

# PATTERN MORPHING for Efficient Graph Mining

Kasra Jamshidi

School of Computing Science  
Simon Fraser University  
British Columbia, Canada  
kjamshid@cs.sfu.ca

Keval Vora

School of Computing Science  
Simon Fraser University  
British Columbia, Canada  
keval@cs.sfu.ca

## Abstract

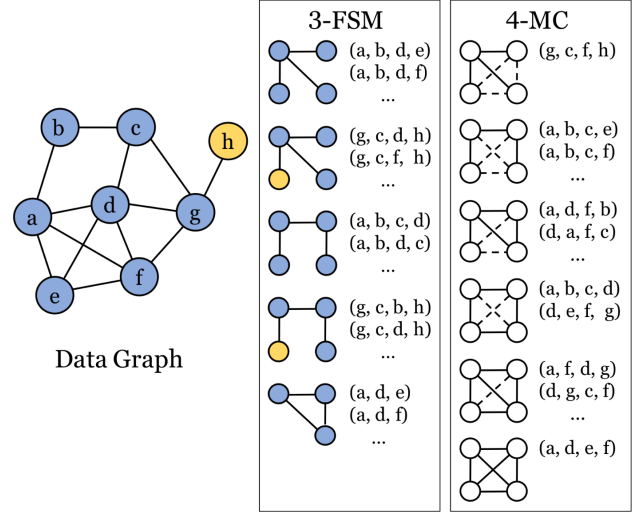
Graph mining applications analyze the structural properties of large graphs, and they do so by finding subgraph isomorphisms, which makes them computationally intensive. Existing graph mining techniques including both custom graph mining applications and general-purpose graph mining systems, develop efficient execution plans to speed up the exploration of the given query patterns that represent subgraph structures of interest.

In this paper, we step beyond the traditional philosophy of optimizing the execution plans for a given set of patterns, and exploit the sub-structural similarities across different query patterns. We propose PATTERN MORPHING, a technique that enables structure-aware algebra over patterns to accurately infer the results for a given set of patterns using the results of a completely different set of patterns that are less expensive to compute. Pattern morphing *morphs* (or converts) a given set of query patterns into alternative patterns, while retaining full equivalency. It is a general technique that supports various operations over matches of a pattern beyond just counting (e.g., support calculation, enumeration, etc.), making it widely applicable to various graph mining applications like Motif Counting and Frequent Subgraph Mining. Since pattern morphing mainly transforms query patterns before their exploration starts, it can be easily incorporated in existing general-purpose graph mining systems. We evaluate the effectiveness of pattern morphing by incorporating it in Peregrine, a recent state-of-the-art graph mining system, and show that pattern morphing significantly improves the performance of different graph mining applications.

## 1 Introduction

With growing interest in analyzing graph-based data, graph mining applications are being widely used across several domains including bioinformatics, computer vision, malware detection, and social network analysis [10, 14, 33, 34, 38, 41, 49, 50].


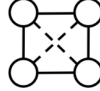


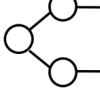

Several general-purpose graph mining systems have been developed in literature [7, 8, 12, 24, 31, 44, 48]. These systems provide an efficient runtime that explores the subgraphs of interest along with a high-level programming model to guide the exploration process and perform computations based on the explored subgraphs. Recent systems like Peregrine [24]



**Figure 1.** Example of vertex-induced patterns v/s edge-induced patterns in graph mining applications. 3-FSM typically explores (labeled) edge-induced matches, and there are three size-3 pattern topologies (i.e., topologies containing 3 edges) which form five patterns with distinct labelings from the data graph. 4-MC typically explores (unlabeled) vertex-induced matches, and there are six size-4 patterns (i.e., patterns containing 4 vertices).

take a pattern-aware approach where the graph mining applications are expressed as pattern matching sub-tasks, and the exploration plan directly finds the matching subgraphs, without performing expensive isomorphism and automorphism checks for every match.

