Dynamic Subgraph Matching via Cost-Model-based Vertex Dominance Embeddings (Technical Report)

Yutong Ye
East China Normal University
Shanghai, China
52205902007@stu.ecnu.edu.cn

Xiang Lian
Kent State University
Kent, United States
xlian@kent.edu

Nan Zhang
East China Normal University
Shanghai, China
51255902058@stu.ecnu.edu.cn

Mingsong Chen
East China Normal University
Shanghai, China
mschen@sei.ecnu.edu.cn

ABSTRACT

In many real-world applications such as social network analysis, knowledge graph discovery, biological network analytics, and so on, graph data management has become increasingly important and has drawn much attention from the database community. While many graphs (e.g., Twitter, Wikipedia, etc.) are usually involving over time, it is of great importance to study the *dynamic subgraph matching* (DSM) problem, a fundamental yet challenging graph operator, which continuously monitors subgraph matching results over dynamic graphs with a stream of edge updates. To efficiently tackle the DSM problem, we carefully design a novel *vertex dominance embedding* approach, which effectively encodes vertex labels that can be incrementally maintained upon graph updates. Inspire by low pruning power for high-degree vertices, we propose a new *degree grouping* technique over basic subgraph patterns in different degree groups (i.e., groups of star substructures), and devise *degree-aware star substructure synopses* (DAS^3) to effectively facilitate our designed *vertex dominance* and *range pruning strategies*. We develop efficient algorithms to incrementally maintain dynamic graphs and answer DSM queries. Through extensive experiments, we confirm the efficiency of our proposed approaches over both real and synthetic graphs.

PVLDB Reference Format:
Yutong Ye, Xiang Lian, Nan Zhang, and Mingsong Chen. Dynamic Subgraph Matching via Cost-Model-based Vertex Dominance Embeddings (Technical Report). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

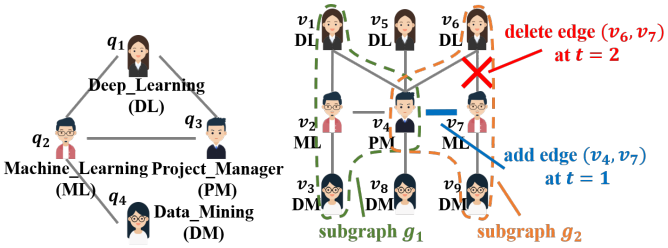
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/JamesWhiteSnow/DSM.

1 INTRODUCTION

For the past few decades, the *subgraph matching* problem has been extensively studied as a fundamental operator in the graph data management [13, 38, 46, 61, 67, 72] for many real-world applications such as social network analysis, knowledge graph discovery, and pattern matching in biological networks. Given a large-scale data graph G, a subgraph matching query finds all the subgraphs of G that are isomorphic to a given query graph q.

While many previous works [8–10, 26, 30, 35, 53, 58] usually considered the subgraph matching over static graphs, real-world graph data are often dynamic and change over time. For example, in social networks, friend relationships among users may be subject to changes (e.g., adding or breaking up with friends); similarly, in collaboration networks, people may start to collaborate on some project and then suddenly stop the collaboration for years. In these scenarios, it is important, yet challenging, to conduct the subgraph matching over such a large-scale, dynamic data graph G_D, upon updates (e.g., edge insertions/deletions). In other words, we need to continuously monitor subgraphs (e.g., user communities or collaborative teams) in G_D that follow query graph patterns q.

Below, we give an example of the subgraph matching over dynamic collaboration networks in the expert team search application.



(a) query graph q (b) dynamic collaboration network G_D

Figure 1: An example of the subgraph matching in dynamic collaboration networks G_D.

Example 1. ***(Monitoring Project Teams in Dynamic Collaboration Networks [19])*** *Due to numerous requests from departments/companies for recruiting (similar) project teams, a job search advisor may want to monitor the talent market (especially teams of experts) in dynamically changing collaboration networks.*

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

Consider a toy example of a collaboration network G_D in Figure 1. Figure 1(a) shows a desirable expert team pattern q, which is represented by a graph of 4 experts q_1~q_4, where each expert vertex q_i is associated with one's skill/role keyword (e.g., q_1 with keyword "Deep_Learning" and q_3 with "Project_Manager"), and each edge indicates that two experts had the collaboration before (e.g., edge (q_1, q_3)).

Figure 1(b) provides an example of a dynamic collaboration network G_D, where edges are inserted or deleted over time. For example, a new edge (v_4, v_7) is inserted at timestamp $t = 1$, implying that two experts v_4 and v_7 start to collaborate with each other in a team. Similarly, at timestamp $t = 2$, an existing edge (v_6, v_7) is removed from the graph G_D, which may indicate that experts v_6 and v_7 have not collaborated with each other for a certain period of time (e.g., two years).

In this example, the job search advisor can register a dynamic subgraph matching query over G_D to continuously identify subgraphs of G_D that match with the query graph pattern q. At timestamp $t = 1$, subgraphs g_1 and g_2 are the subgraph matching answers, whereas at timestamp $t = 2$, subgraph g_1 is the only answer (since the deletion of edge (v_6, v_7) invalidates the subgraph g_2). ■

Dynamic subgraph matching has many other real applications. For example, in the application of anomaly detection for online shopping [49], the interactions (e.g., purchases, views, likes, etc.) among users/products over time form a dynamic social/transaction graph, and it is important to detect some abnormal events such as fraudulent activities (i.e., query graph patterns) over such a dynamically changing graph.

Inspired by the examples above, we formulate a *dynamic subgraph matching* (DSM) query over a dynamic graph, which continuously monitors subgraph matching results, upon updates to the data graph.

Prior Works: Due to NP-completeness of the subgraph isomorphism [15, 21, 22], dynamic subgraph matching is not tractable. To tackle this problem, several exact algorithms have been proposed [14, 20, 33, 34, 37, 40, 43], which consist of three categories: *recomputation-based* [20], *direct-incremental* [37], and *index-based incremental algorithms* [14, 33, 34, 40, 43]. The recomputation-based algorithms re-compute all the subgraph matching answers at each timestamp from scratch, the direct-incremental algorithms incrementally calculate new matching results from dynamic graphs upon updates, and index-based incremental ones incrementally maintain matching results via auxiliary indexes built over query answers.

Our Contributions: Our work falls into the category of the direct-incremental algorithm. Different from existing works that directly use structural information (e.g., vertex label and neighbors' label set) to filter out vertex candidates, we design a novel and effective *vertex dominance embedding* technique for candidate vertex/subgraph retrieval. Specifically, our proposed vertex dominance embeddings can be incrementally maintained in dynamic graphs, and transform our DSM problem into a *dominating region search* problem in the embedding space with no false dismissals. To enhance the pruning power of our embeddings for high-degree vertices, we propose a new *degree grouping* approach for vertex embeddings over basic subgraph patterns in different degree groups (i.e., groups of star substructures). We also devise a cost model to further guide the vertex embeddings to effectively achieve high pruning power. Finally, we develop algorithms to incrementally maintain dynamic graphs and answer DSM queries efficiently.

Table 1: Symbols and Descriptions

Symbol	Description
G	a static graph
G_D	a dynamic graph
ΔG_t	a set of graph updates at timestamp t
G_t	a snapshot graph of G_D at timestamp t
$e+$ (or $e-$)	an insertion (or deletion) of edge e
q	a query graph
Q	a set of query graph patterns
x_i	a SPUR vector of vertex v_i
y_i	a SPAN vector of vertex v_i's 1-hop neighbors
$o(v_i)$ (or $o(g_{v_i}), o(s_{v_i})$)	a vertex dominance embedding vector

Specifically, in this paper, we make the following contributions:

(1) We formally define the *dynamic subgraph matching* (DSM) problem in Section 2, and propose a general framework for DSM query answering in Section 3.
(2) We carefully design incremental vertex dominance embeddings to facilitate dynamic subgraph matching in Section 4.
(3) We devise a degree grouping technique to enhance the pruning power of vertex embeddings, and construct *degree-aware star substructure synopses* (DAS^3) to support the dynamic subgraph matching in Section 5.
(4) We design effective synopsis pruning strategies, and develop efficient DSM processing algorithms in Section 6.
(5) We present a novel cost-model-guided vertex embeddings to further improve the pruning power in Section 7.
(6) We demonstrate through extensive experiments the efficiency and effectiveness of our DSM processing algorithm over real/synthetic graphs in Section 8.

Section 9 reviews previous works on dynamic graph management and graph embeddings. Finally, Section 10 concludes this paper.

2 PROBLEM DEFINITION

Table 1 depicts the commonly used symbols in this paper.

2.1 Static Graph Model

We give the model of a static undirected, vertex-labeled graph G.

Definition 1. ***(Static Graph,** G) A static graph, G, is denoted as a triple $(V(G), E(G), L(G))$, where $V(G)$ is a set of vertices v_i, $E(G)$ is a set of edges $e_{i,j} = (v_i, v_j)$ between vertices v_i and v_j, and $L(G)$ is a label function from each vertex $v_i \in V(G)$ to a label $l(v_i)$.* ■

The static graph model (as given in Definition 1) has been widely used in various real-life applications to reflect relationships between different entities, for example, social networks [1], Semantic Web [29], transportation networks [50], biological networks [36], citation networks [70], and so on.

The Graph Isomorphism: Next, we define the classic graph isomorphism problem between two graphs.

Definition 2. ***(Graph Isomorphism)** Given two graphs G_1 and G_2, graph G_1 is isomorphic to graph G_2 (denoted as $G_1 \equiv G_2$), if there exists a bijection mapping function $M : V(G_1) \Rightarrow V(G_2)$, such that: i) $\forall v_i \in V(G_1)$, we have $L(v_i) = L(M(v_i))$, and; ii) $\forall e_{i,j} \in E(G_1)$, edge $e_{M(v_i),M(v_j)} = (M(v_i), M(v_j)) \in E(G_2)$ holds.* ■

In Definition 2, the graph isomorphism problem checks whether graphs G_1 and G_2 exactly match with each other.

The Subgraph Matching Problem: The subgraph isomorphism (or subgraph matching) problem is defined as follows.

Definition 3. ***(Subgraph Matching)*** *Given graphs G and g, a subgraph matching problem identifies subgraphs g' of G (i.e., $g' \subseteq G$) such that g and g' are isomorphic.* ■

Note that, the subgraph matching problem is NP-complete [21].

2.2 Dynamic Graph Model

Real-world graphs are often continuously evolving and updated. Examples of such dynamic graphs include social networks with updates of friend relationships among users, bibliographical networks with new collaboration relationships, and so on. In this subsection, we provide the data model for dynamic graphs with update operations below.

Definition 4. ***(Dynamic Graph, G_D)*** *A dynamic graph, G_D, consists of an initial graph G_0 (as given by Definition 1) and a sequence of graph update operations $\Delta\mathcal{G} = \{\Delta G_1, \Delta G_2, \ldots, \Delta G_t, \ldots\}$ on G. Here, ΔG_t is a graph update operator, in the form of $(e+, t)$ or $(e-, t)$, which indicates either an insertion (+) or a deletion (−) of an edge e at timestamp t, respectively.* ■

Snapshot Graph, G_t: From Definition 4, after applying to initial graph G all the graph update operations up to the current timestamp t (i.e., ΔG_1, ΔG_2, ..., and ΔG_t), we can obtain a snapshot of the dynamic graph G_D, denoted as G_t.

Discussions on the Edge Insertion: For the edge insertion $(e+, t)$ in Definition 4, there are three cases:

- both ending vertices of e exist in the graph snapshot G_{t-1};
- one ending vertex of e exists in G_{t-1} and the other one is a new vertex, and;
- both ending vertices of e are new vertices.

We will later discuss how to deal with the three cases of edge insertions above for dynamic graph maintenance and incremental query answering.

2.3 Dynamic Subgraph Matching Queries

In this subsection, we define a *dynamic subgraph matching* (DSM) query in a large dynamic graph G_D, which continuously monitors subgraph matching answers for a set, Q, of query graph patterns.

Definition 5. ***(Dynamic Subgraph Matching, DSM)*** *Given a dynamic graph G_D, a set, Q, of query graph patterns q, and a timestamp t, a dynamic subgraph match (DSM) query maintains a subgraph matching answer set $A(q, t)$ for each query graph pattern $q \in Q$, such that any subgraph $g \in A(q, t)$ in G_D is isomorphic to the query graph q (denoted as $g \equiv q$).*

Alternatively, for each $q \in Q$, the DSM problem incrementally computes an update, $\Delta A(q, t)$, to the answer set $A(q, t-1)$, upon the change ΔG_t to G_{t-1}. ■

In Definition 5, the DSM query continuously maintains a subgraph matching answer set $A(q, t)$ (w.r.t. each query graph pattern $q \in Q$) over dynamic graph G_D (upon incremental updates) for a long period of time. The DSM problem is quite useful in real applications such as cyber-attack event detection and credit card fraud monitoring.

