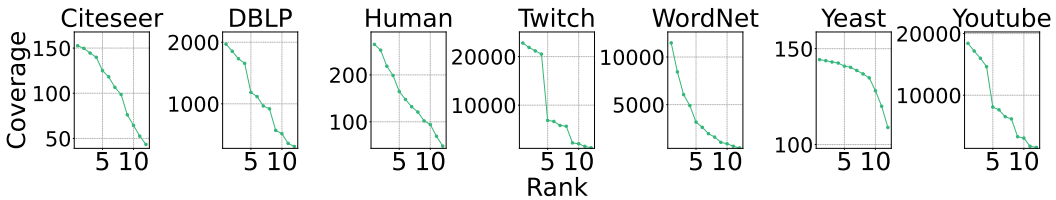


Fig. 18. Average coverage per rank, subgraph coverage; coverage at rank i obtained by the algorithm ranked i.

Table 15. MRR in subgraph coverage.

MRR	cs	db	hm	tw	wn	ys	yt
VOTED-D	0.9276	0.5502	0.7880	0.5897	0.5654	0.9618	0.5527
VOTED	0.9177	0.4314	0.7880	0.3160	0.5654	0.9618	0.3208
Ours TOPX	**0.9350**	**0.6799**	**0.8258**	**0.8089**	**0.6536**	**0.9648**	**0.6935**
Ours INCY	0.9242	0.6667	0.8040	0.8039	0.6503	0.9567	0.6736
Ours WEIGHTED	0.9215	0.6692	0.7879	0.7983	0.6315	0.9539	0.6714

Table 16. Normalized coverage values.

Coverage	cs	db	hm	tw	wn	ys	yt
VOTED-D	0.9837	0.6432	**0.9211**	0.6285	0.6554	**0.9930**	0.6164
VOTED	0.9781	0.5918	**0.9211**	0.4408	0.6554	**0.9930**	0.5588
Ours TOPX	**0.9847**	0.8109	0.9013	**0.9198**	0.7188	0.9909	0.8360
Ours INCY	0.9791	0.5918	0.9059	0.9177	0.7228	0.9882	0.8340
Ours WEIGHTED	0.9811	**0.8225**	0.9158	0.9171	**0.7251**	0.9893	**0.8545**

Tables 15 and 16 show SUBLIME's results on subgraph coverage. On MRR, SUBLIME methods outperform baselines on four datasets and achieve more than 0.6 MRR score, indicating that they

can select first or second best algorithms on average. The MRR score is better than that in subgraph matching because there are fewer options of algorithms (i.e., we have 12 algorithms), and this makes it easy to select the best one. Furthermore, when evaluating with coverage, some algorithms are given a tie rank, especially on ys. Consequently, the MRR score getting higher. On coverage, our methods outperform baselines on five datasets (e.g., cs, db, tw, wn, and yt). On hm and ys, our methods achieve high coverage value close to baselines.

We also evaluate the robustness of SUBLIME to OOD query sizes, datasets, and training data size on subgraph coverage. Figure 19 presents the average MRR and normalized coverage for OOD query sizes. As in subgraph matching experiments, we test on various query sizes (i.e., Q_{16}, Q_{32}, Q_{48}, Q_{96}, Q_{112}, and Q_{128}). SUBLIME methods outperform baselines in MMR on almost all datasets for all OOD query sizes, and also perform better than baselines on coverage. For instance, on yt for Q_{128}, SUBLIME methods achieve twice as much coverage as the baselines.