With such an approach, details about how to match the patterns naturally become part of the pattern itself. For instance, Figure 1 shows the difference in the patterns expressed for Frequent Subgraph Mining (FSM) and Motif Counting (MC) applications. Since Frequent Subgraph Mining typically computes frequencies based on *edge-induced* matches (i.e., subgraphs that match all the edges in the pattern), the three size-3 topologies (i.e., topologies containing 3 ‘edges’) are simply expressed by the edges that are present in the pattern. Motif Counting, on the other hand, usually counts the occurrences of *vertex-induced* matches (i.e., subgraphs that match all the edges between vertices in the data graph), and hence the six size-4 patterns (i.e., patterns containing 4 ‘vertices’) contain

	4-Cycle		Chordal 4-Cycle		5-Cycle	
	Edge-Induced	Vertex-Induced	Edge-Induced	Vertex-Induced	Edge-Induced	Vertex-Induced
						
Mico	2.09s	<b>1.89s</b>	<b>0.08s</b>	3.04s	258.90s	<b>23.56s</b>
YouTube	<b>27.84s</b>	31.91s	<b>1.62s</b>	8.07s	111.89s	<b>67.03s</b>

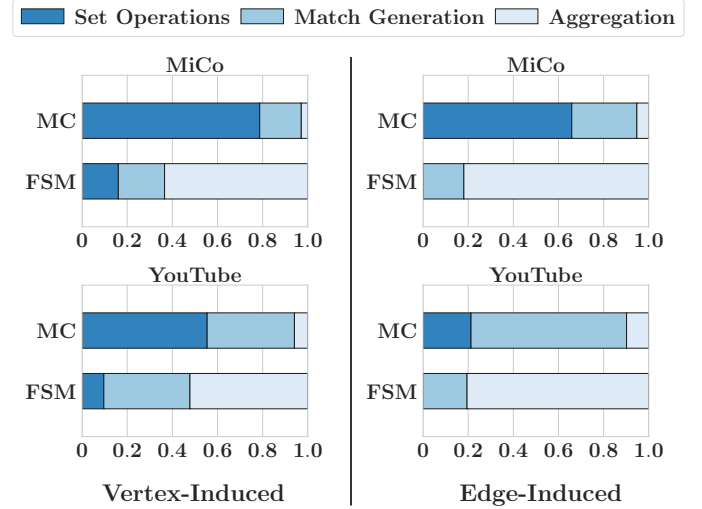
**Table 1.** Execution times (in seconds) for matching various patterns in Peregrine [24].

*anti-edges* [24] indicating that edges should not be present between those vertices. General-purpose graph mining systems often provide flexibility to use different exploration models for different workloads: for instance, FSM can be performed in edge-induced manner or vertex-induced manner on Peregrine [24], and switching between the two requires just a single line change in its FSM program.

The performance of graph mining systems is mainly dependent on three factors: (1) how many (partial and complete) matches are present in the data graph (i.e., the structure of the data graph); (2) how expensive it is to match the given set of patterns (i.e., the structure of the patterns); and, (3) what to do once the matches have been found (i.e., the application semantics). Figure 2 shows the performance breakdown of 3-FSM and 4-MC on two graphs; as we can see, the majority of time in MC is spent on finding matches and little time is taken for its counting aggregation, whereas the performance of FSM is dominated by performing aggregation (computing pattern supports [6]) over the matches. Hence, it is crucial to reduce both the time taken for matching, and the time taken to perform aggregation in order to enable high performance graph mining.

Existing systems already incorporate efficient pattern matching algorithms; however, they limit their analysis to only those set of patterns that are explicitly requested by the mining application. In this work, we aim to exploit the structural relationships between different patterns to accelerate the overall mining task. To understand how we achieve that, first we describe a key insight.

**Observations & Key Insight.** Table 1 shows the time taken for matching three patterns in vertex-induced and edge-induced manner on two data graphs. As we can see, there is no correlation between performance of edge-induced patterns and their respective vertex-induced variants; for instance, matching the edge-induced chordal 4-cycle is faster than matching the vertex-induced chordal 4-cycle, whereas matching the edge-induced 5-cycle is slower than matching its vertex-induced variant. This is because even though anti-edges provide pruning benefits for vertex-induced patterns at each exploration step, enforcing them using set differences can be more expensive than performing set intersections in edge-induced variants. Furthermore, two patterns appearing



**Figure 2.** Performance of Frequent Subgraph Mining (FSM) and Motif Counting (MC) on Mico [13] and YouTube [9] graphs in Peregrine [24] with vertex-induced and edge-induced exploration.

structurally similar to each other can end up taking drastically different amounts time; for instance, an edge-induced 4-cycle and an edge-induced chordal 4-cycle differ by only one edge, and still matching the former is over 26× slower than matching the latter.