Algorithm 1: The Dynamic Subgraph Matching Framework

Input: i) a dynamic data graph G_D; ii) a set, Q, of registered query graph patterns, and; iii) current timestamp t
Output: the subgraph matching answer set, $A(q, t)$, for each registered query graph $q \in Q$

```
// Graph Maintenance Stage
generate a vertex dominance embedding vector o(v_i) for each vertex v_i ∈ G_0
build m synopses, Syn_j, over vertex embeddings with degree grouping
for each graph update ΔG_t ∈ ΔG do
    obtain G_t by applying the graph update ΔG_t over G_{t-1}
    maintain synopses Syn_j by updating the corresponding vertex embeddings
// DSM Query Answering
// Initial Subgraph Matching Answer Set Generation
for each query graph q ∈ Q do
    compute a query embedding vector o(q_i) of each vertex q_i ∈ V(q)
    find candidate vertex sets q_i.cand_set for query vertices q_i by accessing synopses Syn_j
    enumerate candidate subgraphs g from candidate vertices in q_i.cand_set
    obtain an initial subgraph matching answer set, A(q, t), of matching subgraphs g (≡ q)
// Continuous Subgraph Matching Answer Set Monitoring
for each graph update ΔG_t ∈ ΔG do
    for each query graph q ∈ Q do
        update candidate vertex sets q_i.cand_set by checking the updated vertices in ΔG_t
        compute answer changes ΔA(q, t) from the new candidate sets q_i.cand_set
        apply changes ΔA(q, t) to A(q, t − 1) and obtain the latest subgraph answers in A(q, t)
        return A(q, t)
```

2.4 Challenges

The subgraph matching problem in a data graph has been proven to be NP-complete [21]. Thus, it is even more challenging to design effective techniques for answering subgraph matching queries in the scenario of dynamic graphs. Due to frequent updates to the data graph, it is non-trivial how to effectively maintain a large-scale dynamic graph that can support effectively obtaining initial subgraph matching results, and continuously and efficiently monitor DSM answer sets, upon fast graph updates.

To tackle the DSM problem, in this paper, we will propose a novel and effective vertex dominance embedding technique over dynamic graphs, which transforms the subgraph matching to a dominance search problem in a graph-node-embedding space. Our proposed embedding technique can guarantee the subgraph matching with no false dismissals. Furthermore, we also design effective pruning, indexing, and incremental maintenance mechanisms to facilitate efficient and scalable dynamic subgraph matching.

3 THE DYNAMIC SUBGRAPH MATCHING FRAMEWORK

In this section, we illustrate a general framework for dynamic subgraph matching in Algorithm 1, which consists of two stages, *graph maintenance* and *DSM query answering* stages.

Specifically, in the graph maintenance stage, we first pre-process initial graph G_0, by constructing a set of m graph synopses, Syn_j, over vertex dominance embeddings $o(v_i)$ with m degree groups, respectively (lines 1-2). Then, for each graph update operation $\Delta G_t \in \Delta\mathcal{G}$, we incrementally maintain the dynamic graph G_t and synopses Syn_j (lines 3-5).

In the DSM query answering stage, for each newly registered query graph q, we obtain the initial query answer set over the current snapshot G_t of the dynamic graph G_D (lines 6-10). Then, we monitor the updates in query answer sets over continuously changing dynamic graph G_t (lines 11-16).

In particular, for initial answer set generation, we compute query vertex embeddings $o(q_i)$ from vertices $q_i \in V(q)$ in each query graph q (lines 6-7). Then, for each query vertex q_i, we find and store candidate vertices in a set $q_i.cand\_set$ by accessing Syn_j (line 8). After that, we enumerate candidate subgraphs g by assembling candidate vertices in $q_i.cand\_set$ (line 9), and refine/return matching subgraphs g in an initial subgraph matching answer set $A(q,t)$ (line 10).

Next, to answer the DSM query, for each graph update operation $\Delta G_t \in \Delta\mathcal{G}$ and each query graph $q \in Q$, we check the updated vertices in ΔG_t, update candidate vertex sets $q_i.cand\_set$, and monitor the changes $\Delta A(q,t)$ in $A(q,t)$ (lines 11-14). Finally, we obtain $A(q,t)$ by applying changes $\Delta A(q,t)$ to $A(q,t-1)$, and return $A(q,t)$ as the DSM query answer set (lines 15-16).

4 INCREMENTAL VERTEX DOMINANCE EMBEDDINGS

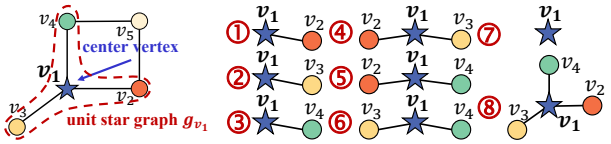
In this section, we will present our vertex dominance embeddings, which can be incrementally maintained, preserve dominance relationships between basic subgraph patterns (i.e., star subgraphs), and support efficient subgraph matching over dynamic graphs.

4.1 Preliminaries and Terminologies

We first introduce two terms of basic subgraph patterns (used for our proposed vertex dominance embedding), that is, *unit star subgraphs* and *star substructures*:

- **Unit Star Subgraph g_{v_i}:** A unit star subgraph g_{v_i} is defined as a (star) subgraph in G_t containing a center vertex v_i and its one-hop neighbors.
- **Star Substructure s_{v_i}:** A star substructure s_{v_i} is defined as a (star) subgraph of the unit star subgraph g_{v_i} in G_t (i.e., $s_{v_i} \subseteq g_{v_i}$), which shares the same center vertex v_i.

As an example in Figure 2(a), the unit star subgraph g_{v_1} in data graph G_t contains a center vertex v_1 and its 1-hop neighbors v_2, v_3, and v_4. Figure 2(b) illustrates 8 (= 2^3) possible star substructures s_{v_1} ($\subseteq g_{v_1}$), which are centered at vertex v_1 and with different combinations of v_1's 1-hop neighbors.



(a) unit star subgraph g_{v_1} (b) 8 possible star substructures s_{v_1}

Figure 2: Illustration of unit star subgraph and star substructures.

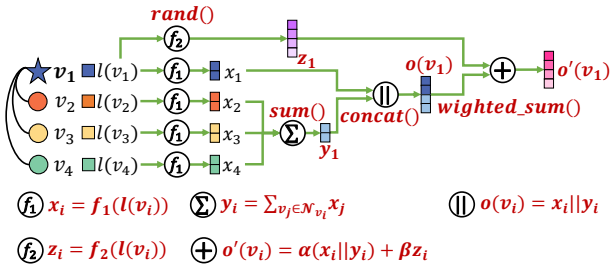


Figure 3: Illustration of vertex dominance embedding $o(v_1)$ and the optimized vertex dominance embedding $o'(v_1)$.

4.2 Vertex Dominance Embedding

In this subsection, we discuss how to obtain an embedding vector, $o(v_i)$, for each vertex v_i, which embeds labels of vertex v_i and its 1-hop neighbors (i.e., unit star subgraph or star substructure). Specifically, the embedding vector $o(v_i)$ consists of two portions, a *Seeded PseUdo Random* (SPUR) vector x_i and a *SPUR Aggregated Neighbor* (SPAN) vector y_i.

Seeded PseUdo Random (SPUR) Vector, x_i: In the dynamic graph G_D, each vertex v_i is associated with a label $l(v_i)$, which can be encoded by a nonnegative integer. By treating the vertex label $l(v_i)$ as the seed, we can generate a *Seeded PseUdo Random* (SPUR) vector, x_i, of arity d via a pseudo-random number generator $f_1(\cdot)$. That is, we have:

$$x_i = f_1(l(v_i)). \tag{1}$$

We denote $x_i[j]$ as the j-th element in SPUR vector x_i, where $1 \le j \le d$ and $x_i[j] > 0$.

Note that, since our randomized SPUR vector x_i is generated based on the seed (i.e., label $l(v_i)$ of each vertex v_i), vertices with the same label will have the same SPUR vector.

In the example of Figure 3, vertices $v_1 \sim v_4$ are associated with labels $l(v_1) \sim l(v_4)$, respectively. We generate their corresponding SPUR vectors, $x_1 \sim x_4$, via a randomized function $f_1(\cdot)$ in Eq. (1).

SPUR Aggregated Neighbor (SPAN) Vector, y_i: Next, for each vertex v_i, we consider its 1-hop neighbors, $v_j \in \mathcal{N}_{v_i}$, in a star subgraph pattern (i.e., the unit star subgraph g_{v_i} or star substructure s_{v_i}), and sum up their corresponding SPUR vectors x_j to generate a *SPur Aggregated Neighbor* (SPAN) vector y_i:

$$y_i = \sum_{\forall v_j \in \mathcal{N}_{v_i}} x_j, \tag{2}$$

where $\mathcal{N}_{v_i}$ is a set of 1-hop neighbors of v_i, and x_j are the SPUR vectors of 1-hop neighbors of v_i.

Taking Figure 3 as an example, the SPAN vector of the vertex v_1 is calculated by $y_1 = x_2 + x_3 + x_4$.

Vertex Dominance Embedding, $o(v_i)$: We define the *vertex dominance embedding* vector, $o(v_i)$, of each vertex v_i, by concatenating its SPUR vector x_i and SPAN vector y_i:

$$o(v_i) = x_i \,||\, y_i, \tag{3}$$

where $||$ is the concatenation operation of two vectors, and x_i and y_i are given in Eqs. (1) and (2), respectively.

Intuitively, vertex dominance embedding $o(v_i)$ encodes label features of a star subgraph pattern (i.e., g_{v_i} or s_{v_i}) containing center vertex v_i and its 1-hop neighbors. To distinguish vertex dominance embeddings $o(v_i)$ from different star subsgraph patterns, we also use notations $o(g_{v_i})$ or $o(s_{v_i})$.

As shown in Figure 3, the vertex dominance embedding of vertex v_1 can be computed by $o(v_1) = x_1 \,||\, y_1$.

4.3 Properties of Vertex Dominance Embedding

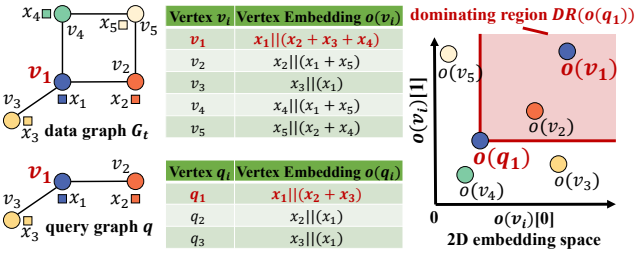
The Dominance Property of Vertex Embeddings: From Eq. (3), the vertex dominance embedding $o(v_i)$ is a combination of SPUR and SPAN vectors. Given any unit star subgraph g_{v_i} and its star substructure s_{v_i} (i.e., $s_{v_i} \subseteq g_{v_i}$), their vertex dominance embeddings always satisfy the *dominance* [11] (or equality) condition: $o(s_{v_i})[j] \le o(g_{v_i})[j]$, for all dimensions $1 \le j \le 2d$, which is denoted as $o(s_{v_i}) \preceq o(g_{v_i})$ (including the case where $o(s_{v_i}) = o(g_{v_i})$).

The reason for the property above is as follows. Since star patterns s_{v_i} and g_{v_i} have the same center vertex v_i, the vertex embeddings, $o(s_{v_i})$ and $o(g_{v_i})$, for s_{v_i} and g_{v_i}, respectively, must share the same SPUR vector x_i. Moreover, since $s_{v_i} \subseteq g_{v_i}$ holds, 1-hop neighbors of vertex v_i in s_{v_i} and g_{v_i} must satisfy the condition that $\mathcal{N}_{v_i}(s_{v_i}) \subseteq \mathcal{N}_{v_i}(g_{v_i})$. As the SPAN vectors are defined as the summed SPUR aggregates of 1-hop neighbors in $\mathcal{N}_{v_i}(s_{v_i})$ and $\mathcal{N}_{v_i}(g_{v_i})$, respectively, we have the dominance relationship between SPAN vectors $y_i(s_{v_i})$ and $y_i(g_{v_i})$ (i.e., $y_i(s_{v_i}) \preceq y_i(g_{v_i})$).

Lemma 4.1. *(The Dominance Property of Vertex Embeddings) Given a unit star subgraph g_{v_i} centered at vertex v_i and any of its star substructures s_{v_i} (i.e., $s_{v_i} \subseteq g_{v_i}$), their vertex dominance embeddings satisfy the condition that: $o(s_{v_i}) \preceq o(g_{v_i})$ (including $o(s_{v_i}) = o(g_{v_i})$) in the embedding space.*

Proof. For a given unit star subgraph g_{v_i} centered at vertex v_i and any of its star substructures s_{v_i} (i.e., $s_{v_i} \subseteq g_{v_i}$), since they share the same center vertex v_i (i.e., with the same label $l(v_i)$), they must have the same SPUR vector x_i.

Moreover, since $s_{v_i} \subseteq g_{v_i}$ holds, we also have $\mathcal{N}_{v_i}(s_{v_i}) \subseteq \mathcal{N}_{v_i}(g_{v_i})$, where $\mathcal{N}_{v_i}(s_{v_i})$ (or $\mathcal{N}_{v_i}(g_{v_i})$) is a set of v_i's 1-hop neighbors in subgraph s_{v_i} (or g_{v_i}). Thus, for SPAN vectors $y_i(s_{v_i})$ and $y_i(g_{v_i})$ of s_{v_i} and g_{v_i}, respectively, it must hold that: $y_i(s_{v_i})[k] \le y_i(g_{v_i})[k]$ for all dimensions k (since we have $y_i(s_{v_i})[k] = \sum_{\forall v_j \in \mathcal{N}_{v_i}(s_{v_i})} x_j[k] \le \sum_{\forall v_j \in \mathcal{N}_{v_i}(g_{v_i})} x_j[k] = y_i(g_{v_i})[k]$). In other words, their SPAN vectors satisfy the condition that $y_i(s_{v_i}) \preceq y_i(g_{v_i})$. Therefore, for $s_{v_i} \subseteq g_{v_i}$, their vertex dominance embeddings satisfy the condition that: $o(s_{v_i}) = (x_i||y_i(s_{v_i})) \preceq (x_i||y_i(g_{v_i})) = o(g_{v_i})$ (including $o(s_{v_i}) = o(g_{v_i})$) in the embedding space, which completes the proof. □



Vertex v_i	Vertex Embedding $o(v_i)$
v_1	$x_1 \|\| (x_2 + x_3 + x_4)$
v_2	$x_2 \|\| (x_1 + x_5)$
v_3	$x_3 \|\| (x_1)$
v_4	$x_4 \|\| (x_1 + x_5)$
v_5	$x_5 \|\| (x_2 + x_4)$

Vertex q_i	Vertex Embedding $o(q_i)$
q_1	$x_1 \|\| (x_2 + x_3)$
q_2	$x_2 \|\| (x_1)$
q_3	$x_3 \|\| (x_1)$

Figure 4: An example of vertex dominance embeddings in subgraph matching (|| is the concatenation operation of two vectors).

