Fast and Accurate Efficient Streaming Subgraph Isomorphism

Chi Thang Duong [1], Trung Dung Hoang [2], Hongzhi Yin [3], Matthias Weidlich [4], Quoc Viet Hung Nguyen [2], Karl Aberer [1]

[1] École Polytechnique Fédérale de Lausanne, Switzerland, [2] Griffith University, Australia,
[3] The University of Queensland, Australia, [4] Humboldt-Universität zu Berlin, Germany

ABSTRACT

Queries to detect isomorphic subgraphs are important in graph-based data management. While the problem of subgraph isomorphism search has received considerable attention for the static setting of a single query, or a batch thereof, existing approaches do not scale to a dynamic setting of a continuous stream of queries. In this paper, we address the scalability challenges induced by a stream of subgraph isomorphism queries by caching and re-use of previous results. We first present a novel subgraph index based on graph embeddings that serves as the foundation for efficient stream processing. It enables not only effective caching and re-use of results, but also speeds-up traditional algorithms for subgraph isomorphism in case of cache misses. Moreover, we propose cache management policies that incorporate notions of reusability of query results. Experiments using real-world datasets demonstrate the effectiveness of our approach in handling isomorphic subgraph search for streams of queries.

PVLDB Reference Format:
Subgraph Isomorphism Search. *PVLDB*, xx(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Graphs are a natural representation of relations between entities in complex systems, such as social networks, chemical compounds, or biological structures [49, 13, 48, 24, 12]. Hence, efficient management of graph-structured data is of crucial importance in diverse domains and subgraph isomorphism queries are an important means to detect patterns in larger graphs [42, 50, 58]. Specifically, given a query graph q (i.e., the pattern) and a data graph g, such a query returns all mappings of nodes of q to nodes of g that preserve the respective edges. Answering subgraph isomorphism queries is useful, for instance, to analyze propagation patterns in social networks or to query protein connections in protein interaction networks.

Since the problem of subgraph isomorphism search is NP-Hard, various heuristics to speed up the search have been proposed [10, 58, 20, 50]. These algorithms have in common that they are based on measures of node similarity and subgraph similarity. The former enables conclusions on nodes of the data graph that cannot be mapped to nodes of the query graph and are, therefore, filtered. Measuring subgraph similarity, in turn, is the first step of verifying whether a subgraph of the data graph is isomorphic to the query graph.

In domains such as social networks, chemistry, or biology, subgraph isomorphism queries occur frequently. They are issued concurrently by many users and systems. For instance, ChemSpider is a search engine with an API that answers subgraph isomorphism queries for molecular structures in a database of more than 77 million molecules [7]. Once a stream of queries is considered, the aforementioned algorithms to subgraph isomorphism search become infeasible. They employ notions of similarity for nodes and subgraphs that are based on the actual structure of the graphs. Since the respective structural comparison has a worst-case runtime complexity of $O(N! \cdot N^2)$ in the size of the graphs [10] for large query graphs, or $O(N^k)$ for small query graphs with k nodes [34], traditional approaches do not scale to a streaming setting.

For other data models, techniques to process a continuous stream of queries are commonly addressed using caching strategies. Caching is possible in these cases as the queries show a large overlap, which enables re-use of previous results. Examples include techniques to evaluate queries in web search engines [2, 39, 30] and to answer resource requests in web applications [14, 1, 15]. In either case, cached query results are re-used when answering subsequent queries. However, this principle cannot be adopted directly for subgraph isomorphism queries, since it was shown empirically that most existing techniques for structural indexing have an exponential runtime [26]. Hence, it is infeasible to index the data graph, or parts thereof, as it would be required for efficient caching and re-use of query results.

In this paper, we propose to use embeddings as a foundation for the evaluation of subgraph isomorphism queries. An embedding maps a graph to a numerical space, such that structurally-similar nodes and subgraphs are close to each other [37, 41, 19]. Embeddings support indexing naturally. Nodes and subgraphs are points in a high-dimensional space, so that indices for space partitioning, e.g., R-tree [18] or kd-tree [11], may be leveraged. Based thereon, similarity computation or nearest neighbor search are realized efficiently.

Using embeddings as the basis for subgraph isomorphism further has the advantage that it enables cache management based on diversity considerations. A diversified cache is more effective since query graphs in a stream are more likely to be similar to those cached already. However, diversity-based caching for subgraph isomorphism is infeasible for traditional structural indexing due to the computational overhead induced by a structural comparison of query graphs and cached graphs. Yet, using embeddings, the graph comparisons becomes fast and accurate, so that our work is the first to propose such diversity-based cache management for subgraph isomorphism.

To realize the above vision, we define the problem of *streaming subgraph isomorphism* and propose a framework to solve it (§2). We

Proceedings of the VLDB Endowment, Vol. xx, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

then instantiate this framework, making the following contributions:

- *Graph indexing using embeddings* (§3). As the basis for our work, we propose an indexing mechanism based on node, edge, and subgraph embeddings. While we incorporate state-of-the-art techniques for graph representation learning, we provide a theoretical justification for our mechanism by showing that the embedding process is similar to Weisfeiler-Lehman isomorphism testing [35].
- *Query stream processing with a cache* (§4). Using the indices, we show how to answer a stream of subgraph isomorphism queries while exploiting cached results. Specifically, upon the arrival of a query, similar past queries are identified to re-used their results. In the case of a cache miss, subgraph embeddings are exploited to speed up traditional algorithms for subgraph isomorphism (e.g., TurboISO [20]). In case of a cache hit, we assess the overlap of the current query with the cached ones and derive an answer from the cached results.
- *Cache management* (§5). As the size of a cache is limited, we need to control cache admission and eviction. To this end, we propose a policy that minimizes the number of cache misses. Compared to traditional policies (LRU [57] or Greedy-Dual [57]), it assesses the utility of a query result not only based on processing time, but includes a notion of diversity.

We evaluate our approach using several real-world datasets in §6. We show that our embedding-based index outperforms structural indices by two orders of magnitude. When answering subgraph isomorphism queries, our approach based on caching and re-use of results leads to runtime improvements of at least 100% over state-of-the-art algorithms such as MQO [42] and TurboISO [20]. We review related work in §7 and conclude in §8.

2. MODEL AND APPROACH

2.1 Model

We target the problem of subgraph isomorphism search for undirected, labelled graphs. Let $g = (V, E)$ be a *graph* with a set of nodes V and a set of edges $E \subseteq V \times V$. It is associated with a labeling function $l : V \rightarrow \Sigma$ that captures intrinsic properties of its nodes. If the alphabet of labels Σ is defined as $\mathbb{R}^k$, i.e., labels are k-dimensional real vectors, we refer to (g, l) as an attributed graph.

Two attributed graphs (g_1, l_1) and (g_2, l_2) are *isomorphic*, if there exists an edge-preserving bijective function $f : V_1 \rightarrow V_2$ such that:
(1) $\forall\, v \in V_1 : l_1(v) = l_2(f(v))$, and
(2) $\forall\, (v_1, v_2) \in E_1 : (f(v_1), f(v_2)) \in E_2$.

If g_1 is isomorphic to an induced subgraph g_2' of g_2, g_1 is *subgraph isomorphic* to g_2, written as $g_1 \preceq g_2$. We call the bijection between g_1 and g_2' a *mapping*, and g_1 is said to have a mapping in g_2. There may be several mappings of g_1 in g_2. We write $F(g_1, g_2) = \{f_1, f_2, \ldots, f_k\}$ for the set of all mappings. The *subgraph isomorphism* problem is to find all mappings $F(g_1, g_2)$ for a given pair of graphs.

In graph-based data management, a subgraph isomorphism query is defined through a *query graph* $q = (V', E')$ for which the subgraph isomorphism problem shall be solved regarding a *data graph* $g = (V, E)$. We target scenarios in which queries arrive continuously. We therefore define a *query stream* as a sequence of queries, $Q = \langle q_1, q_2, \ldots \rangle$, arriving one after another. Each query arrives at a particular point in time, denoted by $q_i.t$, and the stream is totally ordered by these time points, i.e., for any two queries q_i and q_j of the stream, if $i < j$ then $q_i.t < q_j.t$. We denote the finite prefix of stream Q until index k as $Q_{[k]} = \langle q_1, \ldots, q_k \rangle$. In our setting, the queries in the stream may overlap or repeat, so that results stored for previous queries may be re-used.

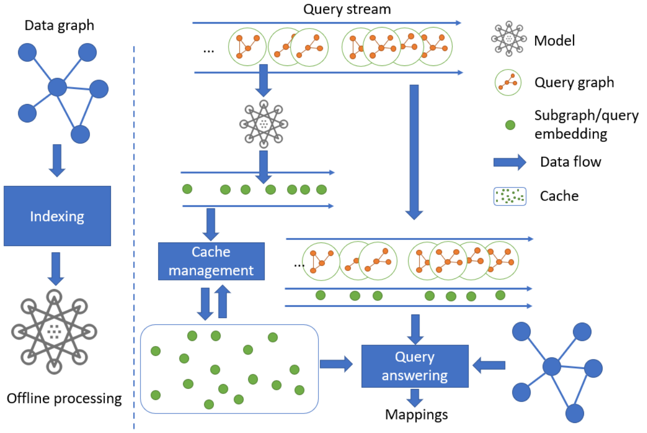

Figure 1: Framework for streaming subgraph isomorphism.

Query processing incurs a latency, i.e., the time between the arrival of a query and the time it is answered. Based thereon, we capture the problem addressed in this paper, as follows:

PROBLEM 1 (STREAMING SUBGRAPH ISOMORPHISM). *Given a data graph, the problem of* streaming subgraph isomorphism *is to solve the subgraph isomorphism problem for all queries of a query stream, while minimizing the processing latency.*

2.2 Approach

To address the problem of streaming subgraph isomorphism, we propose a framework that exploits caching strategies. Our idea is to re-use query results for a large number of the queries in the query stream, thereby minimizing the processing latency. However, a realization of this idea raises several research questions: (Q1) *how to index* nodes, edges, and subgraphs for efficient caching and re-use of query results? (Q2) *how to answer* queries based on cached results? (Q3) *how to manage* the result cache? The interplay of these questions is shown in the illustration of our framework in Fig. 1. Below, we summarize our techniques to instantiate this framework.