These drastic variations in performance across structurally similar patterns present opportunities to perform analysis across different patterns, including those that are not explicitly requested by the mining application. This means *the structural similarities between different patterns can be exploited to infer the results for a given set of patterns using the results of a different set of patterns that are relatively inexpensive to compute.*

**Pattern Morphing.** In this paper, we propose PATTERN MORPHING, a technique that enables structure-aware algebra over patterns to *morph* (or convert) the query patterns into alternative patterns, which can then be used to quickly compute accurate final results for the original query patterns. We first formalize the semantics of pattern morphing to understand how patterns can be morphed into alternative patterns while retaining full equivalency with the original patterns so

that correct final results can be guaranteed. Then we develop an automatic pattern morphing engine that reformulates the original graph mining query into a query on the morphed (alternative) patterns, hence exploiting alternative matching plans that deliver different performance.

Our pattern morphing theory is general, and captures the application semantics in form of operations to be performed based on the matched subgraphs. This not only enables simple counting based applications like Motif Counting, but also supports complex aggregation types like MNI support computations [6] required in Frequent Subgraph Mining.

Since any given set of patterns can be morphed into different but equivalent alternative patterns, selecting the right alternative pattern sets is crucial to deliver high performance. This is addressed by developed a *pattern morphing optimizer* that minimizes the cost of pattern sets to construct the best alternative pattern sets, where the cost of a given pattern set is based on system-specific nuances (e.g., exploration plans), application-specific operations (e.g., counting, computing support, etc.), and details of the data graph.

**Results.** To demonstrate the generality and effectiveness of graph morphing, we integrated the pattern morphing engine with Peregrine [24]<sup>1</sup>, a recent state-of-the-art graph mining system. Our evaluation on different graph datasets and graph mining applications shows that pattern morphing accelerates the graph mining applications by up to 11.79×, which is a significant benefit since it is on top of Peregrine’s pattern-aware runtime.

**Other Applications of Pattern Morphing.** Graph morphing enables structure-aware algebra over patterns, and hence it is a generic technique to leverage structural relationships between different patterns. While in this paper we use graph morphing to accelerate general-purpose graph mining, graph morphing can be used for various other applications that inherently operate on graph sub-structures like incremental mining, approximate graph computations, mining on graph streams, interactive graph exploration, graph compression, and several others. The theory developed in this paper can be further enhanced with details like the application setup, execution environment, etc. to leverage it across those other applications.

**Evolution of Pattern Morphing Project.** With pattern-awareness deeply embedded in Peregrine, we had implemented custom pattern transformation rules designed specifically for motif counting since its inception (i.e., it has been publicly available since the first commit d1d3df9 on April 28, 2020). This paper generalizes those transformations as the pattern morphing technique to develop its well-defined theory and enable its wider applicability to various graph mining applications beyond just motif counting.

<sup>1</sup>Peregrine source code available here: <https://github.com/pdclab/peregrine>

## 2 Preliminaries

We build on the graph mining terminology from [24].

**Graph Isomorphism.** Given a graph  $g$ ,  $V(g)$  and  $E(g)$  denote its set of vertices and edges. If  $g$  is a labeled graph,  $L(g)$  denotes its set of labels.

An *isomorphism* between graphs  $g$  and  $h$  is a bijective mapping  $f : V(g) \rightarrow V(h)$  that preserves edge relationships: if  $(u, v) \in E(g)$ , then  $(f(u), f(v)) \in E(h)$ . If there is an isomorphism between  $G$  and  $H$  we say they are *isomorphic*. This is an equivalence relation.

A *subgraph isomorphism* from a graph  $h$  to a graph  $g$  is an injective mapping  $f : V(h) \rightarrow V(g)$  that preserves edge relationships: if  $(u, v) \in E(h)$ , then  $(f(u), f(v)) \in E(g)$ . We will re-define subgraph isomorphism in terms of graph patterns below.

**Pattern Semantics.** Graph mining applications involve finding subgraphs of interest in a given input graph (i.e., data graph). These subgraphs of interests are represented using *patterns*. A pattern is a simple connected graph, possibly with labels on its vertices. Apart from regular edges between vertices, a pattern can contain special edges called *anti-edges* (originally defined in [24]). An anti-edge indicates disconnections between pairs of vertices. We write  $A(p)$  to denote the set of anti-edges of a pattern  $p$ . Note that  $E(p)$  and  $A(p)$  are disjoint.