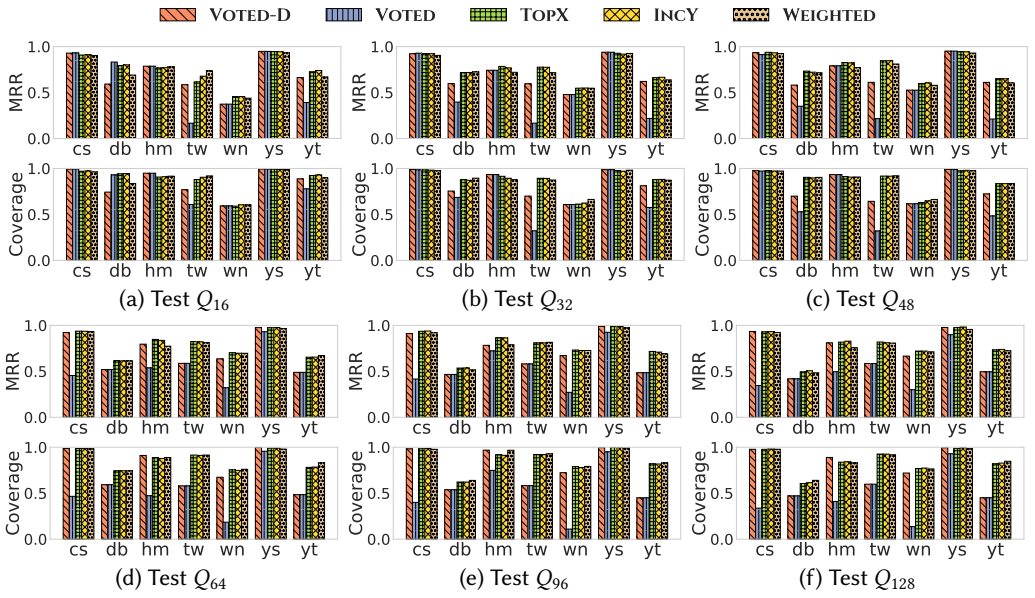


Fig. 19. Subgraph coverage: MRR and coverage on OOD query sizes, removing a query size from training.

Figure 20 shows the impact of training data size on average MRR and normalized coverage. The default training set contains about 1900 queries per dataset. The amount of queries is larger than in the subgraph matching experiment, as we use more query sizes in subgraph coverage. SUBLIME methods retain their performance on both measures, while baselines suffer. When reducing the training set size to 800, the performance of baselines drops further and SUBLIME outperforms baselines in all datasets.

Figure 21 shows the average MRR and average normalized coverage against OOD datasets. SUBLIME methods outperform VOTED on five datasets in both MRR and coverage. In almost all cases, SUBLIME methods achieve more than 0.5 MRR score. Even though SUBLIME methods fail to outperform VOTED, their MRR and coverage are still high, around 0.9 score on both metrics.

Figure 22 presents Shapley values, following the same methodology as applied for Section 5.8. In this case, *data degeneracy* has the largest contribution to prediction, while the relative importance of *candidate size* and *label ratio* is reduced compared to their impact in subgraph matching (Figure 15).

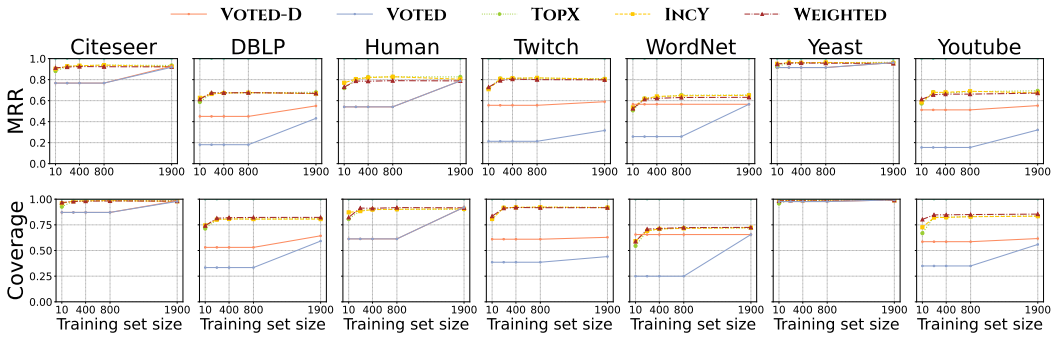


Fig. 20. Impact of training data size on MRR and coverage, subgraph coverage.

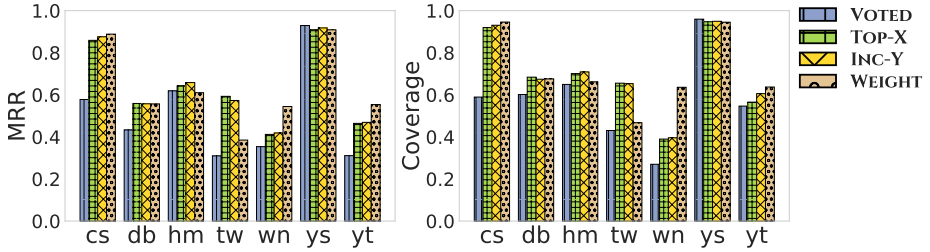


Fig. 21. MRR & coverage on OOD data, subgraph coverage.

This outcome indicates that SUBLIME captures the specific relationship between features and performance for each problem.