We present a novel method for graph indexing, which speeds up the search for isomorphic subgraphs. The index is based on embeddings of nodes, edges, and subgraphs, in which similar nodes, edges, and subgraphs have similar index values. While the subgraph index enables us to identify re-usable query results in a swift manner, the node and edge indices accelerate traditional algorithms for subgraph isomorphism by pruning the search space.

Our indices serve as a foundation for a novel evaluation algorithm for streaming subgraph isomorphism. It exploits cached results whenever possible. In case of a cache miss, our node and edge indices speed up *any* existing branch-and-bound algorithm used to solve the subgraph isomorphism problem.

In the light of a limited cache size, we further present policies for cache management. Specifically, we propose to store only a fixed amount of results per query to enable uniform retrieval. To guide cache admission and eviction, we adapt the Landlord algorithm to the setting of streams of subgraph isomorphism queries, which results in a high ratio of cache hits.

3. GRAPH INDEXING

This section introduces indices for nodes, edges, and subgraphs based on graph embeddings. To this end, we first give further background on embeddings (§3.1). We then introduce approaches to learn node and edge embeddings (§3.2) and subgraph embeddings (§3.3). Based there, we define the respective indices (§3.4).

3.1 Background on Embeddings

Embeddings are a model to represent concepts in some numeric space. Yet, this representation shall be such that semantically related concepts have close representations, i.e., their geometric relation in the embedding space encodes their semantic relation. Compared to symbolic representations that consider each concept as independent, embeddings enable conclusions on the relation of the concepts based on their representations. Moreover, embeddings are succinct in the sense that with a d-dimensional embedding space (d is called the embedding size) where the domain of each dimension has size k, k^d concepts can uniquely be described.

In our setting, the embeddings of nodes in a graph that are structurally similar are vectors that are close to each other. Similarly, subgraphs of similar structure are assigned close vectors.

3.2 Node and Edge Embeddings

When computing embeddings for nodes, there are two kinds of semantic information to consider: The labels assigned to nodes and their connections to other nodes in the graph. Assuming that semantically similar nodes are assigned similar labels, the respective representation can be incorporated directly in a node embedding. Yet, labels commonly capture external knowledge, not the graph structure. Therefore, we propose to follow the idea of message passing neural networks (MPNN) to enrich the node embeddings with structural information.

Note that we use the embeddings as a means to index nodes, edges, and subgraphs. This is different from traditional graph indexing [5, 29] that relies on subgraphs such as paths, triangles, and cliques as reference points in graph comparison. Our approach avoids the need to detect such subgraphs to construct the respective indices.

Message-passing framework. In a Message Passing Neural Network (MPNN) [16], a node representation is created by combining the representation of its own properties with those of its neighbors, through message-based interactions. A message sent from one node to its neighbors is constructed based on the node's current representation. Since messages are exchanged only between nodes connected by edges, the graph structure is incorporated. Message passing happens in several rounds, each involving three steps [13]:

Sending: A node u constructs a message in the i-th round based on its representation $z_u^{(i)}$. The node sends the message to a set of *selected* neighbors using a parameterized function $f_s^{(i)}$:

$$m_u^{(i)} = f_s^{(i)}(z_u^{(i)}).$$

Receiving: Once a node v received messages from all its neighbors, denoted by $N(v)$, in a round, it aggregates them using a parameterized function $f_a^{(i)}$:

$$z_{N(v)}^{(i)} = f_a^{(i)}(\{m_u^{(i)}, \forall\, u \in N(v)\}).$$

Updating: A node updates its representation, combining its current representation with the aggregated messages:

$$z_v^{(i+1)} = f_u^{(i)}(z_{N(v)}^{(i)}, z_u^{(i)})$$

An MPNN can be formulated as a function $f(g, l)$, where g is a graph and l is a labeling function. The function f represents the combination of the above functions used in the sending, receiving, and updating steps of all rounds and returns a set of node embeddings $\{z_u\}$ for each node in the graph. Note that the parameters of f need to be learned before the model can be used, though.

EXAMPLE 1. *Given the graph on the left of Fig. 2, an embedding for node B is created using a 1-layer MPNN as: $z_B^{(1)} = f_u^{(0)}(z_{N(B)}^{(0)}, z_B^{(0)})$ where $z_{N(B)}^{(0)} = f_a^{(0)}(\{f_s^{(0)}(z_C^{(0)}), f_s^{(0)}(z_A^{(0)})\})$ is the embedding of the neighborhood of B. In its basic form, function f_s of the sending step is parameterized by a matrix W_s, i.e., $f_s(z) = W_s z$. The aggregation function in the receiving step derives the mean of the node embeddings, i.e., $f_a(N(v)) = 1/|N(v)| \sum_{u \in N(v)} z_u$. Function f_u of the updating step is parameterized by a matrix W_a, before applying a non-linearity, i.e., $f_u(z_{N(v)}, z_v) = \sigma\big((z_{N(v)} + z_v)W_a\big)$. Based thereon, the node embedding of B is computed:*

$$z_B^{(1)} = f_u(\{m_u^{(0)} \mid u \in N(B)\}, z_B^{(0)}) = \sigma\big(((z_A^{(0)}W_s + z_C^{(0)}W_s)/2 + z_B^{(0)})W_a\big) \quad (1)$$

A node-centric view. An MPNN may also be viewed from a node's perspective. Then, the operations to compute the embedding for a node u induce a k-layer tree, rooted at u. The embedding of u is based on the nodes at the i-th layer of the tree, which are neighbors of u within distance i in the graph. Information at the leaves of the tree is given by the labels of the respective nodes, which is then aggregated to the root: The i-th layer employs the parameterized function f_i to aggregate the results of the i+1-th layer.

Node u and its neighbors within distance k induce a subgraph, called the *receptive field* of u. The higher the number of rounds of message passing, the larger the receptive field. The embedding of u represents a summarization of its receptive field, in terms of both structure (as message passing follows the graph structure) and node labels (as messages are constructed initially from node labels).

EXAMPLE 2. *The node-centric view is illustrated in Fig. 2. Given the graph on the left and using a 2-layer MPNN, the embedding of node B is constructed by aggregating the embeddings of nodes C and A in the first layer. These embeddings are, in turn, constructed from the embeddings of their neighbors. This process is captured by a 2-layer tree rooted at B.*

MPNN and isomorphism testing. The use of MPNN in our setting is theoretically well-grounded, as it is related to the Weisfeiler-Lehman (WL) isomorphism test [35]. The WL algorithm also proceeds in rounds and, in the k-th iteration, constructs a node labeling $l^k : V \rightarrow \Sigma$ by considering the labels assigned to nodes and their neighbors in the previous iteration. That is, the label of a node v at the k-th iteration is derived as:

$$l_v^{(k)} = h\big(\{l_u^{(k-1)} \mid u \in N(v)\}, l_v^{(k-1)}\big) \quad (2)$$

where h is a hash function that maps to a new label, not used in previous iterations. Running the above procedure on two graphs simultaneously, we can test if they are isomorphic: If in any iteration, the constructed node labels differs, the graphs are not isomorphic [35]. This process is illustrated in Fig. 2.

EXAMPLE 3. *In Fig. 2, right side, labels are visualized by a pattern. In the first iteration, we construct the label of node C by hashing the set containing its own label and the labels of its neighbors. The same is done for node A. The results are used in the 2nd iteration to derive the label for node B.*

The formulations of the MPNN in Eq. 1 and the WL algorithm in Eq. 2 are similar. Their relationship is formalized as follows.

THEOREM 1 (WEISFEILER-LEHMAN TESTING & MPNN). *Given a labeled graph (g, l), let $l^{(k)}$ be the node labeling obtained using the WL algorithm after k iterations and $z^{(k)}$ be the embeddings obtained by a k-layer MPNN. Then, with suitable initial embeddings $z^{(0)}$ and parameterized functions of the MPNN, for all nodes u, v of g, if $l^{(k)}(u) = l^{(k)}(v)$ then $z^{(k)}(u) = z^{(k)}(v)$.*

The above result follows directly from Theorem 1 in [35]. It shows that the MPNN-based formulation is as strong as the WL isomorphism test, which provides a theoretical basis for applying MPNN

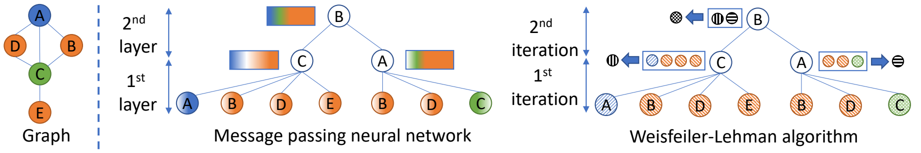

Figure 2: Message-passing neural network (color gradients represent embeddings) vs. Weisfeiler-Lehman algorithm (patterns illustrate symbolic representations).

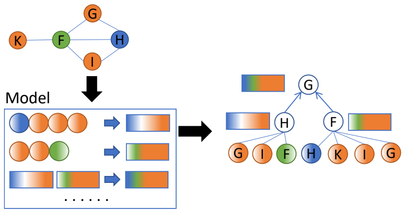

Figure 3: Different, but isomorphic, graph yields equivalent embeddings.

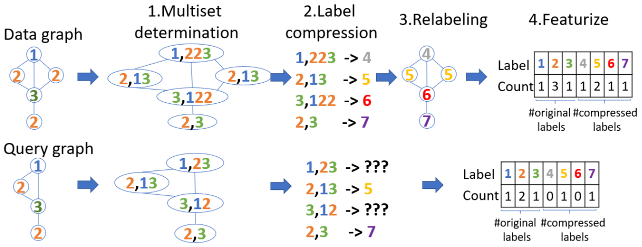

Figure 4: Embedding generation process by WL.

in our framework for streaming subgraph isomorphism. However, the WL algorithm and the MPNN differ in how they represent node labels. The labels derived with the WL algorithm are symbolic representations, i.e., unrelated symbols. The embeddings obtained with the MPNN capture semantic relations, so that an assessment of their similarity is meaningful. In Fig. 2, we distinguish between both representation by color gradients and patterns, respectively.