We re-define subgraph isomorphism for patterns to take anti-edges into account. A subgraph isomorphism from a pattern  $p$  to a graph  $g$  is an injective mapping  $f : V(p) \rightarrow V(g)$  that preserves edge and anti-edge relationships: if  $(u, v) \in E(p)$ , then  $(f(u), f(v)) \in E(g)$  and if  $(u, v) \in A(p)$ , then  $(f(u), f(v)) \notin E(g)$ . A subgraph isomorphism from a pattern into a graph is often called a match for  $p$ .

We also consider subgraph isomorphisms between patterns. A subgraph isomorphism from a pattern  $p$  to a pattern  $q$  is an injective mapping  $f : V(p) \rightarrow V(q)$  that preserves edge and anti-edge relationships. If  $(u, v) \in E(p)$  then  $(f(u), f(v)) \in E(q)$  and if  $(u, v) \in A(p)$  then  $(f(u), f(v)) \in A(q)$ .

Using these definitions, anti-edges can capture how subgraphs are mapped to patterns. A vertex-induced pattern (denoted as  $p^V$ ) is a pattern containing anti-edges between all of its vertex-pairs that are not connected by a regular edge. Hence, vertex-induced patterns find subgraphs containing a set of vertices and all the edges incident between them. On the other hand, an edge-induced pattern (denoted as  $p^E$ ) is a pattern without any anti-edge. Edge-induced patterns find subgraphs containing a set of edges and their endpoints. Note that fully connected patterns (i.e. *cliques*) are simultaneously edge-induced and vertex-induced since all their vertex-pairs are adjacent. Throughout the paper, we omit the superscript on pattern variables when the discussion applies to both vertex-induced and edge-induced patterns.

Finally, a subpattern of a pattern  $p$  is a pattern  $q$  for which there exists a subgraph isomorphism from  $q$  into  $p$ . Conversely, a superpattern of a pattern  $p$  is a pattern  $q$  such that  $p$  is a subpattern of  $q$ .

**Graph Mining Applications.** There are several graph mining applications like Frequent Subgraph Mining, Motif Counting, Pattern Matching, Clique Finding, and others, that require extracting subgraphs of interest in a given data graph. These applications perform different operations over the extracted subgraphs including counting, listing (or enumerating), computing support measures, etc. A summary of these applications, along with efficient pattern programs to perform them can be found in [24]. Below we summarize important details for two applications that exemplify some of the differences across different applications.

— *Motif Counting.* This problem involves counting the occurrences of all *motifs* up to a certain size in a data graph, where motifs are connected, unlabeled graph patterns. Typically, the exploration is performed in vertex-induced manner, expressed using vertex-induced patterns. Since the aggregation operation over matches is a simple *sum* operation over  $n$ -bit integral values, it requires constant amount of work for each match.

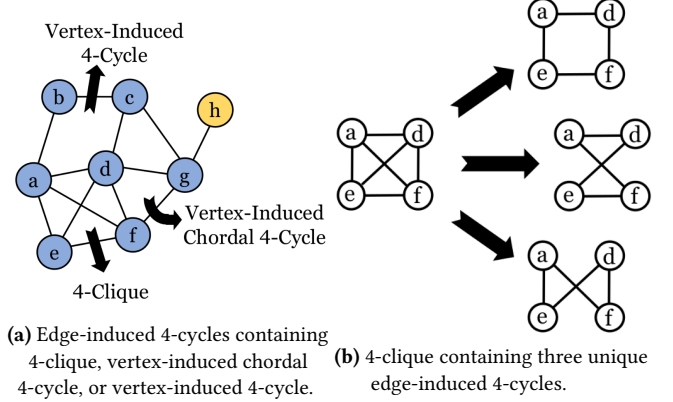
— *Frequent Subgraph Mining (FSM).* This problem involves listing all labeled patterns with  $k$  edges that are frequent in a data graph. Typically, the exploration is performed in edge-induced manner, which gets expressed using edge-induced patterns. The frequency of a pattern (also called support) is often measured using anti-monotonic approaches like the *minimum node image (MNI)* [6] support measure, which ensure that if the support of a pattern is  $s$ , then the support of its subpatterns is at least  $s$ . This property allows skipping exploration of patterns if their subpatterns are not frequent.