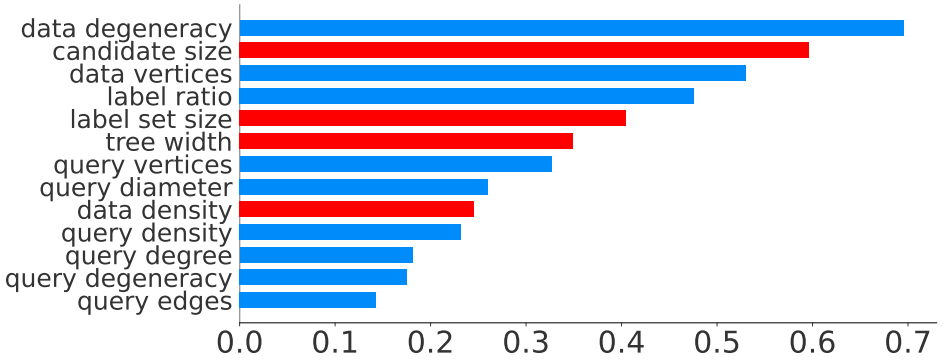


Fig. 22. Average Shapley values, subgraph coverage; red bars: RECALGO features; blue: added in SUBLIME.

Overall, SUBLIME methods outperform baselines in almost all cases. In addition, we show the robustness to the OOD datasets and to the OOD query sizes.

5.12 Discussion

OOD datasets. While SUBLIME excels in in-distribution settings, generalization to out-of-distribution (OOD) data remains a challenge. Yet in practical subgraph matching scenarios, the target data is available in advance. Here, we demonstrate that fine-tuning SUBLIME even with a limited amount of query graphs of a target dataset leads to a significant performance recovery, highlighting its high potential to adapt to OOD data. Figure 23 shows how the performance of SUBLIME on tw, trained on datasets other than tw, which is initially degraded as in Tables 4 and 5 of Section 5.3, gradually recovers by fine-tuning the pretrained model with 10, 100, and 200 additional in-distribution query

graphs. MRR improves by nearly four times when tuning with 100 queries. These results suggest that the challenges posed by OOD datasets can be attenuated through fine-tuning. To render SUBLIME more robust to OOD datasets, we may adopt strategies from related studies [20, 23], e.g., augment the training data with synthetic graphs of diverse structural properties, which would enable learning from a broader distribution.

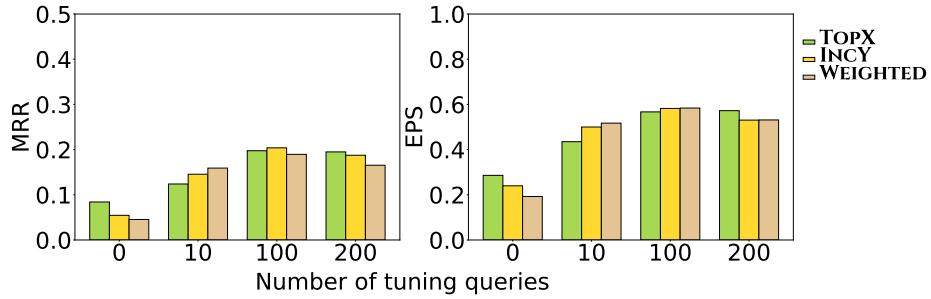


Fig. 23. Performance improvement via fine-tuning on tw; 0-tuning denotes the results on OOD dataset.

Features. While we have employed empirically defined graph features, we may also explore representation learning to automatically capture richer structural features by incorporating Graph Neural Networks (GNNs) into our featurizer [21, 39]. Nevertheless, GNNs may increase inference time and reduce performance in OOD settings. We aim to explore this potential in the future.

Impact of node ID permutation. It is conceivable that the particular node ID permutation used affects performance. To test this hypothesis, we examine whether the algorithm of choice for a given data graph remains consistent across permutations. Figure 24 presents five performance heatmaps generated by randomly permuting node IDs, including the heatmap of Figure 1. Notably, node ID permutation does not significantly differentiate performance.

Classification vs. Regression. Our machine learning model is based on a classification task as Section 4.3 defines. Alternatively, we train the WEIGHTED model as a regressor that directly predicts normalized EPS using the Mean Squared Error (MSE) loss. Table 17 juxtaposes the average MRR and normalized EPS for the regression-based and the classification-based approach on subgraph matching. Results are comparable on both metrics, while the classification-based solution we opted for performs slightly better.

Table 17. Comparison between Classification and Regression.