The Usage of the Vertex Embedding Property: Note that, a star substructure s_{v_i} ($\subseteq g_{v_i}$) can be a potential query unit star subgraph s_{q_i} from the query graph q (containing a center query vertex q_i and its 1-hop neighbors) for dynamic subgraph matching.

During the subgraph matching, given a query embedding vector $o(q_i)$ of query vertex q_i, we can identify those candidate matching vertices v_i in dynamic graph G_D with embeddings $o(v_i)$ dominated by $o(q_i)$. This way, we can convert the dynamic subgraph matching problem into a dominance search problem in the embedding space.

Example 2. *Figure 4 shows an example of our vertex dominance embedding for subgraph matching between a snapshot graph G_t and a query graph q. Based on Eq. (3), we can obtain the embedding for each vertex in G_t or q. For example, vertices v_1 and q_1 have the embeddings $o(v_1) = x_1||(x_2 + x_3 + x_4)$ and $o(q_1) = x_1||(x_2 + x_3)$, respectively.*

Since vertex q_1 in q matches with vertex v_1 in (a subgraph of) G_t, we can see that $o(q_1)$ is dominating $o(v_1)$ in a 2D embedding space (based on the property of vertex dominance embedding), which implies that g_{q_1} is potentially a subgraph of (i.e., matching with) g_{v_1}. Moreover, although $o(v_2)$ is dominated by $o(q_1)$, g_{v_2} does not match g_{q_1}. Thus, in this case, vertex v_2 is a false positive during the subgraph candidate retrieval. On the other hand, since $o(q_1)$ is not dominating $o(v_3)$ in the 2D embedding space, query vertex q_1 cannot match with vertex v_3 in graph G_t. ■

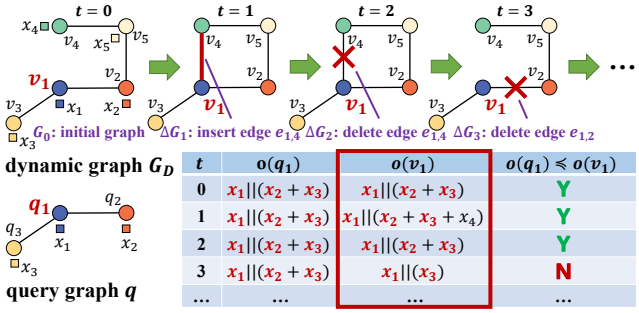
Ease of Incremental Updates for Vertex Dominance Embeddings: In the dynamic graph G_D, our proposed vertex dominance embedding $o(v_i)$ can be incrementally maintained, upon graph update operations in ΔG_t.

Specifically, any edge insertion $(e+, t)$ or deletion $(e-, t)$ (as given in Definition 4) will affect the embeddings of two ending vertices, v_i and v_j, of edge $e = (v_i, v_j)$. We consider incremental update of the embedding vector $o(v_i)$ for vertex v_i as follows.

- When v_i is a newly inserted vertex, we obtain its embedding vector $o(v_i)$ from scratch (i.e., $o(v_i) = x_i||x_j$), and;
- When v_i is an existing vertex in dynamic graph G_D, for an insertion (or deletion) operator of edge $e = (v_i, v_j)$, we update the SPAN vector, y_i, in $o(v_i)$ with $(y_i + x_j)$ (or $(y_i - x_j)$), where x_j is the SPUR vector of vertex v_j.

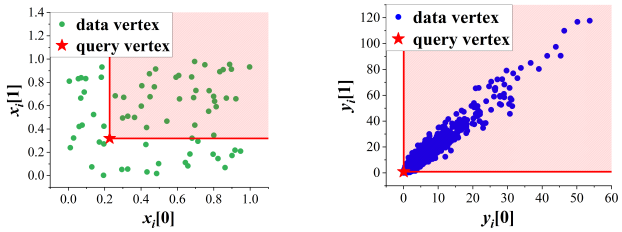
The embedding update of vertex v_j is similar and thus omitted.

Note that, for each graph update (i.e., edge insertion $(e+, t)$ or deletion $(e-, t)$), the time complexity of incrementally updating vertex dominance embedding vectors is given by $O(d)$, where d is the dimension of SPUR/SPAN vectors.



t	$o(q_1)$	$o(v_1)$	$o(q_1) \preccurlyeq o(v_1)$
0	$x_1 \| \| (x_2 + x_3)$	$x_1 \| \| (x_2 + x_3)$	Y
1	$x_1 \| \| (x_2 + x_3)$	$x_1 \| \| (x_2 + x_3 + x_4)$	Y
2	$x_1 \| \| (x_2 + x_3)$	$x_1 \| \| (x_2 + x_3)$	Y
3	$x_1 \| \| (x_2 + x_3)$	$x_1 \| \| (x_3)$	N
...	...	...	...

Figure 5: An example of vertex embeddings for DSM.



(a) SPUR vectors x_i (b) SPAN vectors y_i

Figure 6: An example of SPUR/SPAN vector distributions in vertex dominance embedding $o(v_i)$.

Example 3. *Figure 5 illustrates an example of using our vertex dominance embeddings to conduct the subgraph matching over dynamic graph G_D over time t from 0 to 3, where q is a given query graph. Consider two vertices v_1 and q_1 in graphs G_D and q, respectively. At timestamp $t = 0$, their vertex embeddings satisfy the dominance relationship, that is, $o(q_1) \preceq o(v_1)$, thus, vertex v_1 is a candidate matching with q_1.*

At timestamp $t = 1$, a new edge $e_{1,4}$ is inserted (i.e., ΔG_1), and the vertex embedding $o(v_1)$ changes from $x_1||(x_2+x_3)$ to $x_1||(x_2+x_3+x_4)$ (by including new neighbor v_4's SPUR vector x_4). Since $o(q_1) \preceq o(v_1)$ still holds, v_1 remains a candidate matching with q_1.

At timestamp $t = 2$, since edge $e_{1,4}$ is deleted (i.e., ΔG_2), the vertex embedding $o(v_1)$ changes back to $x_1||(x_2 + x_3)$. At timestamp $t = 3$, edge $e_{1,2}$ is deleted (i.e., ΔG_3), and $o(v_1)$ is updated with $x_1||(x_3)$ (by removing the expired neighbor v_2's SPUR vector x_2). In this case, $o(q_1) \preceq o(v_1)$ does not hold, and vertex v_1 fails to match with q_1. ■

4.4 Embedding Optimization with Base Vector z_i

Figure 6 shows the distributions of 2D SPUR and SPAN vectors, x_i and y_i, respectively, in vertex dominance embedding $o(v_i)$ over Yeast graph [58] (with 3,112 vertices and 12,519 edges). We can see that, in Figure 6(a), the SPUR vectors x_i generated by a seeded randomized function are distributed uniformly in the embedding space, whereas the SPAN vectors (i.e., the summed SPUR vectors) y_i are more clustered along the reverse diagonal line in Figure 6(a).

As mentioned in Section 4.3, given a query vertex q_i, we can use its embedding vector $o(q_i)$ as a query point to find the dominated embedding vectors $o(v_i)$ $(= x_i \,||\, y_i)$ in the 4D embedding space. Due to the scattered SPUR/SPAN vectors in the embedding space, it is very likely that some false alarms of embedding vectors $o(v_i)$ are included as candidate matching vertices (i.e., dominated by $o(q_i)$) .

In order to further enhance the pruning power, our goal is to update our vertex dominance embedding vector from $o(v_i)$ to $o'(v_i)$, such that the number of candidate vertices dominated by $o(q_i)$ is reduced (i.e., as small as possible). Specifically, we have the optimized embedding $o'(v_i)$, given by:

$$o'(v_i) = \alpha(x_i||y_i) + \beta z_i, \tag{4}$$

where α and β are positive constants ($\alpha \ll \beta$), and $z_i = f_2(l(v_i))$ is a random *base vector* of size $(2d)$, generated by a pseudo-random function $f_2(\cdot)$ with $l(v_i)$ as the seed. Please refer to Figure 3 on how to compute this optimized vertex dominance embedding.

Note that, since the base vector z_i is generated only based on the label $l(v_i)$ of center vertex v_i, different star subgraphs with the same center vertex label $l(v_i)$ will result in the same base vector z_i. Therefore, the new (optimized) vertex embedding vector $o'(v_i)$ in Eq. (4) still follows the dominance relationship, that is, $o'(q_i) \preceq o'(v_i)$ holds, if a query star pattern s_{q_i} is a subgraph of unit star subgraph g_{v_i} (i.e., $s_{q_i} \subseteq g_{v_i}$ and $l(q_i) \equiv l(v_i)$).

Lemma 4.2. *(The Dominance Property of the Optimized Vertex Embeddings) Given a unit star subgraph g_{v_i} centered at vertex v_i and any of its star substructures s_{v_i} (i.e., $s_{v_i} \subseteq g_{v_i}$), their optimized vertex embeddings (via base vector z_i) satisfy the condition that: $o'(s_{v_i}) \preceq o'(g_{v_i})$ (including $o'(s_{v_i}) = o'(g_{v_i})$).*

Proof. Given a unit star subgraph g_{v_i} centered at vertex v_i and any of its star substructures s_{v_i} (i.e., $s_{v_i} \subseteq g_{v_i}$), since they have the same center vertex v_i (with the same vertex label), their base vectors have the same values, i.e., $z_i(s_{v_i}) = z_i(g_{v_i})$, or equivalently $\beta z_i(s_{v_i}) = \beta z_i(g_{v_i})$ (for $\beta > 0$).

Due to the property of vertex dominance embeddings (as given in Lemma **??**), we have $o(s_{v_i}) \preceq o(g_{v_i})$, or equivalently $\alpha o(s_{v_i}) \preceq \alpha o(g_{v_i})$ (for $\alpha > 0$).

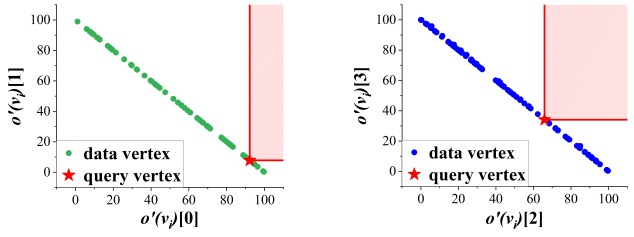
Therefore, we can derive that $\alpha o(s_{v_i}) + \beta z_i(s_{v_i}) \preceq \alpha o(g_{v_i}) + \beta z_i(g_{v_i})$, or equivalently $o'(s_{v_i}) \preceq o'(g_{v_i})$ (including $o'(s_{v_i}) = o'(g_{v_i})$), which completes the proof. □

Discussions on How to Design a Base Vector, z_i: To improve the pruning power, we would like to make the distribution of the updated vertex embeddings in Eq. (4) more dispersed along diagonal line (or plane), so that fewer false alarms (dominated by $o'(q_i)$) can be obtained during dynamic subgraph matching.

In this paper, we propose to design the base vector z_i as a random vector distributed on the unit diagonal line/hyperplane (L_1-norm) in the first quadrant. That is, we can normalize z_i to $\frac{z_i}{||z_i||_1}$, where $||\cdot||_1$ is L_1-norm. Moreover, since constant α is far smaller than β, $\alpha(x_i||y_i)$ can be considered as noises added to the scaled base vector βz_i. Thus, vertex embedding vectors $o'(v_i)$ $(= \alpha(x_i||y_i)+\beta z_i)$ in Eq. (4) are still distributed close to a diagonal line (or hyperplane) in the first quadrant.

We would like to leave interesting topics of using other base vectors z_i (e.g., L_2-norm) as our future work.

Figure 7 shows the distributions of the updated 2D SPUR and SPAN vectors (from $o'(v_i)$), by adding a base vector z_i, where z_i is given by L_1-norm. Since we have $\alpha \ll \beta$ in Eq. (4), after adding



(a) $\alpha x_i + \beta(z_i[0], z_i[1])^T$ (b) $\alpha y_i + \beta(z_i[2], z_i[3])^T$

Figure 7: An example of $o'(v_i)$ distributions with L_1-norm diagonal-line base vector z_i ($\alpha = 0.01, \beta = 100$).

$\alpha(x_i||y_i)$ to the scaled base vector βz_i, the previous embedding vector $o(v_i)$ (as shown in Figure 6) is re-located close to diagonal lines in Figure 7. This way, we can reduce the number of vertex false alarms dominated by $o(q_i)$ in both SPUR and SPAN spaces.

5 DEGREE-AWARE STAR SUBSTRUCTURE SYNOPSES

In this section, we will build effective synopses over embeddings of star substructures of different degrees in dynamic graph G_D, which can be incrementally updated and support efficient DSM query processing.

5.1 Vertex Embedding via Degree Grouping

Note that, for the subgraph matching over a dynamic graph G_t, those vertices $v_i \in V(G_t)$ with high degrees $deg(v_i)$ are less likely to be pruned in the embedding space. This is because, for high degrees, the SPAN vectors in $o'(v_i)$ tend to have large values. Thus, $o'(v_i)$ is more likely to be dominated by query embedding vector $o'(q_i)$, and treated as candidate vertices for the refinement (i.e., v_i cannot be pruned).