The MNI support measure consists of a table with a column for each group of symmetric vertices in  $p$ . For each  $v \in V(p)$ , the column corresponding to  $v$  contains  $m(v)$  from every match  $m$ . The support is defined to be the size of the smallest column in this table. Hence, to compute this support, the aggregation operation is to join tables by concatenating their respective columns. Each column contains  $O(|V|)$  vertices which are merged during aggregation, so while each match incurs a constant amount of work to append to the table, combining the tables can take  $O(|V|)$  work.

### 3 Pattern Morphing

Pattern morphing is a novel technique that enables structure-aware algebra over patterns to *morph* (or convert) them into alternative patterns so that matches for the original patterns get expressed in terms of matches for other patterns.

We first describe the main idea behind pattern morphing using illustrative examples, and then formalize the semantics of pattern morphing.



**Figure 3.** Identifying matches for different patterns.

#### 3.1 Intuition & Main Idea

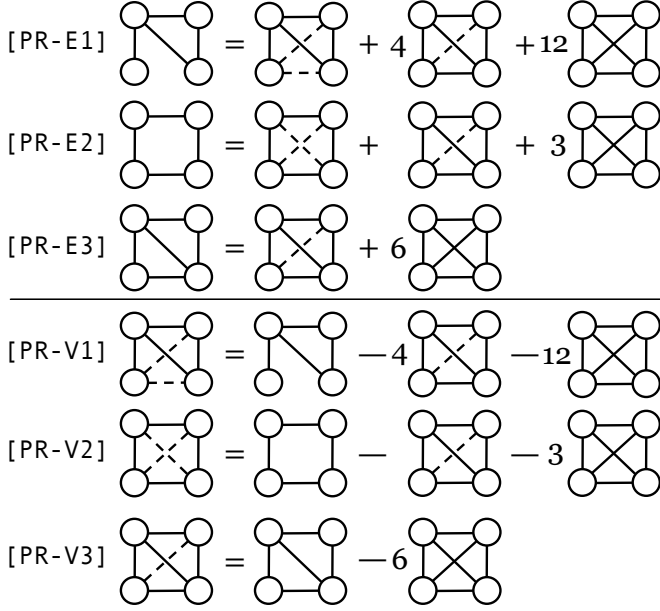
Pattern morphing primarily exploits the structural similarities across different patterns. The intuition behind pattern morphing can be summarized using two key observations.

- A match for an edge-induced pattern  $p^E$  on  $n$  vertices is also a match for all of its subpatterns on  $n$  vertices. For example, a match for a 4-clique is also a match for an edge-induced 4-cycle, since the 4-clique contains all the edges of the 4-cycle. Note that this does not apply to vertex-induced subpatterns: even though the vertex-induced 4-cycle contains the same 4 edges, the additional two anti-edges exclude matches for 4-cliques.
- A match for a vertex-induced pattern  $p^V$  is always a match for the corresponding edge-induced pattern  $p^E$ , since  $p^V$  matches exactly the edges in  $p^E$ .

These observations mean that we can logically partition the matches for an edge-induced pattern into disjoint sets of matches for vertex-induced patterns. For example, consider a match for an edge-induced 4-cycle. The vertices in this match may have edges between them in the data graph which are not present in the pattern. If they do not, they form a match for the vertex-induced 4-cycle (match a-b-c-d in Figure 3a). If they have exactly one extra edge, they form a match for the vertex-induced chordal 4-cycle (match d-c-g-f in Figure 3a). And finally, if they have two extra edges, they form a match for the 4-clique (match a-d-f-e in Figure 3a). Note that these situations are mutually exclusive since they depend on specific edges being present or absent.

While the above partitioning essentially allows converting the matches, a match for a given pattern can potentially contain multiple matches for another pattern. For example, a match for 4-clique contains 3 unique matches for edge-induced 4-cycle, as shown in Figure 3b. Hence, in order to convert a match for a 4-clique into a match for a 4-cycle, we must correctly map its vertices, using the subgraph isomorphisms of the 4-cycle into the 4-clique. It is crucial to note that these subgraph isomorphisms are needed to be