WL vs. MPNN. Theorem 1 shows that WL and the MPNN have the same strength to detect graph isomorphism. However, WL requires defining a hash function to compress a multiset to a label as shown in Fig. 4. This is problematic, once isomorphism shall be detected for graphs with unseen properties. Consider Fig. 4, where a query graph comprises two multisets (1,23) and (3,12) that have not yet been encountered. WL cannot compress the label and, hence, cannot conduct the isomorphism test effectively. This issue could be addressed in three ways: 1) assuming knowledge of all graphs, a multi-graph WL algorithm is employed to construct all required hash functions; 2) new labels are assigned to new multisets, which are then added in the featurization; or 3) hash functions are removed and the comparison is performed directly on the multiset. While the first solution is not realistic, the second one incurs many zero values in the embeddings, so that comparison becomes imprecise. The third solution incurs significant overhead in terms of processing time to measure the similarity of multisets. The MPNN, in turn, can handle new query graphs seamlessly. Intuitively, it compares multisets, but relies on the embeddings as succinct representations of fixed size, which renders this comparison more efficient. We later confirm empirically that WL is computationally more expensive than the MPNN regarding new queries.

Parameter learning. To learn the parameters of the MPNN, a loss function needs to be defined. As mentioned, a node embedding represents a summarization of its receptive field. Hence, we define a loss function that rewards if similar embeddings are assigned to similar nodes, i.e., those that are close in the graph:

$$L(z_v) = -log(\sigma(z_v^T z_u)) - Q\mathbb{E}_{u_n \sim P_n(v)} log(\sigma(-z_v^T z_{u_n}))$$

where v is called a positive sample such as u's neighbor, u_n is a negative sample obtained from a negative sampling distribution P_n, and Q is the number of negative samples. The above function strives for similar representations for similar nodes u, v by maximizing $z_v^T z_u$, while minimizing $z_v^T z_{u_n}$ fosters different representations for dissimilar nodes v, u_n. We observe that adding a supervised loss function to reconstruct the node labels to the unsupervised loss can also improve the model's performance.

Edge embeddings. As usual, we construct edge embeddings by averaging the embeddings of the adjacent nodes. We later show that edge embeddings are more discriminative than node embeddings as they enable better pruning of candidates for subgraph isomorphism.

3.3 Subgraph Embeddings

For subgraph embeddings to be meaningful, similar subgraphs shall have close embeddings and the labels of nodes shall be incorporated. Moreover, when considering the problem of streaming subgraph isomorphism, we need to cater for large differences in the sizes of the assessed graphs. Given a small query graph, there are potentially very many isomorphic subgraphs in a large data graph [42]. An embedding shall support a test for isomorphism that is independent of the specific locations of these subgraphs.

Truncated message-passing for subgraph embedding. Our approach to embed subgraphs of a labeled graph (g, l) (the data graph, in our setting) builds on the function $Z = f(g, l)$ that returns embeddings for all nodes in g. This model, learned on the whole graph, captures the graph's structure in a comprehensive manner. Hence, for a labeled subgraph (s, l'), we can project the model on the respective nodes and their labels, which yields an embedding $Z' = f(s, l')$. Such a projection is akin to truncated message passing, in which solely the nodes in s send messages to neighboring nodes that are also in s. Note though that the parameters of the functions used for sending, receiving, and updating are taken from the MPNN learned to embed the individual nodes.

EXAMPLE 4. *Fig. 5 illustrates truncated message passing for a graph of four nodes, A-D, which is a subgraph of the one in Fig. 2. Messages are exchanged only within the subgraph, but not with node E. Hence, the tree of operations, rooted at B, does not include E.*

The above process yields embeddings for all nodes in a subgraph. Since each embedding summarizes the node's receptive field, i.e., the subgraph, it is a candidate to represent the whole subgraph. Against this background, we follow a compositional approach and average the node embeddings to represent the subgraph.

In addition, we propose an approach to construct subgraph embeddings from the edge embeddings. Again, a compositional approach is adopted that averages the edge embeddings to represent the subgraph. This is equivalent to a degree-weighted combination of the node embeddings as shown in the following formula:

$$\begin{aligned} z_s &= \frac{1}{|E_s|} \sum_{(u,v) \in E_s} z_{(u,v)} = \frac{2}{\sum_{u \in V_s} deg(u)} \sum_{(u,v) \in E_s} \frac{z_u + z_v}{2} \\ &= \frac{2}{\sum_{u \in V_s} deg(u)} \sum_{u \in V_s} deg(u) z_u = \frac{\sum_{u \in V_s} deg(u) z_u}{\sum_{u \in V_s} deg(u)} \end{aligned} \quad (3)$$

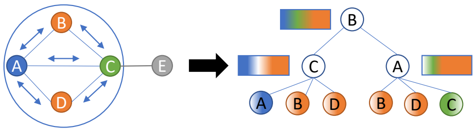

Figure 5: Illustration of truncated message passing.

We later show empirically that edge embeddings lead to better subgraph embeddings, as cache management becomes more effective.

Transferring models. The model learned to embed the nodes (and, hence, subgraphs) of one graph, may also be transferred to another graph. In our context, it enables us to apply the model learned for the data graph g also to a query graph q. Specifically, we derive the node embeddings $Z_q = f(q, L_q)$ of the query graph, which are then aggregated to obtain an embedding for the whole query graph using the above process. This way, subgraph embeddings of the data graph and the embedding of the query graph are constructed using the same model. Hence, a subgraph of the data graph that is isomorphic to the query graph has an equivalent embedding.

EXAMPLE 5. *Fig. 3 illustrates the application of the earlier model to a new graph. Intuitively, the model defines 'rules' to combine embeddings at different layers. Applying the model to isomorphic graphs, see Fig. 2 and 3, yields equivalent embeddings.*

3.4 Indexing Embeddings

Similarity computation. To assess the structural similarity of two nodes, or subgraphs, we compute the cosine similarity of their embeddings. This choice is motivated by the locality property of the cosine similarity: It emphasizes the immediate neighborhood of the nodes, independent of their global location in the graph [6].

Indexing high-dimensional embeddings. In our context, the relative similarity of embeddings is more important than their absolute similarity. When answering streams of subgraph isomorphism queries, it is important to find nearest neighbors in the high-dimensional embedding space. Since we approach the problem of subgraph isomorphism based on embeddings, we can rely on a large body of work on indexing for fast nearest neighbor search in numeric spaces. Specific examples include R-trees [18] and kd-trees [11], which have been shown to be efficient and scalable.

Note that these indexing techniques can be applied to cosine similarity by normalizing each embedding to have a length of one. In this case, the cosine similarity corresponds to the dot product between two embeddings, which is negatively correlated with their Euclidean distance. Moreover, several variants of R-trees and kd-trees that can handle high-dimensional embeddings have been proposed [44, 36, 25]. In our experiments, we later adopt an improved version of the kd-tree [44] and also observe that a relatively small embedding size is sufficient to achieve good performance.

4. QUERY STREAM PROCESSING

This section introduces our approach to answer a stream of subgraph isomorphism queries. Our idea is to cache and re-use the results of past queries to derive a full or a partial answer to the current query. To identify suitable past queries, we leverage the subgraph indices introduced above. Specifically, for each answered query w, the embedding z_w is indexed. Given a new query q with embedding z_q, identify those past queries w, for which the distance between the embeddings z_w and z_q is below a threshold τ. Depending on whether at least one such query w is identified, we refer to the situation as a cache miss or a cache hit, respectively. For either case, we describe our approach in the remainder of this section.

Algorithm 1: Generic search for subgraph isomorphism.

```
Input  : Query graph $q$; data graph $g$.
Output : All isomorphic subgraphs of $q$ in $g$.
1 $M = \emptyset$;
2 $\Omega = EnumerateStructures(g)$;
3 while $|M| <> |q|$ do
4     $s \leftarrow GetNextStructure(q)$;
5     for $s' \in FilterCandidates(s, M, \Omega)$ do
6         $M \leftarrow Combine(M, s')$;
7         if $M$ is valid then $R \leftarrow R \cup \{M\}$;
8         $M \leftarrow Backtrack()$;
9 return $R$
```

4.1 Handling Cache Misses

In case of a cache miss, we resort to traditional algorithms for subgraph isomorphism. These algorithms follow a backtracking strategy, which explores solutions incrementally, abandoning those that turn out to be invalid. Alg. 1 illustrates this generic process for a given query graph q and a data graph g. Here, a crucial step is to filter candidate structures to map to those of the query graph (line 5). In the worst case, filtering is invoked an exponential number of times, in the size of the data graph, since it relies on the current partial mapping. Hence, the filter step needs to be efficient.

Common subgraph isomorphism algorithms match a query graph and a data graph based on their nodes. In that case, Ω in line 2 contains the nodes of the data graph and s' is the data graph node that matches the query graph node s according to the filter strategy in line 5. Depending on the specific algorithm, the filter leverages a node's label and degree (Ullmann's algorithm [50]) or its connections (VF2 [10] and QuickSI [43]). Yet, simple, efficient strategies based on a node's label or degree, tend to be of limited effectiveness.

Advanced strategies to filter candidates for subgraph isomorphism work on the level of subgraphs, not on the level of nodes. While they are commonly very selective, they also suffer from a high computational overhead. For instance, QuickSI [43] constructs minimum spanning trees and GADDI [58] is based on shortest path computation. These algorithms need to enumerate particular structures in both, the query and the data graph, which are then used for similarity computation [43, 58]. This enumeration is expensive, as it essentially solves another graph isomorphism problem.

Embedding-based pruning. We propose to filter candidate structures using their embeddings. Specifically, using the model f, subgraph embeddings of structures of interest in the data graph are created (Ω in Alg. 2-line 2). Note that these embeddings are computed offline. For each structure s identified in the query graph (Alg. 2-line 4), we also construct an embedding. Based thereon, we identify candidate structures in g for s by extracting k nearest neighbors of z_s. Alg. 2 summarizes this idea. First, subgraph embeddings of g are computed offline (line 1-line 3). Then, for each structure of interest in q (line 5), the embedding z_s is computed online (line 6). It is used for a k-nearest neighbor search over the embeddings Z of subgraphs of g (line 7) to obtain candidate structures.