To enhance the pruning power for high-degree vertices, in this subsection, we propose a novel and effective *degree grouping* technique, which separately maintains vertex embeddings, $o'(v_i)$, for star substructures s_{v_i} of different (lower) degree groups (instead of unit star subgraph g_{v_i}) in G_t.

Equi-Frequency Degree Grouping: One straightforward way to perform the degree grouping is to let each single degree value be a degree group. However, since vertex degrees in real-world graphs usually follow the power-law distribution [2, 5, 6, 16, 45], only a small fraction of vertices have high degrees. In other words, only a few unit star subgraphs g_{v_i} have star substructures s_{v_i} of very high degrees, which is not space- and time-efficient to maintain in a dynamic graph G_t.

Inspired by this, in this paper, we propose to maintain vertex embeddings for star substructures with m degree groups (of different degree intervals). We assume that the degree statistics of the initial graph G_0 are similar to that in G_t.

This way, we obtain the *probability mass function* of vertex degrees in G_0 is $pmf(\delta) = \frac{freq(deg(v_i) \geq \delta)}{|V(G_0)|}$, where $freq(deg(v_i) \geq \delta)$ is the number of vertices $v_i \in V(G_0)$ whose degrees are greater than or equal to a degree value δ. Our goal is to divide the degree interval, $[1, deg_{max}]$, into m degree groups $\mathcal{B} = \{B_1, B_2, \cdots, B_m\}$, such that each *degree bucket* B_j ($= (\delta_{j-1}, \delta_j]$) contains the same (or similar) mass in the distribution of $pmf(\delta)$ (for $\delta \in B_j$).

5.2 The Construction of Degree-Aware Star Substructure Synopses (DAS^3)

Based on the degree grouping, we will construct m *degree-aware star substructure synopses* (DAS^3), Syn_j (for $1 \leq j \leq m$), which are essentially m grid files [24], respectively, that store (aggregated) vertex embeddings for star substructures in specific degree groups.

Data Structure of DAS^3 Synopses: Specifically, each DAS^3 synopsis Syn_j partitions the embedding data space into cells, C, of equal size. Each grid cell C is associated with a list, $C.list$, of vertices v_i. Each vertex $v_i \in C.list$ contains the following aggregates:

- an *embedding upper bound* vector, $v_i.UB_\delta$, for all star substructures s_{v_i} with center vertex degrees $\delta \in (\delta_{j-1}, \delta_j]$, and;
- a list of MBRs, $v_i.MBR_\delta$ (for $\delta \in (\delta_{j-1}, \delta_j]$), that minimally bound all embedding vectors, $o'(s_{v_i})$, of star substructures s_{v_i} with degrees equal to δ.

To facilitate the access of cells in DAS^3 synopsis Syn_j efficiently, we sort all cells in descending order of their keys $C.key$. Here, for each cell C, we use $C.UB$ to denote its maximal corner point (i.e., taking the maximum value of the cell interval on each dimension). The key, $C.key$, is defined as $||C.UB||_2$, where $||Z||_2 = \sum_{i=1}^{d} Z[i]^2$ for a d-dimensional vector Z.

Note that, if a query vertex embedding $o(q_i)$ dominates $C.UB$, then the key $||o(q_i)||_2$ of query vertex embedding $o(q_i)$ must be smaller than or equal to the cell key $C.key$. Thus, with such a sorted list of cells, we can efficiently access all cells that are fully or partially dominated by a given query vertex embedding $o(q_i)$.

DAS^3 Synopsis Construction: Each DAS^3 synopsis Syn_j corresponds to a degree group B_j ($1 \leq j \leq m$) with degree interval $(\delta_{j-1}, \delta_j]$. We build the DAS^3 synopsis Syn_j as follows. For each vertex v_i with degree $deg(v_i) > \delta_{j-1}$ in G_t and for each degree value δ ($\in (\delta_{j-1}, \delta_j]$), we obtain all its star substructures, s_{v_i}, with degrees of center vertex v_i equal to δ, compute their embedding vectors $o'(s_{v_i})$, and use a *minimum bounding rectangle* (MBR), $v_i.MBR_\delta$, to minimally bound these embedding vectors.

Let $v_i.UB_\delta$ be the maximum corner point of the MBR $v_i.MBR_\delta$ (by taking the upper bound of the MBR on each dimension). Then, for vertex v_i, we insert vertex v_i into the vertex list, $C.list$, of a cell $C \in \mathcal{I}_j$, into which corner point $v_i.UB_{ub\_\delta}$ falls, where degree upper bound $ub\_\delta = \min\{deg(v_i), \delta_j\}$. Intuitively, this corner point $v_i.UB_{ub\_\delta}$ is the one most likely to be dominated by a query vertex embedding $o'(q_i)$ in this degree group B_j.

Moreover, we also associate corner point $v_i.UB_{ub\_\delta}$ with the list of MBRs, $v_i.MBR_\delta$ (for each $\delta \in (\delta_{j-1}, ub\_\delta]$), in synopsis Syn_j, which can be used for further pruning for a specific degree in interval $(\delta_{j-1}, \delta_j]$.

Discussions on MBR Computations and Incremental Maintenance: Note that, the MBRs, $v_i.MBR_\delta$, mentioned above minimally bound embedding vectors $o'(v_i)$ for all star substructures s_{v_i}. Since there are an exponential number of possible star structures (i.e., $2^{deg(v_i)}$), it is not efficient to enumerate all star substructures, compute their embedding vectors, and obtain the MBRs. Therefore, in

this paper, we propose a sorting-based method to obtain $v_i.MBR_\delta$ without enumerating all star substructures with degree δ.

Specifically, for each vertex v_i, we maintain d sorted lists, each of which, $v_i.spur\_list_k$ (for $1 \le k \le d$), contains the k-th elements, $y_j[k]$, of SPUR vectors y_j (obtained from v_i's 1-hop neighbors $v_j \in \mathcal{N}(v_i)$) in ascending order.

For a degree δ in the degree group $B_j = (\delta_{j-1}, \delta_j]$, we obtain lower and upper bounds of MBR $v_i.MBR_\delta$, that is, $v_i.MBR_\delta[2k]$ and $v_i.MBR_\delta[2k+1]$, respectively, as follows:

$$v_i.MBR_\delta[2k] = \sum_{deg \in [1,\delta]} v_i.spur\_list_k[deg], \quad (5)$$

$$v_i.MBR_\delta[2k+1] = \sum_{deg \in [deg(v_i)-\delta+1, deg(v_i)]} v_i.spur\_list_k[deg], \quad (6)$$

where $deg(v_i)$ is the degree of vertex v_i in G_t (or the size of the sorted list $v_i.spur\_list_k$).

Intuitively, Eqs. (5) and (6) give the lower/upper bounds of the MBR $v_i.MBR_\delta$ on the k-th dimension, by summing up the first δ and the last δ values, respectively, in the sorted list $v_i.spur\_list_k$.

The Time Complexity of the MBR Maintenance: The time complexity of sorting the k-th dimension of SPUR vectors (for $1 \le k \le d$) is given by $O(d \cdot deg(v_i) \cdot log(deg(v_i)))$, which is much less than that of enumerating all star substructures (i.e., $O(d \cdot 2^{deg(v_i)})$).

For the maintenance of a sorted list $v_i.spur\_list_k$ in dynamic graph G_t, upon edge insertion or deletion (or insertion/deletion of a 1-hop neighbor v_j), we can use a binary search to locate where we need to insert into (or remove from) $v_i.spur\_list_k$ elements $y_j[k]$ in the SPUR vector y_j. Therefore, the time complexity of incrementally maintaining d sorted lists $v_i.spur\_list_k$ is given by $O(d \cdot log_2(deg(v_i)))$.

6 DSM QUERY PROCESSING

In this section, we will illustrate query processing algorithms based on vertex dominance embeddings in DAS^3 synopses to efficiently answer DSM queries.

6.1 Synopsis Pruning Strategies

Given a query vertex q_i in the query graph q and SAS^3 synopses Syn_j, we would like to obtain candidate vertices v_i from synopsis Syn_j which may match with the query vertex q_i, where $deg(q_i)$ falls into the degree group $B_j = (\delta_{j-1}, \delta_j]$.

In this subsection, we present two effective pruning methods over index Syn_j, named *embedding dominance pruning* and *MBR range pruning*, which are used to rule out false alarms of cells/vertices.

Embedding Dominance Pruning: We first provide the embedding dominance pruning method, which filters out those cells/vertices in synopsis Syn_j that are not dominated by a given query embedding vector $o'(q_i)$.

Lemma 6.1. **(Embedding Dominance Pruning)** *Given a query embedding vector $o'(q_i)$ of the query vertex q_i, any cell C or vertex v_i can be safely pruned, if $o'(q_i)$ does not dominate any portion of cell C or embedding upper bound vector $v_i.UB_\delta$.*

Proof. If a query vertex q_i in query graph q matches with a data vertex v_i in a subgraph of the data graph, then their vertex dominance embeddings must hold that $o'(q_i) \preceq o'(v_i)$. Therefore, if $o'(q_i)$ does not dominate the cell C, then $o'(q_i)$ cannot dominate any vertex $o'(v_i)$ inside cell C, and all vertices in cell C (or cell C) can be safely pruned.

Moreover, if $o'(q_i)$ does not dominate embedding upper bound vector $v_i.UB_\delta$, i.e., $v_i.UB_\delta \notin DR(o'(q_i))$, then $o'(q_i)$ does not dominate any star substructure s_{v_i} with center vertex v_i and the corresponding degree group. In other words, q_i does not match with v_i. Thus, vertex v_i can be safely pruned. □

Algorithm 2: Initial DSM Answer Set Generation

Input: i) a snapshot graph G_t at current timestamp t; ii) a DAS^3 synopsis Syn_j on G_t, and; iii) a query graph q
Output: a set, $A(q,t)$, of initial subgraph matching results in G_t

```
compute vertex dominance embeddings o'(q_i) for all query vertices q_i ∈ V(q)
// traverse synopses Syn_j to obtain candidate vertices
for each query vertex q_i ∈ V(q) do
    obtain a synopsis Syn_j such that deg(q_i) ∈ (δ_{j-1}, δ_j]
    for each cell C ∈ Syn_j with C.key ≥ ||o'(q_i)||_2 do
        for each candidate vertex v_i ∈ C.list do
            if o'(q_i) ≼ v_i.UB_δ then
                if o'(q_i) ∈ v_i.MBR_{deg(q_i)} then
                    q_i.cand_set ← q_i.cand_set ∪ {v_i}
                        // Lemmas 6.1 and 6.2
// generate a query plan Q
select a query vertex q_i with the smallest candidate set size |q_i.cand_set| as the first query vertex in an ordered list Q
iteratively select a neighbor q_k of q_i ∈ Q with minimum |q_k.cand_set| and append q_k to Q, until all query vertices in V(q) have been added to Q
// assemble and refine candidate subgraphs
A(q,t) ← ∅
invoke Refinement(q, Q, G_t, A(q,t), ∅, 0) to obtain actual DSM query answers in A(q,t)
return A(q,t);
```

MBR Range Pruning: Next, we give an effective *MBR range pruning* method, which further utilizes the MBR ranges, $v_i.MBR_\delta$, of star substructure embeddings and prunes vertices v_i that do not fall into the MBRs.

Lemma 6.2. **(MBR Range Pruning)** *Given a query embedding $o'(q_i)$ of the query vertex q_i and a vertex v_i in a cell of DAS^3 synopsis Syn_j (for $deg(q_i) \in (\delta_{j-1}, \delta_j]$), vertex v_i can be safely pruned, if it holds that $o'(q_i) \notin v_i.MBR_{deg(q_i)}$.*

Proof. The MBR $v_i.MBR_{deg(q_i)}$ minimally bounds vertex embeddings for all possible star substructures s_{v_i} with center vertex v_i and degree $deg(q_i)$. Thus, if query vertex $q_i \in V(q)$ and its 1-hop neighbors match with some star substructures with the same degree $deg(q_i)$, then its query embedding vector $o'(q_i)$ must fall into this MBR $v_i.MBR_{deg(q_i)}$. Therefore, if this condition does not hold, i.e., $o'(q_i) \notin v_i.MBR_{deg(q_i)}$, then q_i does not match with v_i, and v_i can be safely pruned, which completes the proof. □

Algorithm 3: Refinement

Input: i) a query graph q; ii) a sorted list (query plan) Q; iii) a snapshot graph G_t; iv) matching results set A; v) a sorted list, M, of vertices matching with query vertices in Q, and; vi) the recursion depth n

Output: a set, A, of subgraph matching results

1 **if** $n = |Q|$ **then**
2 $A \leftarrow A \cup \{M\}$
3 **return**;
4 **else**
5 $S_{cand} = \emptyset$
6 **if** $n = 0$ **then**
7 $S_{cand} \leftarrow S_{cand} \cup Q[n].cand\_set$
8 **else**
9 **for** *each candidate vertex* $u \in Q[n].cand\_set$ *and* $u \notin M$ **do**
10 **if** *edges* $(M[i], u)$ *exist in* $E(G_t)$ *for all edges* $(Q[i], Q[n]) \in E(q)$ *(for* $0 \leq i < n$*)* **then**
11 $S_{cand} \leftarrow S_{cand} \cup \{u\}$
12 **for** *each candidate vertex* $u \in S_{cand}$ **do**
13 $M[n] = u$
14 Refinement $(q, Q, G_t, A, M, n+1)$;

6.2 Initial DSM Answer Set Generation

Algorithm 2 illustrates the algorithm of generating the initial subgraph matching answer set, $A(q,t)$, over a snapshot G_t of a dynamic subgraph G_D, by traversing synopses over vertex dominance embeddings. Specifically, given a query graph q, we first obtain vertex dominance embeddings $o'(q_i)$ for each query vertex $q_i \in V(q)$ (line 1), and then traverse DAS^3 synopses, Syn_j, to retrieve vertex candidate sets $q_i.cand\_set$ (lines 2-8). Next, we generate a query plan Q, which is an ordered list of connected query vertices that can be used to join their corresponding candidate vertices (lines 9-10). Finally, we assemble candidate vertices into candidate subgraphs, and refine candidate subgraphs by the *left-deep join based method* [37, 53] by invoking function Refinement(·), and return the initial subgraph matching answers in $A(q,t)$ (lines 11-13).