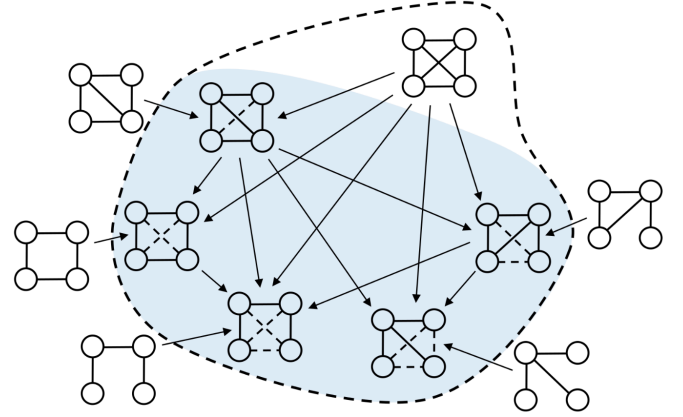


**Figure 4.** Sample equations resulting from pattern morphing. [PR-E1] - [PR-E3] morph edge-induced patterns (on left-hand side) to vertex-induced patterns, whereas [PR-V1] - [PR-V3] morph vertex-induced patterns to a mix of vertex-induced and edge-induced patterns. The coefficients besides the patterns indicate the number of unique matches resulting from subgraph isomorphisms. Note that patterns in the right-hand side of these equations can be substituted to generate different equations for any given pattern. The optimal equation (i.e., one that delivers best performance) for any given pattern depends on various factors including: details of data graph and pattern structures, application-specific operations, and system-specific nuances.

performed only across patterns before matching starts (i.e., once only), and not across the individual matches.

It is such kind of structure-based match conversions that pattern morphing enables. Pattern morphing performs structure-aware algebra over patterns (and hence, their matches) to capture all the matches for a given pattern using matches of different patterns in any data graph. Figure 4 shows a few examples of how patterns can be morphed to any given pattern. The coefficients besides the patterns indicate the number of unique matches resulting from subgraph isomorphisms. Hence, for our example of morphing 4-clique to 4-cycle (discussed above), the 4-clique in Equation [PR-E2] has a coefficient 3. On the other hand, there is only a single unique subgraph isomorphism from edge-induced 4-cycle to vertex-induced 4-cycle and chordal 4-cycle, which is indicated in the same equation by no coefficients for those patterns.

Since pattern morphing mainly relies on structural similarities, the injective nature of subgraph isomorphism allows



**Figure 5.** Motif counting example with pattern morphing. The patterns inside the dashed boundary are query patterns for motif counting. With pattern morphing, the patterns outside the shaded regions are matched, and their results are transformed to directly generate results for patterns inside the shaded region.

us to go the other direction as well (i.e., from edge-induced match to vertex-induced match) by simply manipulating the equations to put the desired pattern on the left-hand side. This can be seen in Equations [PR-V1] to [PR-V3] in Figure 4.

**Motif Counting Example.** The equations in Figure 4 can be directly used for 4-motif counting: the drawing of each pattern stands for the number of matches for it, and they get multiplied by the coefficients to capture the number of subgraph isomorphisms across patterns. Hence, the equations can be algebraically manipulated in order to express any set of patterns in terms of another. The resulting outcome of pattern morphing for 4-motif counting is shown in Figure 5: the final counts for all the 4-sized motifs can be directly computed using the counts for the 4-clique as well as the edge-induced variants of the other patterns.

**Discussion.** While pattern morphing can theoretically be applied across patterns of different sizes, it is often unnecessary to convert results of larger patterns to smaller patterns since it is usually expensive to match larger sized patterns instead of smaller sized patterns. Hence, we focus on morphing patterns to alternative patterns that have the same number of vertices as the patterns being morphed.

The pattern morphing strategy discussed so far is purely based on the structural similarities across patterns. Hence, the details regarding morphing across pattern pairs can be generated once only, and reused across different data graphs and applications. In Section 4.1, we will discuss how the structure of the data graph and the complexity of the aggregation type (for different applications) can be used to select the specific morphings for a given execution.

Finally, pattern morphing can be applied to any pattern and any aggregation type; while we only showed the motif

counting use case above, complex aggregation types like MNI tables [6] (required in Frequent Subgraph Mining), or simple enumeration can also be performed with pattern morphing. As we will see next, the semantics of pattern morphing will be formalized by abstracting out details like aggregation types.

### 3.2 Semantics of Pattern Morphing

We now formalize the semantics of constructing the correct set of alternative patterns for any given (edge-induced or vertex-induced) pattern.