Selecting subgraphs. When searching for isomorphic subgraphs, pruning of candidate structures may be based on different kinds of subgraphs, e.g., nodes [10], trees [43], or paths [58]. Smaller and simpler subgraphs are easier to enumerate, whereas they are less selective. In our setting, we use 2-node subgraphs (i.e., edges) as the basis for a pruning strategy. Edges are easy to enumerate while being more discriminative than nodes. Although it is possible to use 3- or 4-node subgraphs, the exponential growth of the respective subgraphs, illustrated in Table 1 for the datasets later used in our experiments, induces severe computational challenges.

Table 1: Number of subgraphs of different sizes

	1-node	2-node	3-node	4-node
yeast	3'101	12'519	487'696	34'508'857
human	4'674	86'282	7'106'210	>154'376'187
cora	2'708	5'278	59'707	2'270'091
citeseer	3'327	4'600	29'930	643'980
pubmed	19'717	44'324	818'753	28'779'383
wordnet	82'670	127'124	3'260'142	>155'289'760

Algorithm 2: Embedding-based candidate filtering.

```
Input  : Query graph $q$; data graph structures $\Omega$; model $f$; threshold $k$.
Output : Matching candidates for every structure in $q$.
for $s' \in \Omega$ do
    $z_{s'} \leftarrow f(s', L)$ ;                    // Conducted offline
    $Z \leftarrow Z \cup \{z_{s'}\}$ ;
$\mathbb{C} \leftarrow \emptyset$;
for $s \leftarrow GetNextStructure(q)$ do
    $z_s \leftarrow f(s, L')$;                         // Conducted online
    $C_s \leftarrow kNNSearch(z_s, Z, k)$;
    $\mathbb{C} \leftarrow \mathbb{C} \cup \{C_s\}$
return $\mathbb{C}$
```

The above strategy based on embeddings is orthogonal to other filter mechanisms. Note that, we later show experimentally that, this way, adding our strategy to *any* subgraph isomorphism algorithm reduces the number of candidates to consider significantly.

4.2 Handling Cache Hits

Whether the cached results of a past query can be reused for the current query depends on their overlap. Traditionally, the overlap between the query graph q and a cached query graph w is determined by their maximum common subgraph (MCS). The larger the MCS, the better can the results of w be reused for q [33]. Yet, MCS algorithms run in exponential time in the size of the graphs [21]. Hence, the computation of the MCS of the query graph q with *every* cached query graph induces a significant performance penalty.

To speed up this process, we propose to limit the MCS computation to promising cached queries w, i.e., those that are structurally similar to q. To this end, we use the subgraph indices introduced in §3. Specifically, we employ the embeddings of the query graph q and the cached query graphs to find the k nearest neighbors to q. While the kNN search can be done efficiently using our index, the MCS problem needs to be solved solely for k pairs of graphs.

The above steps are the first ones in our general approach to query stream processing, as formalized in Alg. 3. Once promising cached queries have been identified by kNN search (line 1), the MCS is computed for each of them and the query graph (line 4). Based on the MCS node mapping, we then assess the level of reusability of the cached results in terms of four cases:

Case 1: When there is an exact match of q and a cached query w, we return the mapping cached for w after projecting it from q to w and w to g, to obtain a mapping from q to g (line 5).

Case 2: If q is isomorphic with a *subgraph* of a cached query w (line 10), we proceed similarly to the first case and construct the result through projection of the cached mapping.

Case 3: A cached query w is said to have a some overlap with the query $q = (V_q, E_q)$, if their MCS has at least size $|V_q| - \omega$, where ω is an overlap threshold (line 12). The threshold avoids the re-use of results with a very small overlap, which would not pay-off due to the implied overhead. If the overlap is sufficiently large, the cached mapping, after it has been projected from q to w and w to g, is only a subset of the mapping between q and g. However, this partial mapping is useful in the construction of the actual result. Since common algorithms for subgraph isomorphism construct a mapping by establishing correspondence between nodes, one at a time, they can incorporate the partial mapping derived from the cached result as a starting point for the search.

Algorithm 3: Approach to query stream processing.

```
Input  : Data graph $g$; query graph $q = (V_q, E_q)$ with embedding $z_q$;
         cached query embeddings $Z$; neighbors threshold $k$; overlap
         threshold $\omega$.
Output : Node mapping $M$ for $g$ and $q$.
1  $R \leftarrow kNNSearch(z_q, Z, k)$;
2  $bestW, bestMap \leftarrow \emptyset$;
3  for $w \in R$ do
4      $map \leftarrow mcs(w, q)$ ;            // Max common subgraph
5      if $|map| = |V_q| = |w|$ then                    // Case 1
6          return $projectMapping(q, w, map, g)$;
7      if $|map| > |bestMap|$ then
8          $bestW \leftarrow w$;
9          $bestMap \leftarrow map$;
10 if $|bestMap| = |V_q| < |w|)$ then                  // Case 2
11     return $projectMapping(q, w, bestMap, g)$;
12 if $|bestMap| \geq |V_q| - \omega$ then              // Case 3
13     $M \leftarrow projectMapping(q, bestW, bestMap, g)$;
14     return $subgraphIsomorphismInitMap(q, g, M)$
15 else return $subgraphIsomorphism(q, g)$ ;            // Case 4
```

Case 4: If no promising cached queries can be identified (line 15), we observe a cache miss and resort to the procedure described in §4.1, i.e., Alg. 1 with embedding-based filtering (Alg. 2).

In our experiments, we observe that a small value of ω leads to minor differences between a cached query and the current query. In some cases, the difference is a single node, which is not meaningful. A large ω reduces the size of the overlap, which increases the time to detect subgraph isomorphism. In our experiments, we observed that setting ω to be 10% of $|V_q|$ strikes a good balance of this trade-off.

5. CACHE MANAGEMENT

Since a cache has limited size, cache admission and eviction shall be managed such that the number of cache misses is reduced and there is a large overlap between the current and past queries. In this section, we first discuss our general approach to cache management, before we turn to the specific policy.

5.1 General Approach

Cache management in our context resembles the problem of online file caching [57], defined by a cache of fixed size and a sequence of requests to files with assigned retrieval costs. If a file is not in the cache, it is retrieved for the assigned cost, while other files are evicted to make space for it. While such an approach seems useful also for the problem of streaming subgraph isomorphism, there are additional requirements that render existing solutions for online caching to perform poorly in our context.

Cache requirements for streaming subgraph isomorphism. Traditional online caching assumes that every request needs to be answered. However, we may shed queries from cache management, if the incurred delay becomes too large. This provides an additional degree of freedom for a caching policy. Also, in our context, a query may be answered partially by a cached result, see §4.2. Hence, a cache management policy shall consider partial cache hits.

Existing online caching algorithms. While various algorithms for online caching have been proposed in the literature, most of which can be described in the framework of the *Landlord algorithm* [40]. The algorithm assigns a credit to each query to denote the cost of answering it. Keeping the query in the cache, this answer cost is saved. Upon the arrival of a new query that does not match any query in the cache, the credit is decreased for all cached queries. Queries without any remaining credit are evicted. On the other hand, when a query is reused, its credit is increased.

The Landlord algorithm does not satisfy the above requirements: It requires *every* query to be put into cache and incorporates solely complete cache hits. Online caching algorithms such as Greedy-Dual and LRU are instances of the Landlord algorithm, so that they suffer from the same shortcomings [40].

5.2 Query Utility

The above requirements motivate our design of a new policy for cache management, which we coin the *Screening Landlord* (SL) strategy. It adapts the Landlord algorithm and relies on the *utility* of queries to decide on cache admission or eviction. Here, the utility is both, *time-based*, to incorporate the effort to answer the query, and *diversity-based*, to reflect the query re-usability.

Time-based utility. Three aspects influence the time saved by keeping a query in the cache. First, the *answer time* is the time needed to answer the query and add its results to the cache, which corresponds to runtime for the fourth case in Alg. 2. The answer time for a query q, denoted by $a(q)$, is saved in case of cache hit.

Second, the *reuse time* is the time needed to access and reuse the cached result for a query w to answer a query q. It is denoted by $r(q, w)$ and captures the overhead induced by the cache, i.e., the runtime of the first three cases in Alg. 2. Note that the reuse time varies depending on the size of the overlap of the current query and a cached query. The larger the overlap, the smaller the reuse time.

Given a query stream Q, the answer time and reuse time are aggregated per cached query w as follows:

$$\mu_t(w) = \sum_{q\in Q, mcs(q,w)\geq|V_q|-\omega} (a(q) - r(q, w)) \tag{4}$$

Here, the condition $mcs(q, w) \geq |V_q| - \omega$ ensures that only the first three cases of Alg. 2 are considered.

Diversity-based utility. Traditional instances of the Landlord algorithm such as LRU or Recache consider only time and frequency when determining the utility of cached results [57, 40]. However, we strive for caching results of queries that can be reused for many other queries, so that we propose to incorporate a notion of reusability. Intuitively, it is not useful to have many ‘similar’ queries in the cache, as a diverse set of queries increases the chance that results may be reused for a new query. Therefore, we define a notion of diversity-based utility based on the average embedding distance between a query w and a set C of cached queries:

$$\mu_d(q) = \frac{1}{|C|}\sum_{w\in C} dist(z, z_w) \tag{5}$$

The above notion of utility, for the first time, incorporates the diversity of query graphs in cache management. Such an approach would be extremely hard to realize for traditional, structure-based indexing: Measuring dis/similarity between query graphs based on structural properties is computationally expensive and hence, not suited for cache management. Only once embeddings are used, cache management that is guided by the diversity considerations becomes feasible. We later demonstrate empirically that cache diversity is indeed beneficial. It increases the number of cache hits and reduces the answering time significantly.