The Synopsis Search: For each query vertex $q_i \in V(q)$, we first need to find a synopsis Syn_j that matches the degree group of q_i (i.e., $deg(q_i) \in (\delta_{j-1}, \delta_j]$ must hold; line 3).

As mentioned in Section 5.2, cells, C, in the DAS^3 synopsis Syn_j are sorted in descending order of their keys $C.key$. We thus only need to search for those cells $C \in Syn_j$ satisfying the condition that $C.key \geq ||o'(q_i)||_2$ (line 4). Then, for each candidate vertex $v_i \in C.list$, we apply our proposed embedding dominance and MBR range pruning strategies (as discussed in Lemmas 6.1 and 6.2, respectively), and obtain the candidate vertex set, $q_i.cand\_set$, for the query vertex q_i (lines 5-8).

Query Plan Generation: After obtaining candidate vertices, we generate a query plan Q for refinement. Intuitively, we would like to reduce the size of intermediate join results. Therefore, we first initialize Q with a query vertex $q_i \in V(q)$ having the minimum candidate set size $|q_i.cand\_set|$ (line 9), and then iteratively add to Q the one, q_k, that is connected with those selected query vertices in Q and having the smallest $|q_k.cand\_set|$ value (line 10).

Refinement: Algorithm 3 uses a *left-deep join based method* [37, 53] to assemble candidate vertices in $q_i.cand\_set$ into candidate subgraphs, and obtain the actual matching subgraph answer set $A(q,t)$ (i.e., lines 11-13 of Algorithm 2).

Specifically, we maintain a vertex vector M that will store vertices of the subgraph g matching with ordered query vertices in Q. To enumerate all matching subgraphs, each time we recursively expand partial matching results by including a new candidate vertex u as $M[n]$ that maps with the n-th query vertex $Q[n]$. If we find all vertices in M mapping with Q (i.e., recursion depth $n = |Q|$), then we can add M to the answer set A (lines 1-3). Otherwise, we will find a vertex candidate set S_{cand} from $Q[n].cand\_set$, such that each vertex u in S_{cand} has the same edge connections to that in M as that in Q (lines 5-11). Next, for each vertex $u \in S_{cand}$, we treat it as vertex $M[n]$ matching with $Q[n]$, and recursively call function Refinement(·) with more depth $n+1$ (lines 13-17).

After the recursive function Refinement(·) has been executed, the answer set A (or $A(q,t)$ in Algorithm 3) will contain a set of actual subgraph matching results.

Complexity Analysis: In Algorithm 2, since we need to access each query vertex q_i and its 1-hop neighbors $\mathcal{N}_{v_i}$, the time complexity of computing vertex dominance embeddings $o'(q_i)$ (line 1) is given by $O(|V(q)| + |E(q)|)$.

For the synopsis traversal (lines 2-8), assume that PP_C is the pruning power of the cells' key value and PP_v is the pruning power in each cell's point list. Thus, the synopsis traversal cost is $O(|V(q)| \cdot (1 - PP_C)K^d \cdot (1 - PP_v)|C.list|)$, where K^d is the number of cells and $|C.list|$ is the number of vertices in the cell C.

Next, for the greedy-based query plan generation (lines 9-10), we need to iteratively select a neighbor of vertices in the query plan Q, which requires $O(|V(q)|^2)$ cost.

Finally, we invoke the recursive function Refinement(·) to find actual subgraph matching results (lines 11-13), with the worst-case time complexity $O(\prod_{i=0}^{|V(q)|-1} |q_i.cand\_set|)$.

Therefore, the overall time complexity of Algorithm 2 is given by: $O(|V(q)| + |E(q)| + |V(q)| \cdot (1 - PP_C)K^d \cdot (1 - PP_v)|C.list| + \prod_{i=0}^{|V(q)|-1} |q_i.cand\_set| + |V(q)|^2)$.

6.3 DSM Query Answering

Given a dynamic graph G_{t-1} at timestamp $(t-1)$, a registered query graph q, and a graph update operation ΔG_t at timestamp t, Algorithm 4 provides a DSM query answering algorithm, which maintains/obtains the latest subgraph matching answer set, $A(q,t)$.

Specifically, if the graph update operation ΔG_t is an insertion of edge $e = (v_i, v_j)$, we will calculate an incremental update, $\Delta A(q,t)$, of new subgraph matching results (lines 1-11).

Edge Insertion: For the insertion of an edge $e = (v_i, v_j)$, we will first find matching edge(s) (q_i, q_j) in the query graph with the same labels (line 3). Then, we will check whether or not edges (v_i, v_j) and (q_i, q_j) match each other, by applying the *embedding dominance* and *MBR range pruning* strategies (described in Section 6.1; lines 4-5). If the answer is yes, then we will generate a new query plan Q' starting from query vertices q_i and q_j (note: the remaining ones are obtained from Q, similar to line 10 of Algorithm 2; line 7), initialize the first two matching vertices v_i and v_j in a sorted list M' (lines 8-9), and invoke the function Refinement(·) with parameters of

Algorithm 4: The Query Answering Algorithm for Continuous Dynamic Subgraph Matching Query

Input: i) a dynamic graph G_{t-1} at previous timestamp $(t-1)$; ii) a graph update operation ΔG_t; iii) a registered query graph q, and; iv) matching result set $A(q, t-1)$ at timestamp $(t-1)$

Output: an updated set, $A(q, t)$, of subgraph matching results at timestamp t

```
// edge insertion operation ΔG_t = (e+, t)
1  if ΔG_t = (e+, t) for edge e = (v_i, v_j) then
2      ΔA(q, t) ← ∅
3      for each query edge (q_i, q_j) ∈ E(q) with labels matching with
         that of (v_i, v_j) do
           // check whether (v_i, v_j) matches (q_i, q_j)
4          if o'(q_i) ≤ o'(v_i) and o'(q_j) ≤ o'(v_j) then
5              if o'(q_i) ∈ v_i.MBR_deg(q_i) and
                 o'(q_j) ∈ v_j.MBR_deg(q_j) then
6                  // Lemmas 6.1 and 6.2
7                  generate a query plan Q' by using q_i and q_j as
                     the first two query vertices and the remaining
                     ones from Q
8                  M'[0] = v_i
9                  M'[1] = v_j
10                 invoke Refinement(q, Q', G_t, ΔA(q, t), M', 2) to
                     incrementally obtain new matching results in
                     ΔA(q, t)
11     A(q, t) ← A(q, t-1) ∪ ΔA(q, t)
12 else
       // edge deletion operation ΔG_t = (e−, t) for edge
          e = (v_i, v_j)
13     A(q, t) = A(q, t-1)
14     for each subgraph answer g ∈ A(q, t-1) do
15         if edge (v_i, v_j) exists in E(g) then
16             A(q, t) ← A(q, t) − {g}
17 return A(q, t)
```

new query plan Q', incremental answer set $\Delta A(q, t)$, the initialized sorted list M', and recursion depth $n = 2$ (as two matching vertices have been found; line 10). After that, we can update the answer set $A(q, t)$ with $A(q, t-1) \cup \Delta A(q, t)$ (line 11).

Edge Deletion: When ΔG_t is an edge deletion operation $(e-, t)$, we can simply remove those existing subgraph matching answers $g \in A(q, t-1)$ which contain the deleted edge e, and obtain the updated answer set $A(q, t)$ (lines 12-16).

Finally, we return the latest DSM query answer set $A(q, t)$ at timestamp t (line 17).

Complexity Analysis: In Algorithm 4, for the edge insertion operation, we check all query edges, generate a new query plan Q', and refine new candidate subgraphs with the new edge (lines 1-11). Therefore, the worst-case time complexity is given by $O((|V(q)|^2 + \prod_{i=2}^{|V(q)|-1} |Q'[i].cand\_set|) \cdot |E(q)|)$.

For the edge deletion operation (lines 12-16), since we delete matching results from $A(q, t-1)$ that contain the deleted edge, by using the hash file to check the edge existence with $O(1)$ cost, the time complexity is given by $O(|A(q, t-1)|)$.

7 COST-MODEL-GUIDED EMBEDDING OPTIMIZATION

In this section, we will design a novel cost model for evaluating the pruning power of our proposed vertex dominance embedding technique. Then, we will utilize this cost model to guide the embeddings and achieve better performance of dynamic subgraph matching.

7.1 Cost Model for Dynamic Subgraph Matching via Vertex Embeddings

In this subsection, we will propose a cost model to evaluate the performance of our dynamic subgraph matching (or the pruning power of using vertex dominance embeddings).

Specifically, we estimate the number, $Cost_{DSM}$, of candidate vertices v_i (to retrieve and refine) whose embedding vectors $o'(v_i)$ are dominated by query embedding vector $o'(q_i)$ (given by Lemma 6.1). That is, we have:

$$Cost_{DSM} = \sum_{v_i \in V(G_t)} \prod_{j=1}^{2d} Pr\{o'(q_i)[j] \le o'(v_i)[j]\}, \quad (7)$$

where $2d$ is the dimension of the embedding vector $o'(\cdot)$, and $Pr\{\cdot\}$ is a probability function.

Analysis of the Cost Model: We consider $o'(v_i)[j]$ as a random number generated from a random variable, with mean $\mu_{o'(v_i)[j]}$ and variance $\sigma^2_{o'(v_i)[j]}$. Moreover, $o'(q_i)[j]$ can be considered as a constant. We have the following equation:

$$\begin{aligned} & Pr\{o'(q_i)[j] \le o'(v_i)[j]\} \\ = \; & Pr\left\{ \frac{(o'(q_i)[j] - o'(v_i)[j]) - (o'(q_i)[j] - \mu_{o'(v_i)[j]})}{\sigma^2_{o'(v_i)[j]}} \right. \\ & \left. \le \frac{-(o'(q_i)[j] - \mu_{o'(v_i)[j]})}{\sigma^2_{o'(v_i)[j]}} \right\}. \end{aligned} \quad (8)$$

By applying *Central Limit Theorem* (CLT) [69] to Eq. (8), we have:

$$Pr\{o'(q_i)[j] \le o'(v_i)[j]\} \approx \Phi\left(\frac{\mu_{o'(v_i)[j]} - o'(q_i)[j]}{\sigma^2_{o'(v_i)[j]}} \right), \quad (9)$$

where $\Phi(\cdot)$ is the *cumulative density function* (cdf) of *standard normal distribution*.

We substitute Eq. (9) into Eq. (7) and obtain:

$$Cost_{DSM} = |V(G_t)| \cdot \prod_{j=1}^{2d} \Phi\left(\frac{\mu_{o'(v_i)[j]} - o'(q_i)[j]}{\sigma^2_{o'(v_i)[j]}} \right). \quad (10)$$

7.2 Design of Cost-Model-Based Vertex Dominance Embeddings

Eq. (10) estimates the (worst-case) query cost for one query vertex q_i during the dynamic subgraph matching. Intuitively, our goal is to design vertex embeddings $o'(\cdot)$ that minimizes the cost model $Cost_{DSM}$ (given in Eq. (10)).

Note that, $\Phi(\cdot)$ in Eq. (10) is a monotonically increasing function. Therefore, if we can minimize the term $\frac{\mu_{o'(v_i)[j]} - o'(q_i)[j]}{\sigma^2_{o'(v_i)[j]}}$ in Eq. (10), then low query cost $Cost_{DSM}$ can be achieved. In other words,

guided by our proposed cost model (to minimize the query cost $Cost_{DSM}$), **our target is to design/select a "good" distribution of vertex dominance embeddings $o'(v_i)$ with:**

(1) low mean $\mu_{o'(v_i)[j]}$, and;

(2) high variance $\sigma^2_{o'(v_i)[j]}$.

In this paper, unlike standard vertex dominance embeddings generated by *Uniform* random function $f_1(\cdot)$ (as discussed in Section 4.2), we choose to use a (seeded) *Zipf* random function to produce SPUR vectors x_i in vertex embeddings $o'(v_i)$ (or in turn generate SPAN vectors $y_i = \sum x_j$). The $Zipf$ distribution exactly follows our target of finding a random variable with low mean and high variance, which can achieve low query cost, as guided by our proposed cost model $Cost_{DSM}$ (given in Eq. (7)). We would like to leave the interesting topic of studying other low-mean/high-variance distributions as our future work.

Discussions on the Seeded Zipf Generator: We consider two distributions, $Uniform$ and $Zipf$, each of which is divided to b buckets with the same area. This way, we create a 1-to-1 mapping between buckets in a Uniform distribution and that in a $Zipf$ distribution. Given a seeded pseudo-random number, r, from the Uniform distribution, we can first find the bucket this Uniform random number falls into, obtain its corresponding bucket in the $Zipf$ distribution, and compute its proportional location r' in the $Zipf$ bucket. As a result, r' is the random number that follows the $Zipf$ distribution.