**3.2.1 Definitions & Notations.** We define the key terms using which we build our analysis.

*Subgraph Isomorphisms:* Throughout this section, we will treat subgraph isomorphisms as functions between sequences of numbers, each representing a vertex. This means, for graphs  $p$  and  $q$  we view a subgraph isomorphism from  $p$  into  $q$  as a function  $f : \{1, \dots, |p|\} \rightarrow \{1, \dots, |q|\}$ . Hence, when  $|p| = |q|$ , a subgraph isomorphism is simply a permutation. For example, there are three subgraph isomorphisms from an edge-induced 4-cycle to a 4-clique, as shown in Figure 6.

Given two patterns  $p$  and  $q$ , we define  $\phi(p, q)$  to be the set of all subgraph isomorphisms from  $p$  into  $q$ . Figure 6 shows two examples:  $\phi(p_1^E, p_3^V)$  which contains four subgraph isomorphisms from edge-induced tailed triangle ( $p_1^E$ ) to vertex-induced chordal 4-cycle ( $p_3^V$ ); and,  $\phi(p_2^E, p_4)$  containing the three subgraph isomorphisms from edge-induced 4-cycle ( $p_2^E$ ) to 4-clique ( $p_4$ ).

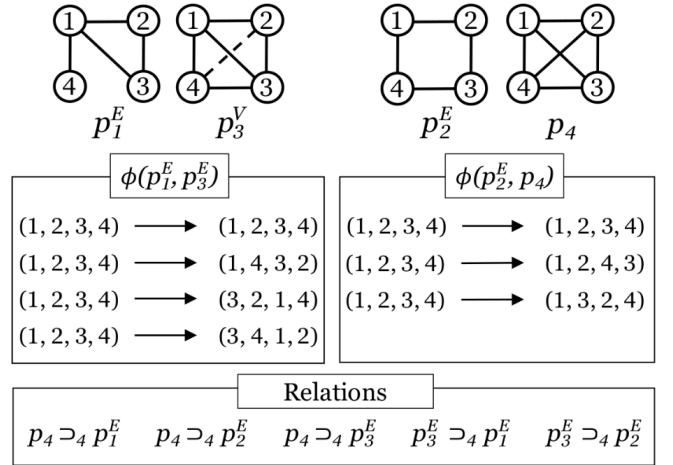
*Match Sets:* Given a pattern  $p$  (either edge-induced or vertex-induced) and a data graph  $G$ ,  $M(p, G)$  denotes the set of all matches for  $p$  in data graph  $G$ . Each individual match in  $M(p, G)$  is simply a function  $m : \{1, \dots, |p|\} \rightarrow \{1, \dots, |G|\}$ . Since our analysis will mainly focus on morphing patterns, which is inherently independent of any data graph, we simply use  $M(p)$  as a shorthand notation (i.e., we drop  $G$ ).

*Permuting Matches based on Subgraph Isomorphisms:* Given a data graph  $G$  and two patterns  $p$  and  $q$  with  $n$  vertices, we write  $M(q) \circ \phi(p, q)$  to mean the permutations of the vertices in each match  $m \in M(q)$  according to the subgraph isomorphisms from  $p$  into  $q$ . Note that the permuted matches are nothing but matches of  $p$  in  $G$ . Specifically, since a match for  $q$  in  $G$  is an injective function  $m : \{1, \dots, n\} \rightarrow \{1, \dots, |G|\}$ , and  $\phi(p, q)$  consists of bijective functions  $f : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ :

$$M(q) \circ \phi(p, q) = \{m \circ f : m \in M(q) \wedge f \in \phi(p, q)\}$$

where  $m \circ f$  means permuting the domain of  $m$  according to  $f$ .

For example, consider the matches for 4-clique (pattern  $p_4$  in Figure 6) in the data graph from Figure 3a. Here,  $M(p_4) =$



**Figure 6.** Examples of subgraph isomorphisms across patterns, and non-isomorphic superpatterns.

$\{\{1 \rightarrow a, 2 \rightarrow d, 3 \rightarrow f, 4 \rightarrow e\}\}$  and hence:

$$M(p_4) \circ \phi(p_2^E, p_4) = \left\{ \begin{array}{l} \{1 \rightarrow a, 2 \rightarrow d, 3 \rightarrow f, 4 \rightarrow e\}, \\ \{1 \rightarrow a, 2 \rightarrow d, 4 \rightarrow f, 3 \rightarrow e\}, \\ \{1 \rightarrow a, 3 \rightarrow d, 2 \rightarrow f, 4 \rightarrow e\} \end{array} \right\}$$

Each of  $m \circ f \in M(p_4) \circ \phi(p_2^E, p_4)$  is a match for  $p_2^E$  (i.e., edge-induced 4-cycle) in subgraph of  $G$  containing the four vertices  $a, d, f$  and  $e$ .