Combined utility. We combine the above notions to define the overall utility of a query. While both aspects of utility are important, we have to acknowledge that they differ in their normalization factors. Hence, to obtain a meaningful combination, the overall utility is defined by their product, $\mu(q) = \mu_t(q)\mu_d(q)$.

Algorithm 4: The Screening Landlord algorithm.

```
1  Proc screeningLandlord(query q, cache C, threshold t):
2      $\mu_d(q) \leftarrow \frac{1}{|C|}\sum_{w\in C} dist(z_q, z_w)$;
3      $\Delta \leftarrow \min_{w\in C} \mu(w)$;
4      $\tau \leftarrow \frac{\Delta}{\mu_d(q)}$;
5      Try:
6          $s \leftarrow$ current time;
7          process $q$ using Alg. 3;
8          $a(q) \leftarrow$ current time $- s$;
9      Catch current time $- s \geq t$: return ;
10     if $a(q) \geq \tau$ then Add $q$ to cache ;
11 Proc OnInsert(query q):
12     if Cache is full then
13         Evict $w$ with minimum utility;
14         for $w \in C$ do $\mu(w) \leftarrow \mu(w) - \Delta$ ;
15     Add $q$ to cache with utility $\mu(q)$;
16 Proc OnCacheHit(cached query w, query q):
17     $\mu(w) \leftarrow \mu(w) + a(w) - r(q, w)$
```

5.3 Utility-based Cache Management

Using the above notions, we present the Screening Landlord algorithm for cache management. Unlike the traditional Landlord algorithm, it includes screening step that employs two bounds to decide whether a new query q shall be admitted to the cache.

First, there is an upper bound for the time to process q, given as a user-defined threshold. If processing q takes longer than this threshold, the query is not considered for admission to the cache. This way, the overhead of caching is limited for challenging queries.

Second, there is also a lower bound for the time to process q. It is derived dynamically from the minimum overall utility of cached queries and the distance-based utility of q. Specifically, the bound is the ratio of these values. Intuitively, it defines after which answer time of q, there is a break-even point, i.e., the overall utility of q is higher than the minimum utility of a query currently in the cache.

Having discussed the general intuition, Alg. 4 formalizes our Screening Landlord algorithm. For a new query q, we compute its distance-based utility $\mu_d(q)$ based on the subgraph embeddings of q and the queries in the cache C (line 2). We then extract the minimum utility of cached queries (line 3), before determining the lower bound for the answer time of q (line 4). That is, the lower bound τ is set based on the incoming queries and the current cache. Next, query q is processed using Alg. 3, while monitoring the runtime and aborting cache management based on a user-defined threshold (lines 5-8). If query processing finishes before the timeout, the query is admitted to the cache (line 10). This ensures that the lowest utility in the cache does not decrease as we process more queries.

When a query shall be added to the cache, space may need to be made by evicting the cached query with minimum utility (line 13). Following the Landlord algorithm, once a query is evicted, we decrease the utility of every cached query by the minimum utility value (line 14). This way, we ensure that the cache is not saturated. Without this mechanism, the utility would never decrease, which would prevent any new admission to the cache.

While processing the query q with Alg. 3, we may reuse the result of a cached query w. In this case, we update the utility of w according to Eq. 4 (line 17). To obtain the reuse cost of a cached query w regarding a query q, we measure the runtime to handle cached results, i.e., the first three cases (lines 4-15) of Alg. 3.

5.4 Further Considerations

Handling cache cold start. Initially, as the cache is empty, no results can be reused and query processing is slow due to the subgraph

Table 2: Statistics of the datasets.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	#Labels	Avg. degree
Yeast	3'101	12'519	71	8.07
Human	4'674	86'282	90	36.92
Wordnet	82'670	127'124	5	3.08
Cora	2'708	5'278	7	3.90
Citeseer	3'327	4'600	6	2.77
Pubmed	19'717	44'324	3	4.5

isomorphism search over the complete data graph. Therefore, we propose to populate the cache pro-actively in an offline phase with results from random subgraphs. That is, we randomly select k diverse nodes in the data graph that are far from each other. Starting with each of these node, we construct an ego-network which serves as a subgraph query for which the results are added to the cache. As the actual query stream is processed, we expect these surrogate queries to be evicted from the cache. We confirm the benefits of populating the cache in this manner with a dedicated experiment.

Minimizing distance computation overhead. To compute the distance from the new query to the cached queries (line 2 in Alg. 4), all cached queries need to be traversed. To reduce the induced overhead, we limit this computation to the k nearest neighbors of the query. While this yields solely an approximation of the cache diversity, in practice, the estimates are sufficiently accurate to make correct decisions about cache eviction and admission.

Result cardinality. Since each query may have a different number of matching subgraphs, the time to store the query results varies between queries. We therefore propose to store solely the first k matching subgraphs, which is akin to displaying the first k results in information retrieval systems. After examining these initial results, a users may decide whether the complete query answer shall be derived. If so, the former results are leveraged, similar to the third case of our approach to query stream processing in Alg. 3.

6. EVALUATION

In this section, we report on comprehensive evaluation experiments. We first clarify the experimental setup (§6.1), before turning to our approaches to construct embeddings for individual nodes (§6.2) and complete subgraphs (§6.3). We then evaluate the effectiveness of our policy for cache management (§6.5), before we close with an end-to-end comparison of our approach (§6.7).

6.1 Experimental setup

Datasets. We used six standard real-world benchmark datasets for subgraph isomorphism: Yeast, Human, Wordnet, Cora, Citeseer, and Pubmed. The first three datasets originate from [42, 32]. Yeast is a protein-protein-interaction (PPI) network with a small average degree, but a large number of labels. Human is also a PPI network, but with a large node degree. Wordnet is a graph capturing relations between English words. It has a small number of labels and a small node degree. The last three datasets were used in [28, 51] and denote citation networks. Nodes of these datasets are attributed. Statistics of the datasets are given in Table 2.

Baselines. We compare our approach against several baselines.

VF2[10]: is the traditional subgraph isomorphism search algorithm. It is an instance of the generic candidate filtering in Alg. 1, using only labels and the nodes' degrees as filtering criteria.

TurboISO[20]: is the state-of-the-art technique for single-query subgraph isomorphism search. It performs candidate search by constructing candidate regions which can be match with the query graphs. During the subgraph search, only candidates in the regions are considered, which reduces the running time significantly.

MQO[42]: is a state-of-the-art technique for multi-query subgraph isomorphism search. It processes queries in batches. For the queries in the same batch, common structures are identified. As a matching subgraph for a common structure can be used for all queries in the batch, this reduces the set of candidate nodes.

For graph indexing, we compare our embedding-based approach with structure-based indices such as GGSX [5] and CTIndex [29]. GGSX uses paths with bounded length as features to compare subgraphs. CTIndex identifies both paths and cycles of interest to create graph fingerprints. Both methods can be seen as manual feature engineering based on the graph structure, whereas our embedding-based approach derives features automatically.

For cache management, we compare our policy with LRU [57] and Recache [3], which are both instances of the Landlord algorithm. Recache manages a cache based on processing times, while LRU incorporates the last access time.

Query streams. Our setting of streaming subgraph isomorphism belongs to the class of multi-query subgraph problems. For such problems, it is a common evaluation strategy to generate queries randomly with parameters that control their overlap and repetition. Specifically, to generate query streams, we follow the generation process from [32, 42]. Given a number of subgraph families m, we randomly select m nodes from the data graph of each dataset. For each node, a core subgraph containing n nodes is derived by a random walk. Note that our generation process creates larger queries than those reported in [32, 42], as this process measured the graph size by the number of edges. Hence, we derive a more challenging query workload. We then create query streams by inserting, in each of them, a core subgraph, before iteratively adding other subgraphs. With probabilities a, b, c, we add a subgraph previously seen in the stream, in its original form (a), with nodes added (b), or with nodes removed (c). With probability d, we add a new core subgraph. This way, we simulate query streams of different characteristics.

Metrics. As our main metric, we measure the average processing time over the whole query stream. To compare approaches to cache management, we also assess the average hit rate.

Implementation and environment. We implemented our model for graph indexing in Python and used Pytorch for offline training. The online query evaluation was implemented in C++.

Our experiments were conducted on a workstation with a 2.4GHz CPU and 24GB RAM. We report average results over 20 experimental runs. Unless stated otherwise, we use a default setting of a query size of 10, a cache size of 20, a timeout of 100ms, a query stream of 1000 queries and an embedding size of eight.

In the following, we first analyze the effects of using embeddings in different components of our framework *independently*. In §6.2, we analyze the effects of embeddings in filtering candidate nodes and edges for subgraph isomorphism as discussed in Alg. 2. Next, §6.3 aims to validate the correctness of our subgraph embeddings in different aspects. In §6.4, we focus on analyzing the effectiveness of the learned embedding function and its training process. §6.5 is dedicated to the cache management in isolation, while 6.7 compares our framework with other baselines in an end-to-end manner.

6.2 Effectiveness of Embeddings in Pruning

To evaluate the benefit of using embeddings in pruning matching candidates for subgraph isomorphism, we construct a subgraph of size 20 for every node in the data graph. Hence, for every node and edge of a subgraph, we know their correct mappings. Then, we construct embeddings for all the nodes and rank the nodes of the data graph by their distance to each node of the subgraph. We repeat the same process for the edges. We measure the percentage of the reduction of candidate nodes and edges achieved by filtering based on embeddings. Fig. 6-A shows that using the embeddings, we are able to filter more than 70% of node candidates and 99% of edge candidates, which highlights the suitability of embeddings in this context. We also observe that edge embeddings are more discriminative than node embeddings, which shows the benefit of using subgraphs (even with only 2 nodes) as pruning criteria.

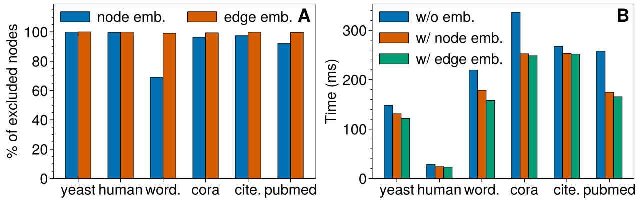

Figure 6: Effects of using embeddings.