Measure	Task	cs	db	hm	tw	wn	ys	yt
MRR	Classification	0.3166	0.3460	0.2127	0.2669	0.3054	0.2920	0.2934
	Regression	0.3179	0.3522	0.2105	0.2649	0.2920	0.2862	0.2793
EPS	Classification	0.5276	0.6206	0.5705	0.6967	0.6594	0.3964	0.6827
	Regression	0.5231	0.6297	0.5638	0.6924	0.6502	0.3922	0.6536

6 Conclusion & Future work

We introduced the problem of selecting a well-performing algorithm for subgraph matching and coverage problems via machine learning and proposed SUBLIME, a novel framework for this task. Our experimental study showed that SUBLIME outperforms baselines that select algorithms by rule-based models or use the algorithm with the best average performance in both subgraph matching and subgraph coverage problems.

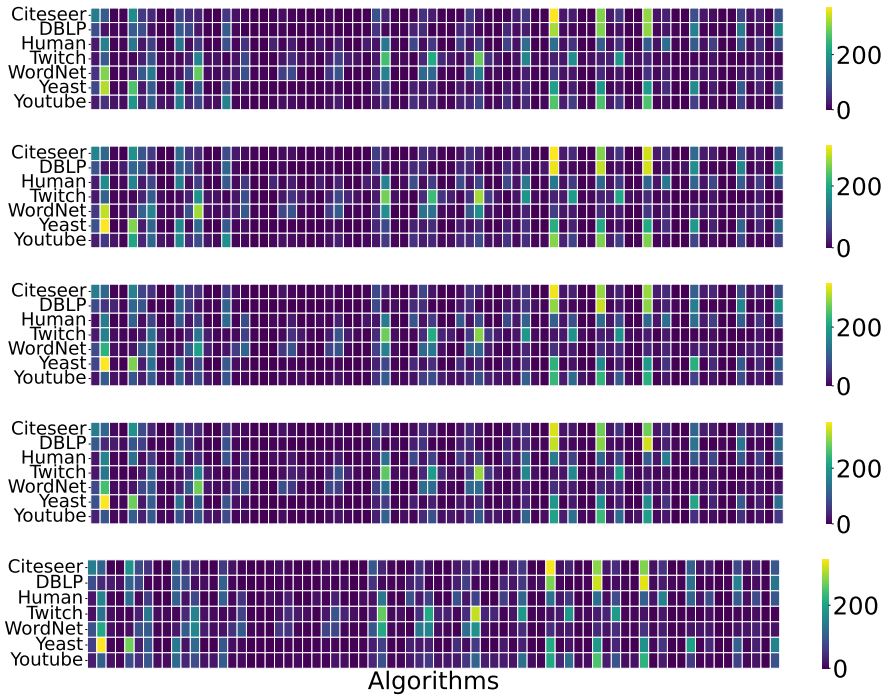


Fig. 24. Node ID permutation on data graph.

Since this paper is the first to examine this problem and aims to investigate the potential, there remains significant space to extend it by exploring more sophisticated models and strategies for generating training data. In the future, we aim to study the tailoring of models, training queries, and features, and build foundation models for subgraph matching algorithm and coverage selection. Furthermore, a systematic investigation of out-of-distribution generalization is a promising direction for future work.

Acknowledgments

The authors are grateful to the University of Copenhagen for their supports and insightful discussions. This study was supported by ASPIRE JPMJAP2328 and JSPS KAKENHI JP23K28096. Konstantinos Skitsas acknowledges the support of DFF.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-Generation Hyperparameter Optimization Framework. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.* 2623–2631.

[2] Yoshua Bengio. 2012. Practical Recommendations for Gradient-based Training of Deep Architectures. (2012), 437–478.

[3] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proc. ACM SIGMOD Int. Conf. Manag. Data.*

[4] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A Subgraph Isomorphism Algorithm and its Application to Biochemical Data. *BMC Bioinformatics* 14 (2013), 1–13.

[5] Leo Breiman, J. H. Friedman, Richard A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Wadsworth.

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.* 785–794.

[7] Wenfei Fan. 2012. Graph Pattern Matching Revised for Social Network Analysis. In *Proc. Int. Conf. Database Theory.* 8–21.

[8] Yurun Ge and Andrea L. Bertozzi. 2021. Active Learning for the Subgraph Matching Problem. In *Proc. IEEE Int. Conf. Big Data.*

[9] Shuyang Guo, Wenjin Xie, Ping Lu, Ting Deng, Richong Zhang, Jianxin Li, Xiangping Huang, and Zhongyi Liu. 2025. Improving Subgraph Matching by Combining Algorithms and Graph Neural Networks. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*

[10] 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 *Proc. ACM SIGMOD Int. Conf. Manag. Data.* 1429–1446.

[11] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proc. ACM SIGMOD Int. Conf. Manag. Data.* 1587–1602.