Integration of the Cost-Model-Based Vertex Embeddings into our Dynamic Subgraph Matching Framework: In light of our cost model above, we design a novel cost-model-based vertex dominance embedding, denoted as $o'_C(v_i)$, for each vertex v_i, that is:

$$o'_C(v_i) = \alpha(x'_i || y'_i) + \beta z_i, \tag{11}$$

where $x'_i = f_Z(l(v_i))$ is the newly designed SPUR vector generated by a seeded $Zipf$ random generator $f_Z(\cdot)$, and $y'_i = \sum_{\forall v_j \in \mathcal{N}_{v_i}} x'_j$.

Note that, since the SPAN vector y'_i in Eq. (11) is given by $\sum_{\forall v_j \in \mathcal{N}_{v_i}} x'_j$, its distribution still follows the property of low mean and high variance to achieve low query cost.

8 EXPERIMENTAL EVALUATION

8.1 Experimental Settings

To evaluate the performance of our DSM approach, we conduct experiments on a Ubuntu server equipped with an Intel Core i9-12900K CPU and 128GB memory. Our source code in C++ and real/synthetic graph data sets are available at URL: *https://github.com/JamesWhiteSnow/DSM*.

Real/Synthetic Graph Data Sets: We evaluate our DSM approach over both real and synthetic graphs.

Real-world graphs. We test five real-world graph data used by previous works [8, 9, 26, 27, 30, 39, 41, 51, 53, 58, 60, 73], which can be classified into three categories: i) biology networks (Yeast and HPRD); ii) bibliographical/social networks (DBLP and Youtube), and; iii) citation networks (US Patents). Statistics of these real graphs are summarized in Table 2.

Synthetic graphs. We generate synthetic graphs via NetworkX [25], and produce small-world graphs following the Newman-Watts-Strogatz model [68]. Parameter settings of synthetic graphs are depicted in Table 3. For each vertex v_i, we generate its label $l(v_i)$ by randomly picking up an integer within $[1, |\sum|]$, following the $Uniform$, *Gaussian*, or $Zipf$ distribution. Accordingly, we obtain 3 types of graphs, denoted as *Syn-Uni*, *Syn-Gau*, and *Syn-Zipf*, resp.

Table 2: Statistics of real-world graph data sets.

Data Sets	$\|V(G)\|$	$\|E(G)\|$	$\|\sum\|$	$avg\_deg(G)$
Yeast (ye)	3,112	12,519	71	8.0
HPRD (hp)	9,460	34,998	307	7.4
DBLP (db)	317,080	1,049,866	15	6.6
Youtube (yt)	1,134,890	2,987,624	25	5.3
US Patents (up)	3,774,768	16,518,947	20	8.8

Table 3: Parameter settings.

Parameters	Values
the dimension, d, of the SPUR/SPAN vector	1 **2**, 3, 4
the ratio, β/α	10, 100, **1,000**, 10,000, 100,000
the number, m, of degree groups	1, 2, **3**, 4, 5
the number, K, of cell intervals in each dimension of Syn_j	1, 2, **5**, 8, 10
the number, $\|\Sigma\|$, of distinct labels	5, 10, **15**, 20, 25
the average degree, $avg\_deg(q)$, of the query graph q	2, **3**, 4
the size, $\|V(q)\|$, of the query graph q	5, 6, **8**, 10, 12
the average degree, $avg\_deg(G_D)$, of dynamic graph G_D	3, 4, **5**, 6, 7
the size, $\|V(G_D)\|$, of the dynamic data graph G_D	10K, 30K, **50K**, 80K, 100K, 500K, 1M

The insertion (or deletion) rate is defined as the ratio of the number of edge insertions (or deletions) to the total number of edges in the raw graph data. Following the literature [14, 40, 43, 59], we set the insertion rate to 10% by default (i.e., the initial graph G_0 contains 90% edges, and the remaining 10% edges arrive as the insertion stream). Similarly, for the case of edge deletions, we set the default deletion rate to 10% (i.e., initial graph G_0 contains all edges, and deletion operations of 10% edges arrive in the stream).

Query Graphs: Similar to previous works [3, 8, 9, 26, 27, 39, 51, 59, 60], for each graph G_D, we randomly extract/sample 100 connected subgraphs as query graphs, where parameters of query graphs q (e.g., $|V(q)|$ and $avg\_deg(q)$) are depicted in Table 3.

Baseline Methods: We compare our DSM approaches (using the cost-model-based vertex dominance embeddings $o'_C(v_i)$, as discussed in Section 7) with five representative baseline methods of dynamic subgraph matching as follows:

(1) **Graphflow (GF)** [37] is a direct-incremental algorithm that enumerates updates of matching results without any auxiliary data structure.

(2) **SJ-Tree (SJ)** [14] is an index-based incremental method that evaluates the join query with binary joins using the index.

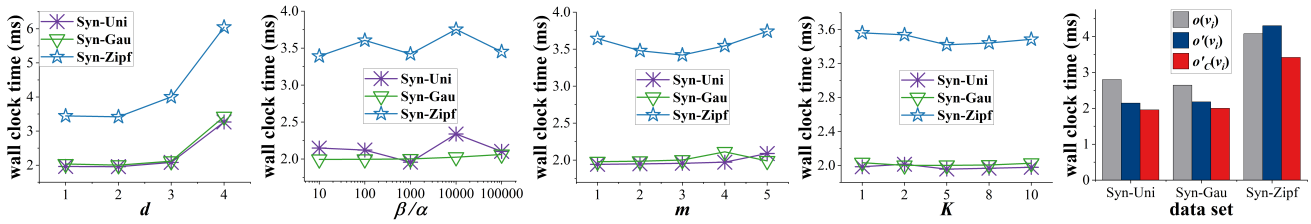
(3) **TurboFlux (TF)** [40] is an index-based incremental method that stores matches of paths in q without materialization and evaluates the query with the vertex-at-a-time method.

(4) **SymBi (Sym)** [43] is an index-based incremental method that prunes candidate vertex sets using all query edges.

(5) **IEDyn (IED)** [33, 34] is an index-based incremental method that supports acyclic queries and can achieve constant delay enumeration under the setting of graph homomorphism.

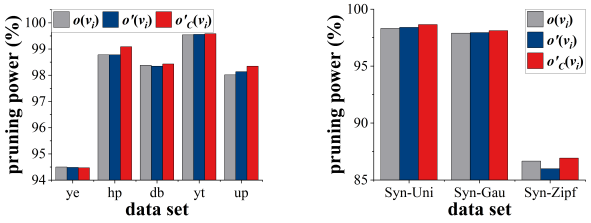
We used the code of baseline methods from [59], which is implemented in C++ for a fair comparison.

Evaluation Metrics: In our experiments, we report the efficiency of our DSM approach and baseline methods, in terms of the *wall clock time*, which is the time cost of processing the DSM query over the entire graph update sequence (i.e., continuous subgraph matching



(a) DSM vs. d (b) DSM vs. β/α (c) DSM vs. m (d) DSM vs. K (e) DSM vs. embeddings

Figure 8: The DSM efficiency w.r.t parameters d, β/α, m, K, and embedding design strategies.



(a) real-world graphs (b) synthetic graphs

Figure 9: The DSM pruning power w.r.t different design strategies of vertex dominance embeddings.

answer set monitoring). In particular, this time cost includes the time of filtering, refinement, embedding updates, and synopsis updates for each DSM query. We also evaluate the *pruning power* of our *embedding dominance pruning* and *MBR range pruning strategies* (as mentioned in Section 6.1), which is defined as the percentage of vertices that can be ruled out by our pruning methods. For all the experiments, we take an average of each metric over 100 runs (w.r.t. 100 query graphs, resp.).

Table 3 depicts parameter settings in our experiments, where default parameter values are in bold. For each set of experiments, we vary the value of one parameter while setting other parameters to their default values.

8.2 Parameter Tuning

In this subsection, we tune parameters for our DSM approach over synthetic graphs.

The DSM Efficiency w.r.t. SPUR/SPAN Vector Dimension d: Figure 8(a) illustrates the DSM performance by varying the dimension, d, of the cost-model-based SPUR/SPAN vector (in $o'_C(v_i)$ of Eq. (11)) from 1 to 4, where other parameters are set to their default values. With higher embedding dimension d, the pruning power of our proposed pruning strategies in higher dimensional space increases. However, the access of synopses with larger d may also incur higher costs due to the "dimensionality curse" [7]. Thus, in this figure, for larger d, the wall clock time of DSM first decreases and then increases over all synthetic graphs. Nonetheless, the time cost remains low (i.e., 1.96 ~ 6.05 *ms*) for different d values.

The DSM Efficiency w.r.t. β/α Ratio: Figure 8(b) varies the ratio, β/α, from 10 to 100,000 for the optimized (or cost-model-based) vertex dominance embeddings $o'(v_i)$ (or $o'_C(v_i)$), where other parameters are set by default. In the figure, we can see that our DSM approaches are not very sensitive to the ratio β/α. For different β/α ratios, the query cost remains low (i.e., 1.96 ~ 3.75 *ms*).

The DSM Efficiency w.r.t. # of Degree Groups, m: Figure 8(c) reports the performance of our DSM approach, by varying the number, m, of degree groups from 1 to 5, where other parameters are set by default. In this figure, the time costs over *Syn-Uni* and *Syn-Zipf* first decrease and then increase when m increases, and there are some fluctuations for *Syn-Gau* (e.g., for $m = 4$ or 5). For all m values, the time cost remains low (i.e., 1.94 ~ 3.74 *ms*).

The DSM Efficiency w.r.t. # of Cell Intervals on Each Dimension, K: Figure 8(d) evaluates the effect of the number, K, of cell intervals on each dimension on the DSM performance, where K varies from 1 to 10, and other parameters are set by default. When K becomes larger, more vertices in synopsis cells can be pruned, however, more cells need to be accessed. Therefore, for DSM, with the increase of K, the time cost first decreases and then increases. Nonetheless, for different K values, the query cost remains low (i.e., 1.96 ~ 3.54 *ms*).

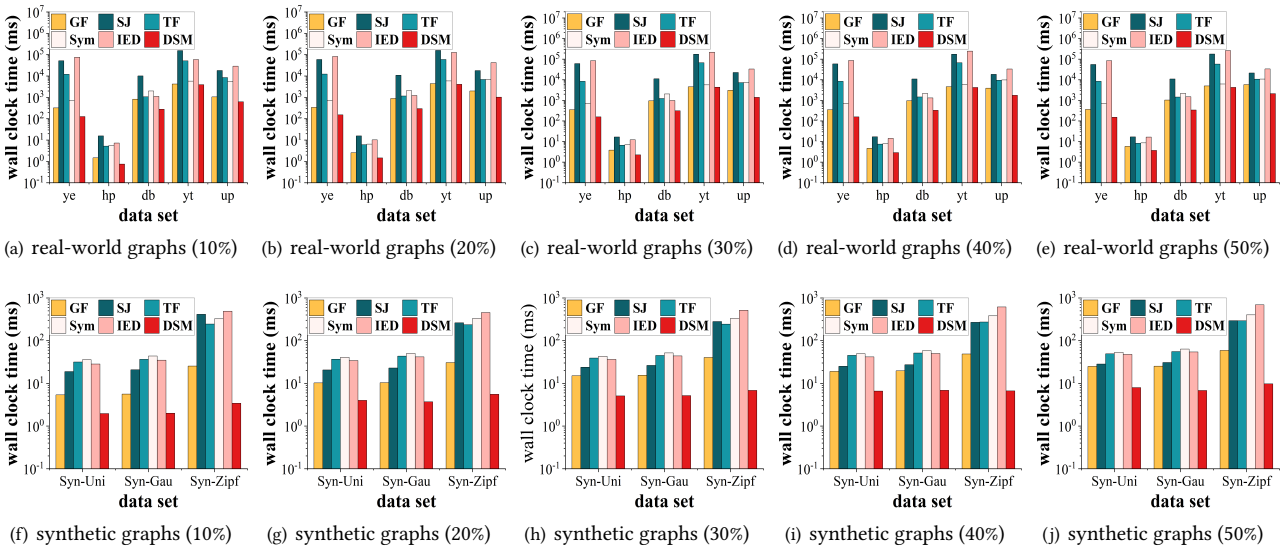
The DSM Efficiency w.r.t. Vertex Dominance Embedding Design Strategy: Figure 8(e) tests the DSM performance with different designs of vertex dominance embeddings, $o(v_i)$ (in Eq. (3)), $o'(v_i)$ (in Eq. (4)), and $o'_C(v_i)$ (in Eq. (11)), where default values are used for parameters. In the figure, we can see that, the optimized vertex embeddings $o'(v_i)$ (via the base vector z_i) incur smaller time cost than $o(v_i)$ in all cases, whereas the cost-model-based vertex embeddings $o'_C(v_i)$ consistently achieve the lowest time. For different vertex embeddings, the query cost remains low (i.e., 1.96~4.3*ms*).

The experimental results on real-world graphs are similar and thus omitted here. In subsequent experiments, we will set parameters $d = 2$, $\beta/\alpha = 1,000$, $m = 3$, and $K = 5$, and use the cost-model-based vertex embeddings $o'_C(v_i)$ (given by Eq. (11)).

8.3 The DSM Effectiveness Evaluation

In this subsection, we report the pruning power of our proposed pruning strategies (as discussed in Section 6.1) for dynamic subgraph matching queries over real/synthetic graphs.