We further enhance TurboISO with node and edge embeddings as a candidate filtering strategy (see Alg. 1). In Alg. 1, the more candidate structures are identified for each structure in the query graph, the more branches need to be considered in the search. We therefore evaluate the effectiveness of using embeddings by the total time required to find the matching subgraph in the data graph. Fig. 6-B shows that by adding embeddings as a filter strategy, we reduce the answering time for all datasets, e.g., from 219ms to 178ms to 157ms for the Wordnet dataset by using node and edge embeddings respectively. The observed benefits are relatively small for the Human dataset. The reason being the high label diversity of the dataset. If filtering based on labels is already very effective, further filtering with embeddings becomes negligible. Given the effectiveness of edge embeddings, in the following experiments, we construct subgraph embeddings using edge embeddings.

6.3 Evaluation of Subgraph Embeddings

Subgraph similarity vs. embedding distance. Next, we evaluate whether the embeddings of structurally-similar subgraphs are indeed close in the embedding space. To measure subgraph similarity, we use two metrics which are the size of the maximum common subgraph (MCS) and the subgraph edit distance. We create a pair of subgraphs with edit distance k by first constructing a two-hop ego graph g from a randomly-selected node in the data graph. We then remove $1 \leq k \leq 7$ edges randomly, so that the subgraph is still connected, to obtain a subgraph g'. As for the size of the MCS, for each dataset, we randomly extract subgraphs of size 15 from the data graph. We then select 5000 pairs of subgraphs randomly and compute the size of its maximum common subgraph. For each pair, we then construct their subgraph embeddings to measure their embedding distance. Our hypothesis is that there is a correlation between the MCS size and edit distances and the embedding distances.

Fig. 7 confirms this hypothesis. When the MCS size increases, the subgraph embedding distance increases as well. This observation is consistent over all datasets. The Pearson's correlation values in Table 3 confirm that there is a strong correlation between the MCS size and edit distances and the embedding distances. Hence, the subgraph embeddings indeed reflect the structural similarity.

Table 3: Pearson's correlation coefficients.

	Yeast	Human	Wordnet	Cora	Citeseer	Pubmed
MCS size	-0.95	-0.93	-0.96	-0.88	-0.87	-0.95
Edit distance	0.99	0.98	0.97	0.99	0.99	0.98

Subgraph embedding visualization. To analyze the subgraph embeddings qualitatively, we extract subgraphs from the data graph of the Human dataset, such that subgraphs belong to 10 families.

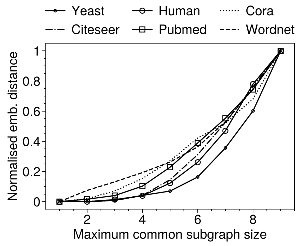

Figure 7: MCS size vs. embedding distance.

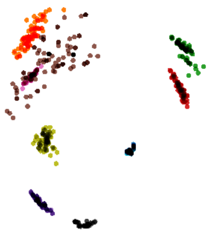

Figure 8: Visualization of subgraph embeddings.

Then, we construct the embeddings of these subgraphs and map them to two dimensions for visualization using PCA. Fig. 8 shows the subgraph embeddings with color-coded families, highlighting a clear clustering into the families. This highlights that the structural similarities of graphs in a family is captured well by the embeddings.

Search performance. To analyze the benefits of using subgraph embeddings for searching similar subgraphs, for each data graph, we create 20 random families of 50 subgraphs. Then, for each query, we use one subgraph to search for the others. We compare our approach of using subgraph embeddings with a traditional approach to search for similar subgraphs based on the graph structure.

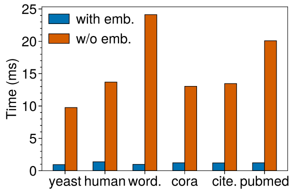

Figure 9: Search time.

Fig. 9 shows a large difference in the observed search times. There is a consistent improvement, over all datasets, when using embeddings, with the difference being at least 8ms. For instance, with subgraph embeddings, we can achieve a speedup of up to 25× on the Wordnet dataset. The reason being that searching for similar subgraphs is significantly faster in the embedding space. Structure-based approaches, in turn, require a costly exploration of the actual graph structure. Note that this experiment does not use any cache.

Effectiveness of indexing. We also compare our embedding-based index to structural indices, such as CTIndex or GGSX in terms of time required to create an index for a subgraph. Note that for our approach, this is the time required to construct a subgraph embedding *after* we already train the model (training time will be investigated in §6.4). In this experiment, we set the subgraph size to 15. The experimental results are shown in Table 4. There is a remarkable difference in the efficiency of structure-based approaches and our embedding-based index. CTIndex requires at least 17ms to create an index and on large datasets, such as Human, it would take 1.2s. GGSX performs significantly better than CTIndex with the indexing time staying below 4ms for most datasets. However, our method is an order of magnitude faster and constructs an index in around 0.02ms. These performance results illustrate another benefit of using embeddings in our context. Note that in this experiment, we evaluate the indexing component in isolation, without any caching. Hence, the observed differences stem exclusively from the use of embeddings.

Table 4: Comparison on indexing time (ms).

		Yeast	Human	Wordnet	Cora	Citeseer	Pubmed
CTIndex (0.5kB)		86.63	1235.45	17.48	59.45	76.68	48.5
GGSX	Time	3.085	528.44	0.395	1.867	3.307	1.142
	Space	14.01	232.8	0.25	1.18	1.46	0.57
Ours (0.5kB)		**0.021**	**0.0195**	**0.0219**	**0.022**	**0.022**	**0.023**

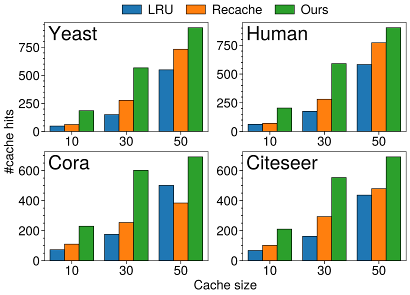

Figure 10: Cache size vs hits.

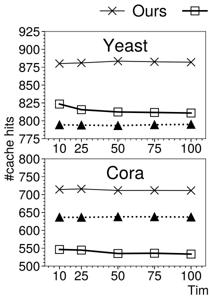

Figure 11: Caching strategy vs hits.

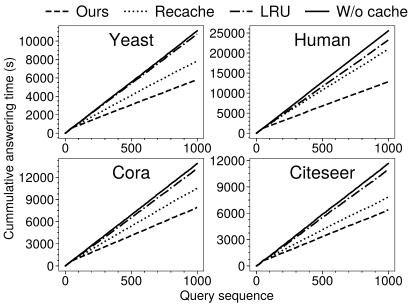

Figure 12: Caching strategy vs time.

Table 5: Time required to compare 1000 subgraph pairs (ms).

	Sub. size	Yeast	Human	Wordnet	Cora	Citeseer	Pubmed
WL	10 nodes	38.89	24.56	22.74	26.42	28.01	22.16
	20 nodes	85.56	48.54	28.05	45.95	52.76	37.62
MPNN		**1.02**	**1.02**	**1.03**	**1.01**	**1.01**	**1.02**

Turning to space requirements, we first note that we used a fixed embedding size for both CTIndex and our embeddings with a similar index size for a fair comparison. As such, CTIndex and our index both require 0.5kB space. The index size of GGSX, in turn, depends on the data graph, see Table 4, and ranges from 0.25kB to 232.8kB. We conclude that our index has comparable size to other approaches, but can be constructed much quicker.

6.4 Evaluation of Parameterized Subgraph Isomorphism

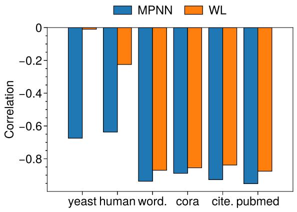

Figure 13: WL vs. MPNN (lower is better).

WL vs. MPNN. We compare the quality of embeddings generated by WL and our MPNN by measuring the correlation between their embedding distance with the subgraph similarity. We randomly extract arbitrary subgraphs from the data graph for each dataset. This is likely to create subgraphs that are *not* observed in the data graph as these subgraphs may not be induced subgraphs. Then, for each pair from a randomly-selected set of subgraph pairs, we measure the size of their MCS. Fig. 13 confirms our hypothesis that WL would perform poorly on subgraphs that are observed in the data graph. MPNN outperforms WL for all datasets, with large differences emerging for the Yeast and Human datasets. This is attributed to a high label diversity in both datasets (see Table 2), so that the generated subgraphs are more likely to be different from ones in the data graphs. As WL relies on statically determined hash functions, unseen combinations of labels cannot be handled. For the other datasets, the labels are more homogeneous, so that most of the label combinations are already available in the data graphs. As a result, the performance of WL is only slightly worse than the one of MPNN.

To make WL able to handle unseen subgraphs, one may ignore the hash functions and measure subgraph similarity based on their multiset representations (see §3.2). We measure the total time required to compare the similarity for each pair of subgraphs generated as detailed above. Table 5 shows that comparing subgraphs based on MPNN embeddings is significantly faster for all datasets, though. Another interesting observation is that comparing using the MPNN embedding is robust against changes in the subgraph sizes. For the case of WL, larger subgraphs require more time for comparison. The reason is that MPNN embeddings have a fixed size, whereas WL uses a symbolic representation, in which the representation size increases as more combinations of labels are present.

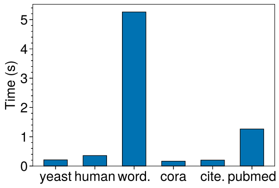

Figure 14: Training time.

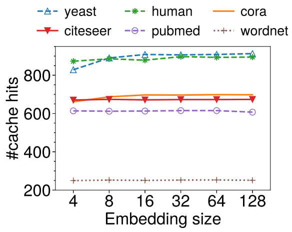

Figure 15: Robustness.

Training time. To measure the training time, we report the time to train one epoch of our model. The number of training epochs can be considered as normalized training time independent of any infrastructure. From Fig. 14, we observe that the training time per epoch is very small. The longest training time is around 5s for the Wordnet dataset. We further observed that the loss converges after around 10 epochs. This means that the total training time to obtain a good model is at most 60s. Hence, even with short training time, we have already obtained a high-quality embedding model.