[12] Juris Hartmanis. 1982. Computers and Intractability: A Guide to the Theory of NP-completeness. *SIAM Rev.* 24, 1 (1982), 90.

[13] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proc. ACM SIGMOD Int. Conf. Manag. Data.* 405–418.

[14] 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 *Proc. ACM SIGMOD Int. Conf. Manag. Data.* 925–937.

[15] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming Subgraph Isomorphism for RDF Query Processing. *Proc. VLDB Endow.* 8, 11 (2015).

[16] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proc. Int. Conf. Learn. Represent.*

[17] Zixun Lan, Ye Ma, Limin Yu, Linglong Yuan, and Fei Ma. 2023. Aednet: Adaptive Edge-deleting Network for Subgraph Matching. *Pattern Recogn.* 133 (2023), 109033.

[18] Zixun Lan, Limin Yu, Linglong Yuan, Zili Wu, Qiang Niu, and Fei Ma. 2023. Sub-GMN: The Neural Subgraph Matching Network Model. In *Proc. Int. Congr. Image Signal Process. BioMed. Eng. Inform.* 1–7.

[19] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proc. VLDB Endow.* 6, 2 (2012), 133–144.

[20] Haoyang Li, Xin Wang, Ziwei Zhang, and Wenwu Zhu. 2025. Out-of-Distribution Generalization on Graphs: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.* (2025).

[21] Ziming Li, Yuequn Dou, Youhuan Li, Xinhuan Chen, and Chuxu Zhang. 2025. RSM: Reinforced Subgraph Matching Framework with Fine-grained Operation based Search Plan. In *Proc. ACM Int. Conf. Web Search Data Min.* 475–483.

[22] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. 2020. Neural Subgraph Matching. *arXiv preprint arXiv:2007.03092* (2020).

[23] Nikolai Merkel, Ruben Mayer, Tawkir Ahmed Fakir, and Hans-Arno Jacobsen. 2023. Partitioner Selection with EASE to Optimize Distributed Graph Processing. In *Proc. IEEE Int. Conf. Data Eng.* 2400–2414.

[24] Michael Ogunsanya, Joan Isichei, and Salil Desai. 2023. Grid Search Hyperparameter Tuning in Additive Manufacturing Processes. *Manufacturing Letters* 35 (2023), 1031–1042.

[25] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 1–45.

[26] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.

[27] Neil Robertson and Paul D. Seymour. 1986. Graph Minors. II. Algorithmic Aspects of Tree-width. *Journal of algorithms* 7, 3 (1986), 309–322.

[28] Reuven Rubinstein. 1999. The Cross-entropy Method for Combinatorial and Continuous Optimization. *Methodology and computing in applied probability* 1 (1999), 127–190.

[29] Stephen B Seidman. 1983. Network Structure and Minimum Degree. *Social networks* 5 (1983), 269–287.

[30] Zhichao Shi, Youhuan Li, Ziming Li, Yuequn Dou, Xionghu Zhong, and Lei Zou. 2025. MatCo: Computing Match Cover of Subgraph Query over Graph Data. *Proc. ACM Manag. Data* (2025), 1–24.

[31] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. PILOS: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *Proc. IEEE Int. Conf. Data Eng.* 1180–1193.

[32] Konstantinos Skitsas, Yuya Sasaki, Davide Mottin, and Panagiotis Karras. 2025. Mix & Match: : Subgraph Matching for Absolute Coverage. *Proc. VLDB Endow.* 18, 13 (2025), 5610–5622.

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[34] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proc. ACM SIGMOD Int. Conf. Manag. Data.* 1083–1098.

[35] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* 14, 2 (2020), 176–188.

[36] Julian R Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.
[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proc. Adv. Neural Inf. Process. Syst.*, Vol. 30.
[38] Todd L. Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. Int. Conf. Database Theory*. 96–106.
[39] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. 2022. Reinforcement Learning Based Query Vertex Ordering Model for Subgraph Matching. In *Proc. IEEE Int. Conf. Data Eng.* 245–258.
[40] Linglin Yang, Lei Zou, and Chunshan Zhao. 2025. NeuSO: Neural Optimizer for Subgraph Queries. *arXiv preprint arXiv:2509.23775* (2025).
[41] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1 (2023), 15:1–15:26.
[42] 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.
[43] Gaoping Zhu, Xuemin Lin, Ke Zhu, Wenjie Zhang, and Jeffrey Xu Yu. 2012. TreeSpan: Efficiently Computing Similarity All-matching. In *Proc. ACM SIGMOD Int. Conf. Manag. Data*. 529–540.

Received October 2025; revised January 2026; accepted February 2026