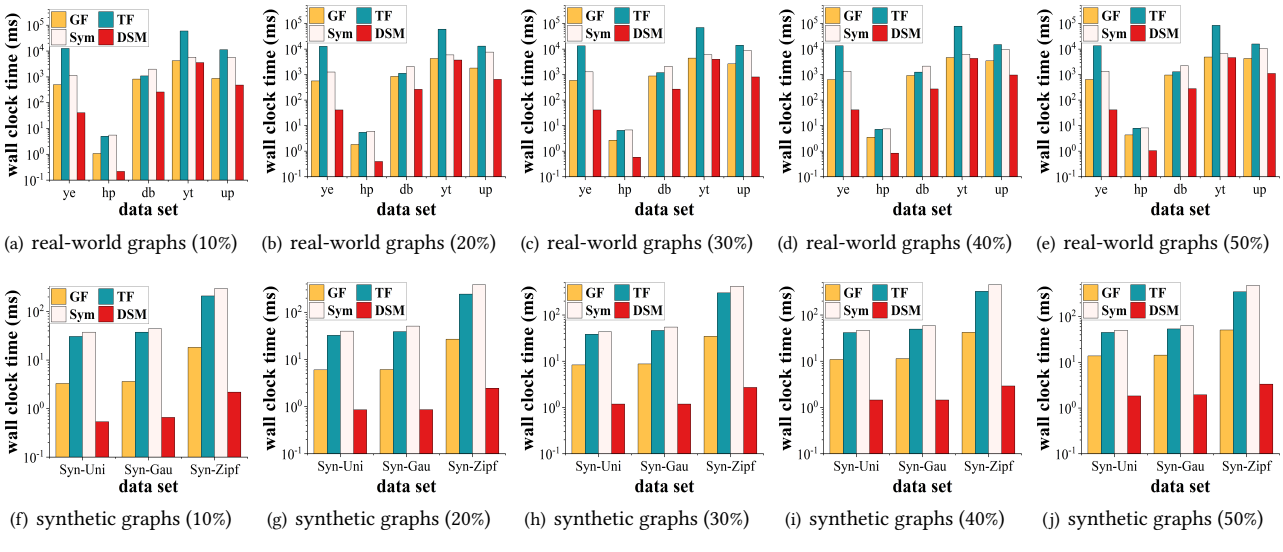
The DSM Pruning Power on Real/Synthetic Graphs: Figure 9 shows the pruning power of our DSM approach over real/synthetic graphs with different embedding designs $o(v_i)$, $o'(v_i)$, and $o'_C(v_i)$, where all parameters are set by default. In figures, we can see that for all real/synthetic graphs, the pruning power of our proposed vertex embedding designs can reach as high as 94.47% ~ 99.58% for real-world graphs and 85.99% ~ 98.65% for synthetic graphs,



(a) real-world graphs (10%) (b) real-world graphs (20%) (c) real-world graphs (30%) (d) real-world graphs (40%) (e) real-world graphs (50%)

(f) synthetic graphs (10%) (g) synthetic graphs (20%) (h) synthetic graphs (30%) (i) synthetic graphs (40%) (j) synthetic graphs (50%)

Figure 10: The DSM efficiency on edge-insertion-only real/synthetic graphs, compared with baseline methods.



(a) real-world graphs (10%) (b) real-world graphs (20%) (c) real-world graphs (30%) (d) real-world graphs (40%) (e) real-world graphs (50%)

(f) synthetic graphs (10%) (g) synthetic graphs (20%) (h) synthetic graphs (30%) (i) synthetic graphs (40%) (j) synthetic graphs (50%)

Figure 11: The DSM efficiency on edge-deletion-only real/synthetic graphs, compared with baseline methods.

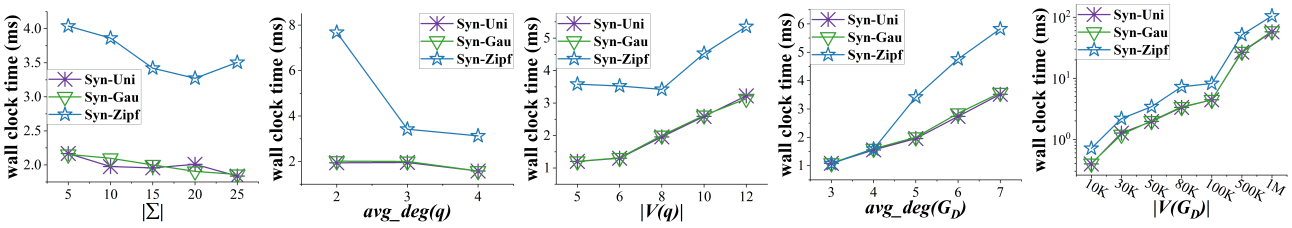
which confirms the effectiveness of our embedding-based pruning strategies. Our cost-model-based vertex embedding strategy $o'_C(v_i)$ always achieves the highest pruning power (i.e., 86.93% ~ 99.58%).

8.4 The DSM Efficiency Evaluation

In this subsection, we report the efficiency of our DSM approaches over real/synthetic graphs.

The DSM Efficiency on Edge-Insertion-Only Real/Synthetic Graphs: Figure 10 compares the efficiency of our DSM approach with that of 5 state-of-the-art baseline methods by varying the insertion ratio from 10% to 50% over both real and synthetic graphs, where all other parameters are set to default values. In Figures 10(a) and 10(f) with the default 10% insertion ratio, we can see that our DSM approach outperforms baseline methods mostly by 1-2 orders of magnitude. Overall, for all graphs (even for *up* with $3.77M$ vertices and $1.65M$ edge insertions), the time cost of our DSM approach remains low (i.e., <3.91 *sec*).

For other subfigures with different insertion ratios from 20% to 50%, we can see similar experimental results over both real and synthetic graphs. Although the total query time will increase as the number of inserted edges increases, our DSM approach can achieve performance that is up to 1-2 orders of magnitude than baseline methods. In summary, for all insertion ratios and graphs (even for



(a) DSM vs. $|\Sigma|$ (b) DSM vs. $avg\_deg(q)$ (c) DSM vs. $|V(q)|$ (d) DSM vs. $avg\_deg(G_D)$ (e) DSM vs. $|V(G_D)|$

Figure 12: The DSM efficiency w.r.t. parameters $|\Sigma|$, $avg\_deg(q)$, $|V(q)|$, $avg\_deg(G_D)$, and $|V(G_D)|$.

up with 3.77M vertices and 8.26M edge insertions), the time cost of our DSM approach remains low (i.e., < 4.37 *sec*).

The DSM Efficiency on Edge-Deletion-Only Real/Synthetic Graphs: Since SJ and IED baselines do not support the edge deletion [59], Figure 11 only compares the efficiency of our DSM approach with that of GF, TF, and SYM over real/synthetic graphs, where the deletion ratio varies from 10% to 50% and all other parameters are set to their default values. From Figures 11(a) and 11(f) with the default 10% deletion ratio, our DSM approach can always outperform the three baseline methods. Overall, for all real/synthetic data sets (even for *up* with 1.65M edge deletions), the wall clock time of our DSM approach remains low (i.e., < 3.55 *sec*).

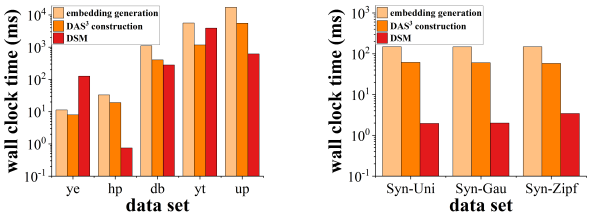
From other subfigures with different deletion ratios 20% ∼ 50%, we can find similar experimental results over both real and synthetic graphs, where our DSM approach can always outperform the three baseline methods. For different edge deletion ratios and graph sizes (even for *up* with 3.77M vertices and 8.26M edge deletions), the time cost of our DSM approach remains low (i.e., < 4.62 *sec*).

To evaluate the robustness of our DSM approaches, in the sequel, we vary the parameter values on synthetic graphs (e.g., $|\Sigma|$, $avg\_deg(q)$, $|V(q)|$, $avg\_deg(G_D)$, and $|V(G_D)|$). To better illustrate the trends of curves, we omit the results of the baselines below.

The DSM Efficiency w.r.t. # of Distinct Vertex Labels, $|\Sigma|$: Figure 12(a) shows the wall clock time of our DSM approach, where $|\Sigma|$ varies from 5 to 25, and other parameters are set to default values. When the number, $|\Sigma|$, of distinct vertex labels increases, the pruning power also increases (i.e., with fewer candidate vertices). Moreover, the query cost is also affected by vertex label distributions. Overall, the DSM query cost remains low for different $|\Sigma|$ values (i.e., 1.83 ∼ 4.04 *ms*).

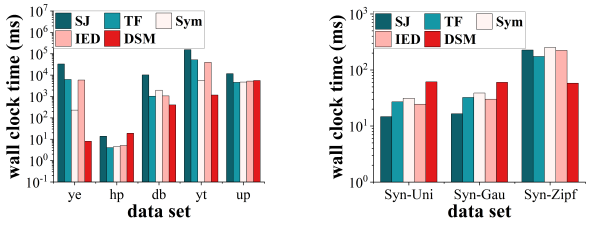
The DSM Efficiency w.r.t. Average Degree, $avg\_deg(q)$, of the Query Graph q: Figure 12(b) examines the DSM performance by varying the average degree, $avg\_deg(q)$, of the query graph q from 2 to 4, where other parameters are set to default values. Higher degree $avg\_deg(q)$ of q incurs higher pruning power of query vertices. Therefore, when $avg\_deg(q)$ increases, the DSM time cost decreases. For different $avg\_deg(q)$ values, the DSM query cost remains low (i.e., 1.58 ∼ 7.68 *ms*).

The DSM Efficiency w.r.t. Query Graph Size $|V(q)|$: Figure 12(c) illustrates our DSM performance, by varying the query graph size, $|V(q)|$, from 5 to 12, where default values are used for other parameters. When the number, $|V(q)|$, of vertices in query graph q increases, fewer candidate subgraphs are expected to match with larger query graph q. On the other hand, larger query graph size



(a) real-world graphs (b) synthetic graphs

Figure 13: The comparison analysis of the DSM offline precomputation and online query costs on real/synthetic graphs.



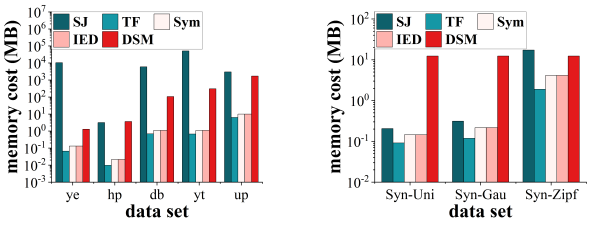
(a) real-world graphs (b) synthetic graphs

Figure 14: DAS^3 construction time on real/synthetic graphs, compared with baseline methods.

$|V(q)|$ will cause higher query costs in finding candidates for more query vertices, through synopsis traversal and refinement. Thus, the query time is influenced by these two factors. Nevertheless, the time cost remains low for different query graph sizes (i.e., <5.36*ms*).

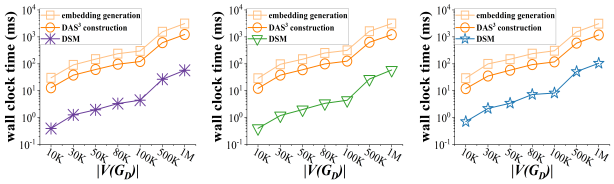
The DSM Efficiency w.r.t. Avg. Degree, $avg\_deg(G_D)$, of Dynamic Graph G_D: Figure 12(d) presents our DSM performance with different average degrees, avg_deg (G_D), of dynamic graph G_D, where $avg\_deg(G_D)$ = 3 ∼ 7, and default values are used for other parameters. Intuitively, higher degree $avg\_deg(G_D)$ in data graph G_D incurs lower pruning power and more candidate vertices. Thus, when $avg\_deg(G_D)$ becomes higher, the wall clock time also increases, especially for *Syn-Zipf* (due to its skewed vertex label distribution). Nevertheless, the DSM query time remains small for different $avg\_deg(G_D)$ values (i.e., 1.08 ∼ 5.82 *ms*).

The DSM Scalability Test w.r.t. Dynamic Graph Size $|V(G_D)|$: Figure 12(e) tests the scalability of our DSM approach with different dynamic graph sizes, $|V(G_D)|$, from 10K to 1M, where default



(a) real-world graphs (b) synthetic graphs

Figure 15: DAS^3 memory cost on real/synthetic graphs, compared with baseline methods.



(a) *Syn-Uni* (b) *Syn-Gau* (c) *Syn-Zipf*

Figure 16: The DSM pre-computation and query costs w.r.t data graph size $|V(G_D)|$.

values are assigned to other parameters. A larger dynamic graph incurs more matching candidate vertices (and, in turn, candidate subgraphs). From this figure, the time cost of DSM increases linearly with the increase of graph size $|V(G_D)|$, nonetheless, remains low (i.e., 0.4 ∼ 104.62 *ms*), which confirms the scalability of our DSM approach for large graph sizes.

8.5 DAS^3 Synopsis Initialization Cost

The DAS^3 Synopsis Initialization Cost on Real/Synthetic Graphs. We compare the DAS^3 synopsis initialization cost of our DSM approach (including time costs of vertex dominance embedding generation and DAS^3 construction over vertex dominance embeddings) with online query time over real/synthetic graphs, where parameters are set to default values. In Figure 13, for graph size from 3K to 3.77M, the overall offline pre-computation time varies from 19 *ms* ∼ 23 *sec*. Specifically, the time costs of embedding generation and DAS^3 construction are 11 *ms* ∼ 17 *sec*, 8 *ms* ∼ 6 *sec*, respectively. On the other hand, since the online query time includes the time cost of DAS^3 and embedding update, the maintenance cost of our DAS^3 is low.