Sensitivity to number of parameters. Exploring the effect of the number of parameters in our model, we vary the number of parameters by changing the embedding size, as they are closely related. We measure the model's performance by the number of cache hits. Fig. 15 illustrates that a larger embedding size tends to lead to more cache hits. Yet, the improvement becomes small after an embedding size of 16. We conclude that our model is relatively robust against the number of parameters, while having a small embedding size is commonly sufficient to achieve good performance.

6.5 Effectiveness of Cache Management

Having evaluated the benefits of using embeddings without any caching, the next set of experiments consider our caching policy.

Effects of cache size. We measure the number of cache hits for different policies when varying the cache size from 10 to 50. Fig. 10 indicates that, as the cache size increases, all methods are able to obtain more cache hits, as expected. In general, our cache management strategy outperforms both Recache and LRU, while Recache tends to yield better results than LRU. For instance, using the Human dataset with a cache size of 30, Recache improves over LRU with the difference being 200 hits, while our technique adds further 180 cache hits. Recache performs better than LRU as it considers differences in the queries' answer time. We conclude that our approach of making the cache diverse through a diversity-based notion of utility helps in achieving more cache hits.

Effects of timeout threshold. In this experiment, we analyze our caching policy when varying the timeout threshold for a query from 0.1 to 1 second. Similar to the above experiment, we compare our approach with LRU and Recache and measure the number of cache hits. The results in Fig. 11 show that our method performs best. For instance, we observe an improvement of 14% and 31% on the Cora dataset over Recache and LRU, respectively.

Evaluation for query streams. Next, we assess the effectiveness of our caching policy when queries arrive as a stream. Fig. 12 confirms the observations made for the case of a single query. That is, our policy consistently outperforms the baselines over all datasets. Compared with using no caching at all, an LRU policy, and Recache, our strategy leads to improvements of 74%, 42%, and 19%, respectively, on the Human dataset at 900 queries. A key observation is that the gap between our strategy and the baselines widens as we process more queries. Hence, using our strategy becomes more beneficial over time, due to increasing cache effectiveness.

Cache initialization. Our technique to alleviate the cold-start problem is evaluated in Fig. 16 for the Pubmed dataset. The cumulative answer time for the first 50 queries shows that, without using cache initialization, the answer time increases linearly. With our initialization strategy, we are able to reduce the total answer time by at least 20%. Note that our cache initialization strategy is enabled by subgraph embeddings, since those support fast cache search and the identification of partial matches. Cache initialization that would rely on structural similarity search, in turn, would imply a significant overhead, i.e., cache initialization may not pay off.

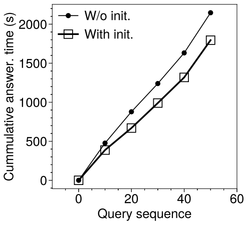

Figure 16: Cache init.

6.6 Workload Evaluation

The following experiments evaluate our framework with respect to different query stream workloads. This requires changing different parameters in our query stream generation, see §6.1. Specifically, we report results on varying the query overlap and repetition.

Effects of query overlapping. We control the overlap of queries by varying the probability of adding subgraphs with added nodes (parameter b in §6.1). This changes the percentage of overlapping queries in the query streams. The results in Fig. 17B show that, as the number of overlapping queries increases, the average time required to answer a query decreases for all datasets. This is expected as more overlapping queries mean higher reusability, which leads to better cache hits and reduced answering time.

Effects of query repetition. We further vary the number of query families in the streams. The more families are present, the higher the diversity of the stream. Fig. 17A shows that a query stream having more families leads to increased answer time. However, the rate with which it increases is small among all datasets, which shows a certain robustness of our approach to high stream diversity.

6.7 End-to-end Comparison

After we evaluated the individual building blocks of our solution to streaming subgraph isomorphism, we analyze its end-to-end performance, also in comparison to other techniques.

Comparative analysis. We first compare our approach with traditional subgraph isomorphism techniques. Table 6 lists the overall processing times observed for the different datasets. VF2, which is the traditional approach to subgraph isomorphism, has the worst performance. TurboISO, which employs more advanced methods to filter candidate nodes leads to a smaller processing time. It often also performs better than MQO, which is a batch-processing technique. However, across all datasets, our technique leads to processing times that are significantly lower.

Table 6: Comparison of different subgraph isomorphism search techniques in terms of overall processing time.

	Yeast	Human	Wordnet	Cora	Citeseer	Pubmed
VF2	45.3	33.7	498.2	60.7	66.2	253.1
TurboISO	12.4	29.6	139.5	15.7	12.7	51.4
MQO	16.2	15.4	315.4	29.8	30.1	80.7
Ours	**3.7**	**7.5**	**114.1**	**9.6**	**7.1**	**31.1**

The results are interesting in particular in relation to MQO. The latter constructs structural indices of queries in the same batch and conducts the search directly on the graph structure. As such, it is similar to our technique in terms of aiming at reuse through indexing. Yet, by relying on embeddings, our approach requires significantly less time to process a query on average.

To provide further insights to the differences between our approach and MQO, we measure the time required to construct the indices (i.e., the induced overhead) and the actual time used for graph search. Fig. 19 shows that, on a simple dataset such as Yeast, our approach incurs more overhead compared to MQO, as the query embeddings need to be constructed and the cache is searched. Yet, the overhead pays off, as our approach spends less time in the actual search in comparison to MQO. For a complex dataset such as Wordnet, both, the overhead and the search time are lower for our approach. This is due to the inherent complexity of the structure-based approach of MQO.

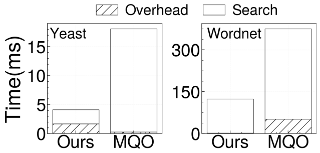

Figure 19: Time break-down.

Effects of query size. Finally, we analyze the impact of the query size on techniques for subgraph isomorphism search. We vary the query size from 10 to 25 and measure the average processing time. Here, our method shows better answering times than the baseline, see Fig. 18. As expected, the response time increases as the query size increases. However, for our method, the respective rate is smaller or on par with the best baseline.

7. RELATED WORK

Single-query subgraph isomorphism. Many approaches have been developed to answer a single subgraph isomorphism query, by leveraging structural equivalence between the query graph and the data graph. The mapping is usually constructed iteratively, preserving the nodes' connectivity. VF2 [10] is the traditional search algorithm. It instantiates the generic approach to candidate filtering as formalized in Alg. 1 and constructs a mapping by comparing the labels and the nodes' degrees. Moreover, WaSQ [33] performs rewriting of the query graph to match structures for which partial mappings are known already. Based thereon, the final mapping is derived from the partial one.

While our focus has been on streaming subgraph isomorphism, our approach may also be used to answer a single query. The presented embeddings of nodes can be seen as an additional data structure to support subgraph search. They serve a similar role as node labels or node degrees employed by existing techniques to filter the set of candidate nodes in the construction of a mapping.

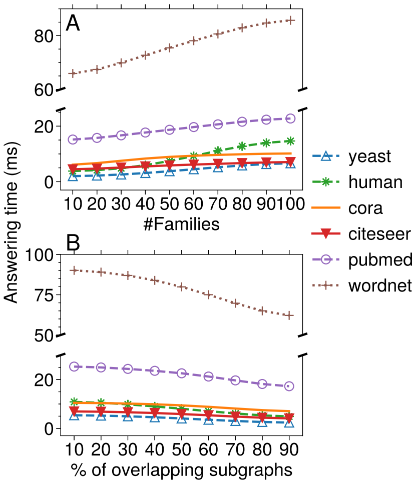

Figure 17: A) Query repetition; B) Query overlap.

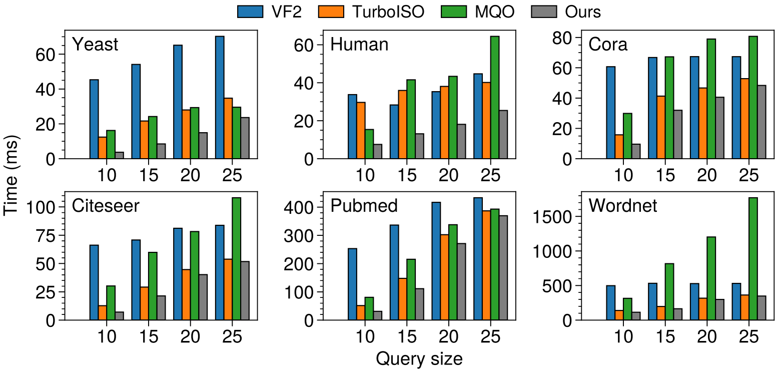

Figure 18: Effects of query size.

Another way to speed up subgraph isomorphism is to identify features that are representative of subgraphs for comparison [5, 29, 4, 26]. These features can be considered as subgraph indices similar to our subgraph embeddings. CPI [4] constructs a data structure called compact path index, which is similar to a spanning tree, to support subgraph search. GGSX [5] considers paths with bounded lengths as features. These paths are stored in a suffix tree to speed up the search. In addition to paths, CTIndex [29] also uses cycles to create graph fingerprints. There are several problems with structure-based indices, though. First, they are not efficient as identifying structure or motifs in graphs is often a subgraph isomorphism problem in itself. Second, they often require users to identify which structures are relevant to the subgraph isomorphism problem. This is a tedious task as these structures are typically specific to a graph. Our embedding-based index avoids these issues as the embeddings can be learned efficiently and capture the graph structure in an automatic manner.