The DAS^3 Index Time/Space Cost on Real/Synthetic Graphs. Since GF directly enumerates matching answers over the original data graph without any auxiliary data structure [37, 59], Figures 14 and 15 compare the construction time and storage cost of our DAS^3 with the other 4 baselines, where parameters are set to their default values. From Figure 14, we can find that our DAS^3 needs shorter construction time over some complex graphs, i.e., ye, db, yt, and *Syn-Zipf*. This is because, different from baseline methods that build auxiliary data structure over initial query answers, our DAS^3 is built over the initial data graph without enumerating the initial query answers and only depends on the data graph size. As shown in Figure 15, except SJ with exponential space cost [14, 59], our DAS^3 needs more storage costs for storing/indexing vertex dominance embeddings. However, different from baseline methods that construct an index for each query graph during the online dynamic subgraph matching process, our DAS^3 construction is *offline* and *one-time only*. Thus, our offline constructed DAS^3 can be used to accelerate numerous online dynamic subgraph matching requests from users simultaneously with high throughput.

The DAS^3 Synopsis Initialization Cost and Online Query Costs w.r.t. Data Graph Size $|V(G_D)|$. Figure 16(a) evaluates the DAS^3 synopsis initialization cost of our DSM approach, including time costs of the vertex dominance embedding generation and DAS^3 construction over embeddings, compared with online DSM query time, over synthetic graphs *Syn-Uni*, where we vary the graph size $|V(G)|$ from 10K to 1M and other parameters are set to default values. Specifically, for graph sizes from 10K to 1M, the time costs of the embedding generation and DAS^3 construction are 0.03∼3.11 *sec* and 0.01∼1.22 *sec*, respectively. The overall offline pre-computation time varies from 0.04 *sec* to 4.33 *sec*, and the dynamic subgraph matching query cost is much smaller (i.e., 0.4∼57.49 *ms*).

In Figures 16(b) and 16(c), we can see similar experimental results over *Syn-Gau* and *Syn-Zipf* graphs, respectively, where the subgraph matching query cost is much smaller than the overall offline pre-computation time.

9 RELATED WORK

Dynamic Subgraph Matching: Existing works on exact dynamic subgraph matching [14, 20, 33, 34, 37, 40, 43] can be classified into three categories: the *recomputation-based* [20], *direct-incremental* [37], and *index-based incremental algorithms* [14, 33, 34, 40, 43]. For the recomputation-based category, IncIsoMatch [20] extracted a snapshot subgraph $g_S \subseteq G_t$ surrounding the updated edges at the timestamp t, conducted the subgraph matching over this snapshot subgraph g_S (w.r.t. query graph q), and updated the answer set $A(q, t-1)$ at timestamp $(t-1)$ with new subgraph matching answers by removing redundant subgraphs. For the direct-incremental approach, GraphFlow [37] computed incremental results over the dynamic graph by using a multi-way join from the updated edge.

Moreover, index-based incremental methods construct an index over the initial subgraph matching answers, and obtain incremental results from the index (instead of the data graph). For example, SJ-Tree [14] modeled a dynamic subgraph query as a multi-way join and organized all partial join results in an index to serve the query. TurboFlux [40] built a tree index in which each node contains candidates of a query vertex, and dynamically maintained the index to keep the consistency with each snapshot of the dynamic graph. SymBi [43] improved the pruning power by constructing a graph index and designing an adaptive ordering method. IEDyn [33, 34] maintained an index for candidate vertex sets and their connection edges, which can achieve constant delay enumeration for acyclic queries with graph homomorphism. In contrast, our work designs a novel and effective *vertex dominance embedding* technique for candidate vertex/subgraph retrieval, which can effectively prune vertex/subgraph candidates and improve query efficiency.

In addition, there are also some studies that consider specific query graph topologies, such as paths [47, 57], cycles [49], and

cliques [44]. Due to the high computation cost of finding exact matching results, several approximate algorithms [12, 18, 20, 31, 54] have been proposed. Different from these works, our paper focuses on the exact subgraph matching query over the dynamic graph, given general query graphs (i.e., with arbitrary query graph structures).

Graph Embeddings: Prior works on graph embedding usually transform graphs or nodes to d-dimensional embedding vectors for different downstream tasks, such as graph classification, vertex classification, link prediction, and so on. Several graph embedding methods used heuristic rules to encode a graph or its nodes, such as DeepWalk [48], LINE [62], SDNE [63], Node2vec [23], and Struc2vec [52]. These methods are designed for static graphs, which generate graph or node embeddings for fixed graph structures. Thus, we cannot directly use them for a dynamic graph task.

Recently, the graph representation learning via *graph neural networks* (GNNs) has exhibited success in various graph-related applications [28, 32, 55, 56, 64–66]. With different network architectures, some previous works [4, 17, 42, 71] proposed to use GNNs to generate graph embeddings for the graph matching. However, these works either cannot guarantee the accuracy of tasks over unseen test graph data [4, 42] (due to the limitation of neural networks), or cannot efficiently and incrementally maintain embeddings in dynamic graphs with continuous updates [17, 71]. In contrast, our vertex dominance embedding technique does not use learning-based graph embedding, which can guarantee exact subgraph matching over dynamic graphs without false dismissals and enable incremental embedding maintenance over dynamic graphs upon updates.

10 CONCLUSIONS

In this paper, we formulate and tackle the dynamic subgraph matching (DSM) problem, which continuously monitors the matching subgraph answers over a large-scale dynamic graph. We propose a general framework for efficiently processing DSM queries, based on our carefully-designed vertex dominance embeddings. We also provide an effective degree grouping technique and pruning strategies to facilitate our efficient algorithms of retrieving/maintaining DSM subgraph answers. Most importantly, we devise an effective cost model for guiding the design of vertex embeddings and further enhance the pruning power of our DSM query processing. Through extensive experiments, we evaluate the performance of our DSM approach over real/synthetic dynamic graphs.

REFERENCES

[1] Ahmed Al-Baghdadi, Gokarna Sharma, and Xiang Lian. 2020. Efficient processing of group planning queries over spatial-social networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2020), 2135–2147.

[2] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Reviews of Modern Physics* 74, 1 (2002), 47.

[3] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. 2019. Sequential and parallel solution-biased search for subgraph algorithms. In *Proceedings of the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. 20–38.

[4] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the International Conference on Web Search and Data Mining (WSDM)*. 384–392.

[5] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512.

[6] Albert-László Barabási and Eric Bonabeau. 2003. Scale-free networks. *Scientific American* 288, 5 (2003), 60–69.

[7] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 28–39.

[8] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1447–1462.

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1199–1214.

[10] 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, 7 (2013), 1–13.

[11] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. 2001. The skyline operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 421–430.

[12] Lei Chen and Changliang Wang. 2010. Continuous subgraph pattern search over certain and uncertain graph streams. *IEEE Transactions on Knowledge and Data Engineering* 22, 8 (2010), 1093–1109.

[13] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. 2009. Monitoring path nearest neighbor in road networks. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 591–602.

[14] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 157–168.

[15] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.

[16] Sergei N Dorogovtsev and José FF Mendes. 2003. *Evolution of networks: From biological nets to the Internet and WWW*. Oxford University Press.

[17] Chi Thang Duong, Trung Dung Hoang, Hongzhi Yin, Matthias Weidlich, Quoc Viet Hung Nguyen, and Karl Aberer. 2021. Efficient streaming subgraph isomorphism with graph neural networks. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, Vol. 14. 730–742.

[18] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph pattern matching: From intractable to polynomial time. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, Vol. 3. 264–275.

[19] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Expfinder: Finding experts by graph pattern matching. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1316–1319.

[20] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems* 38, 3 (2013), 1–47.

[21] Michael R. Garey and David S. Johnson. 1983. Computers and intractability: A guide to the theory of NP-completeness. *The Journal of Symbolic Logic* 48, 2 (1983), 498–500.

[22] Martin Grohe and Pascal Schweitzer. 2020. The graph isomorphism problem. *Commun. ACM* 63, 11 (2020), 128–134.

[23] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 855–864.

[24] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *The International Journal on Very Large Data Bases* 3 (1994), 357–399.

[25] Aric Hagberg and Drew Conway. 2020. Networkx: Network analysis with python. *URL: https://networkx. github. io* (2020).

[26] 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 *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1429–1446.

[27] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 337–348.

[28] Yu Hao, Xin Cao, Yufan Sheng, Yixiang Fang, and Wei Wang. 2021. Ks-gnn: Keywords search over incomplete graphs via graphs neural network. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. 1700–1712.

[29] Oktie Hassanzadeh, Anastasios Kementsietsidis, and Yannis Velegrakis. 2012. Data management issues on the semantic web. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1204–1206.

[30] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 405–418.

[31] Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. 1995. Computing simulations on finite and infinite graphs. In *Proceedings of the IEEE Annual Foundations of Computer Science (AFCS)*. 453–462.

[32] Chenji Huang, Yixiang Fang, Xuemin Lin, Xin Cao, and Wenjie Zhang. 2022. Able: Meta-path prediction in heterogeneous information networks. *ACM Transactions on Knowledge Discovery from Data* 16, 4 (2022), 1–21.

[33] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1259–1274.

[34] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The International Journal on Very Large Data Bases* 29, 2 (2020), 619–653.

[35] Alpár Jüttner and Péter Madarasi. 2018. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81.

[36] Xuan Kan, Zimu Li, Hejie Cui, Yue Yu, Ran Xu, Shaojun Yu, Zilong Zhang, Ying Guo, and Carl Yang. 2023. R-Mixup: Riemannian Mixup for Biological Networks. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1073–1085.

[37] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1695–1698.

[38] Guy Karlebach and Ron Shamir. 2008. Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology* 9, 10 (2008), 770–780.

[39] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2017. Subgraph querying with parallel use of query rewritings and alternative algorithms. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 25–36.

[40] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 411–426.

[41] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 133–144.

[42] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *Proceedings of the International Conference on Machine Learning (ICML)*. 3835–3845.

[43] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. 2021. Symmetric continuous subgraph matching with bidirectional dynamic programming. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 1298–1310.

[44] Jayanta Mondal and Amol Deshpande. 2016. Casqd: continuous detection of activity-based subgraph pattern queries on dynamic graphs. In *Proceedings of the International Conference on Distributed and Event-based Systems (ICDES)*. 226–237.

[45] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics* 46, 5 (2005), 323–351.

[46] Abdelghny Orogat and Ahmed El-Roby. 2022. SmartBench: demonstrating automatic generation of comprehensive benchmarks for question answering over knowledge graphs. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 3662–3665.

[47] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, and Jingren Zhou. 2019. Hop-constrained s-t Simple Path Enumeration: Towards Bridging Theory and Practice. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, Vol. 13. 463–476.

[48] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 701–710.

[49] 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. *Proceedings of the International Conference on Very Large Data Bases*

(PVLDB) 11, 12 (2018), 1876–1888.

[50] Niranjan Rai and Xiang Lian. 2023. Top-k Community Similarity Search Over Large-Scale Road Networks. *IEEE Transactions on Knowledge and Data Engineering* 35, 10 (2023), 10710–10721.

[51] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 617–628.

[52] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. 2017. struc2vec: Learning node representations from structural identity. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 385–394.

[53] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 364–375.

[54] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. 2014. Event pattern matching over graph streams. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, Vol. 8. 413–424.

[55] Li Sun, Zhenhao Huang, Zixi Wang, Feiyang Wang, Hao Peng, and S Yu Philip. 2024. Motif-aware Riemannian Graph Neural Network with Generative-Contrastive Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 38. 9044–9052.

[56] Li Sun, Zhongbao Zhang, Junda Ye, Hao Peng, Jiawei Zhang, Sen Su, and S Yu Philip. 2022. A self-supervised mixed-curvature graph neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 36. 4146–4155.

[57] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. Pathenum: Towards real-time hop-constrained st path enumeration. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1758–1770.

[58] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1083–1098.

[59] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, Vol. 15. 1403–1416.

[60] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 788–799.

[61] Damian Szklarczyk et al. 2015. STRING v10: protein–protein interaction networks, integrated over the tree of life. *Nucleic Acids Research* 43, D1 (2015), D447–D452.

[62] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *Proceedings of the Web Conference (WWW)*. 1067–1077.

[63] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1225–1234.

[64] Hanchen Wang, Defu Lian, Wanqi Liu, Dong Wen, Chen Chen, and Xiaoyang Wang. 2022. Powerful graph of graphs neural network for structured entity analysis. In *Proceedings of the Web Conference (WWW)*. 609–629.

[65] Hanchen Wang, Defu Lian, Ying Zhang, Lu Qin, Xiangjian He, Yiguang Lin, and Xuemin Lin. 2021. Binarized graph neural network. In *Proceedings of the Web Conference (WWW)*. 825–848.

[66] Hanchen Wang, Defu Lian, Ying Zhang, Lu Qin, and Xuemin Lin. 2020. Gognn: Graph of graphs neural network for predicting structured entity interactions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1317–1323.

[67] Stanley Wasserman and Katherine Faust. 1994. Social network analysis: Methods and applications. (1994).

[68] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *Nature* 393, 6684 (1998), 440–442.

[69] Wikipedia. 2023. Central Limit Theorem. *https://en.wikipedia.org/wiki/Central_limit_theorem* (2023).

[70] Carl Yang and Jiawei Han. 2023. Revisiting citation prediction with cluster-aware text-enhanced heterogeneous graph neural networks. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 682–695.

[71] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-based Path Dominance Embedding. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 1628–1641.

[72] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative Subboundedness of Contraction Hierarchy and Hierarchical 2-Hop Index in Dynamic Road Networks. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1992–2005.

[73] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. In *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*. 340–351.