Multiple-query subgraph isomorphism. Recently, the problem of answering multiple subgraph isomorphism queries at the same time has attracted attention. The multi-query setting enables the identification of common structures among the queries, which provides an angle for optimization as exemplified for SPAQRL queries over RDF graphs in [31]. This approach divides queries into groups based on their edge labels, Jaccard similarity, and benefits for batch optimization. Then, a common subgraph pattern is extracted per group and the queries are rewritten to comprise the pattern and optional constraints. A query engine that supports such optional constraints is used to answer the original queries. MQO [42] tackles the multi-query subgraph isomorphism search for general graphs. MQO also groups query graphs that are structurally similar. However, MQO organizes them in a containment tree, called pattern containment map (PCM), in which a directed edge connects queries, where one is a subgraph of the other. Queries are then answered in a top-down manner with respect to the PCM. This allows MQO to use mappings of parent queries to derive answers for their child queries. While MQO is close to our work in terms of striving for reuse in subgraph isomorphism search, there are several important differences. First, MQO is cache-oblivious, whereas our approach heavily relies on caching. Caching makes it possible to reuse not only immediate results (e.g., in the same batch of queries), but the results from all past queries can, potentially, be reused. Second, MQO is a batch processing algorithm that processes one set of queries after another one. We presented a proper stream processing algorithm. Applying MQO over streams would require to partition the stream into batches, which is not practical. Third, MQO exploits the structure of query graphs in the same batch to identify similar queries, which is inefficient time-consuming. By leveraging subgraph embeddings, our approach handles query graphs much more efficiently.

Orthogonal to our work is TurboFlux, a subgraph isomorphism system for handling a streaming data graph [27]. Here, a standing query is posed against a data graph for which the structure changes over time. This is the mirrored case of our setting, in which the data graph is static and a continuous stream of queries needs to be evaluated. Another related problem is multi-data-graph subgraph isomorphism search [60, 61]. This problem is about finding mappings of a query, not for one, but several data graphs. We foresee that our approach can be extended to this setting by learning an embedding for each data graph.

Network representation learning. Techniques to construct graph embeddings are shallow or deep [56, 55, 23, 22, 54, 53, 46, 47, 52, 45, 59, 9, 38, 8]. Shallow approaches [17, 41] start from random embeddings and move them in a high-dimensional space, such that a loss function based on the distance between the embeddings is minimized. Deep approaches such as GNNs [28, 16, 19] start from the node features and perform message passing to minimize a loss function while updating the embedding. This way, the embedding of a node also incorporates its neighbors' features. While node and graph embeddings are well-studied problems, research on subgraph embedding is still in its infancy. Approaches such as subgraph2vec [37] are shallow. Hence, they cannot create subgraph embeddings for graphs that are not known a priori, i.e., the queries in streaming subgraph isomorphism. To overcome this limitation, we presented an approach based on truncated message-passing, which is a first deep embedding technique for subgraphs.

8. CONCLUSION

In this paper, we proposed an approach to handle subgraph isomorphism search for streams of queries. Based on advances in subgraph representation learning, we proposed a novel graph indexing technique. This index provides the foundation for our approach to streaming graph isomorphism that exploits caching and reuse of query results. Moreover, we presented a new policy for cache management that assesses the utility of a query not only based on processing time, but incorporates a notion of reusability. Experiments with several real-world datasets confirm the efficiency of our approach and the effectiveness of our design choices.

9. REFERENCES

[1] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *ICDE*, pages 821–831. IEEE, 2003.

[2] A. Archer, K. Aydin, M. H. Bateni, V. Mirrokni, A. Schild, R. Yang, and R. Zhuang. Cache-aware load balancing of data center applications. 12(6):709–723, 2019.

[3] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: Reactive caching for fast analytics over heterogeneous data. *VLDB*, 11(3):324–337, 2017.

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

[5] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *IAPR-PRIB*, pages 195–203. Springer, 2010.

[6] E. Castrillo, E. León, and J. Gómez. Dynamic structural similarity on graphs. *arXiv preprint arXiv:1805.01419*, 2018.

[7] ChemSpider. Chemspider data sources. www.chemspider.com/DataSources.aspx, 2020.

[8] T. Chen, H. Yin, H. Chen, R. Yan, Q. V. H. Nguyen, and X. Li. Air: Attentional intention-aware recommender systems. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 304–315, 2019.

[9] T. Chen, H. Yin, Q. V. H. Nguyen, W.-C. Peng, X. Li, and X. Zhou. Sequence-aware factorization machines for temporal predictive analytics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1405–1416, 2020.

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

[11] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars. Orthogonal range searching: Querying a database. *Computational Geometry*, pages 95–120, 2008.

[12] C. T. Duong, T. D. Hoang, H. T. H. Dang, Q. V. H. Nguyen, and K. Aberer. On node features for graph neural networks. *arXiv preprint arXiv:1911.08795*, 2019.

[13] C. T. Duong, H. Yin, D. Hoang, M. H. Nguyen, M. Weidlich, Q. V. H. Nguyen, and K. Aberer. Graph embeddings for one-pass processing of heterogeneous queries. In *ICDE*, pages 1994–1997, 2020.

[14] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. *VLDB*, 1(1):550–561, 2008.

[15] S. Ghandeharizadeh and J. Yap. Cache augmented database management systems. In *DBSocial*, pages 31–36, 2013.

[16] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, pages 1263–1272, 2017.

[17] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, pages 855–864, 2016.

[18] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[19] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1024–1034, 2017.

[20] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.

[21] X. Huang, J. Lai, and S. F. Jennings. Maximum common subgraph: some upper bound and lower bound results. *BMC bioinformatics*, 7(4):S6, 2006.

[22] N. Q. V. Hung, H. Jeung, and K. Aberer. An evaluation of model-based approaches to sensor data compression. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2434–2447, 2012.

[23] N. Q. V. Hung, D. C. Thang, M. Weidlich, and K. Aberer. Minimizing efforts in validating crowd answers. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 999–1014, 2015.

[24] T. T. Huynh, C. T. Duong, T. H. Quyet, Q. V. H. Nguyen, A. Sattar, et al. Network alignment by representation learning on structure and attribute. In *PRICAI*, pages 698–711, 2019.

[25] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *ACM Sigmod Record*, 26(2):369–380, 1997.

[26] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *VLDB*, 8(12):1566–1577, 2015.

[27] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *SIGMOD*, pages 411–426, 2018.

[28] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *NIPS*, 2016.

[29] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*, pages 1115–1126. IEEE, 2011.

[30] P.-A. Larson, J. Goldstein, and J. Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE*, pages 177–188. IEEE, 2004.

[31] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *ICDE*, pages 666–677. IEEE, 2012.

[32] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *VLDB*, 6(2):133–144, 2012.

[33] Y. Liang and P. Zhao. Workload-aware subgraph query caching and processing in large graphs. In *ICDE*, pages 1754–1757. IEEE, 2019.

[34] D. Marx and M. l Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism. 2014.

[35] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI*, volume 33, pages 4602–4609, 2019.

[36] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[37] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *arXiv preprint arXiv:1606.08928*, 2016.

[38] T. T. Nguyen, M. Weidlich, H. Yin, B. Zheng, Q. H. Nguyen, and Q. V. H. Nguyen. Factcatch: Incremental pay-as-you-go fact checking with minimal user effort. In *Proceedings of the*

43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 2165–2168, 2020.

[39] R. Ozcan, I. S. Altingovde, and Ö. Ulusoy. Cost-aware strategies for query result caching in web search engines. *TWEB*, 5(2):1–25, 2011.

[40] G. Paschos, G. Iosifidis, and G. Caire. Cache optimization models and algorithms. *arXiv preprint arXiv:1912.12339*, 2019.

[41] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *KDD*, pages 701–710, 2014.

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

[43] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *VLDB*, 1(1):364–375, 2008.

[44] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, pages 1–8. IEEE, 2008.

[45] K. Sun, T. Qian, T. Chen, Y. Liang, Q. V. H. Nguyen, and H. Yin. Where to go next: Modeling long-and short-term user preferences for point-of-interest recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 214–221, 2020.

[46] N. T. Tam, M. Weidlich, D. C. Thang, H. Yin, and N. Q. V. Hung. Retaining data from streams of social platforms with minimal regret. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 2850–2856, 2017.

[47] N. T. Toan, P. T. Cong, N. Q. V. Hung, and J. Jo. A deep learning approach for early wildfire detection from hyperspectral satellite images. In *2019 7th International Conference on Robot Intelligence Technology and Applications (RiTA)*, pages 38–45, 2019.

[48] H. T. Trung, N. T. Toan, T. Van Vinh, H. T. Dat, D. C. Thang, N. Q. V. Hung, and A. Sattar. A comparative study on network alignment techniques. *ESWA*, 140:112883, 2020.

[49] H. T. Trung, T. Van Vinh, N. T. Tam, H. Yin, M. Weidlich, and N. Q. V. Hung. Adaptive network alignment with unsupervised and multi-order convolutional networks. In *ICDE*, pages 85–96, 2020.

[50] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[51] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm. Deep graph infomax. *ICLR*, 2018.

[52] Q. Wang, H. Yin, H. Wang, Q. V. H. Nguyen, Z. Huang, and L. Cui. Enhancing collaborative filtering with generative augmentation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 548–556, 2019.

[53] W. Wang, H. Yin, Z. Huang, Q. Wang, X. Du, and Q. V. H. Nguyen. Streaming ranking based recommender systems. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 525–534, 2018.

[54] H. Yin, H. Chen, X. Sun, H. Wang, Y. Wang, and Q. V. H. Nguyen. Sptf: a scalable probabilistic tensor factorization model for semantic-aware behavior prediction. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 585–594, 2017.

[55] H. Yin, Z. Hu, X. Zhou, H. Wang, K. Zheng, Q. V. H. Nguyen, and S. Sadiq. Discovering interpretable geo-social communities for user behavior prediction. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 942–953, 2016.

[56] H. Yin, X. Zhou, B. Cui, H. Wang, K. Zheng, and Q. V. H. Nguyen. Adapting to user interest drift for poi recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2566–2581, 2016.

[57] N. E. Young. Online paging and caching. 2008.

[58] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, 2009.

[59] S. Zhang, H. Yin, Q. Wang, T. Chen, H. Chen, and Q. V. H. Nguyen. Inferring substitutable products with deep network embedding. In *IJCAI*, pages 4306–4312, 2019.

[60] P. Zhao, J. X. Yu, and S. Y. Philip. Graph indexing: Tree+ delta>= graph. In *VLDB*, volume 7, pages 938–949, 2007.

[61] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng. Answering top-k graph similarity queries in graph databases. *TKDE*, 2019.