*Non-Isomorphic Superpatterns:* Finally, given patterns  $p$  and  $q$ , we write  $q \supset_n p$  if  $q$  is a non-isomorphic superpattern of  $p$  containing at most  $n$  vertices. In Figure 6, 4-clique is a non-isomorphic superpattern of the remaining three patterns, while vertex-induced chordal 4-cycle is a non-isomorphic superpattern of edge-induced 4-cycle and edge-induced tailed triangle.

### 3.2.2 Transforming Matches for Pattern Morphing.

The key idea behind pattern morphing is to transform the matches of a given pattern into matches for other patterns. This is captured by the *Match Conversion Theorem*.

**Theorem 3.1 (Match Conversion Theorem).** Let  $p^E$  be an edge-induced pattern with  $n$  vertices, and  $p^V$  be its vertex-induced variant. Then,

$$M(p^E) = M(p^V) \cup \bigcup_{q^E \supset_n p^E} M(q^V) \circ \phi(p^E, q^E)$$

*Proof.* Since we are proving set equality, we will first show how every match in  $M(p^E)$  is contained in the set on the right-hand side of the equation, and then show that  $M(p^E)$  contains every match from the right-hand side.

Let  $m$  be a match in  $M(p^E)$ , and  $q^V$  be the pattern of the subgraph induced by the image of  $m$ .  $q^V$  must have at least as many edges as  $p^E$ , since it was induced by a match for  $p^E$ . – *Case 1:* If  $q^V$  has the same number of edges as  $p^E$ ,  $q^V$  is isomorphic to  $p^V$  and hence  $m \in M(p^V)$ .

– *Case 2:* Otherwise,  $q^V$  has more edges than  $p^E$ . Consider the edge-induced pattern  $q^E$  corresponding to  $q^V$ .  $q^E$  must be a superpattern of  $p^E$ , since  $q^V$  has more edges than  $p^E$  but contains all the edges of  $p^E$ . This means  $\phi(p^E, q^E)$  is non-empty. For any  $f \in \phi(p^E, q^E)$ , observe that  $m \circ f^{-1} : V(q^E) \rightarrow V(p^E) \rightarrow G$  is a match for  $q^V$ , since  $V(q^E) = V(q^V)$  and  $q^E$  was obtained from the induced pattern  $q^V$ . This means  $m \circ f^{-1} \in M(q^V)$ , and hence  $m \circ f^{-1} \circ f = m \in M(q^V) \circ \phi(p^E, q^E)$ .

Therefore, we showed  $M(p^E)$  is a subset of the right-hand side of the equation. Next, we will prove that the right-hand side is contained within  $M(p^E)$ .

First, note that a match for  $p^V$  is trivially a match for  $p^E$ , since  $p^E$  and  $p^V$  differ only in anti-edges, so  $M(p^V) \subseteq M(p^E)$ .

Next, take any match  $m \in M(q^V)$  where  $q^E \supset_n p^E$ .  $m$  is also in  $M(q^E)$  by the same reasoning as above. But then for any  $f \in \phi(p^E, q^E)$ ,  $m \circ f : V(p^E) \rightarrow V(q^E) \rightarrow G$  is a match for  $p^E$  since any match for  $q^E$  contains all edges required for matching  $p^E$ . Hence,  $M(q^V) \circ \phi(p^E, q^E) \subseteq M(p^E)$ .

Taking the union of  $M(p^V)$  and  $M(q^V) \circ \phi(p^E, q^E)$  for each  $q^E \supset_n p^E$  gives the set of matches that are contained in  $M(p^E)$ .  $\square$

The Match Conversion Theorem reflects a useful relationship between edge-induced and vertex-induced patterns. In fact, although the theorem is stated in terms of morphing from vertex-induced to edge-induced, we can move in the other direction as well:

**Corollary 3.1.** *Let  $p^E$  be an edge-induced pattern with  $n$  vertices, and  $p^V$  be its vertex-induced variant. Then,*

$$M(p^V) = M(p^E) \setminus \bigcup_{q^E \supset_n p^E} M(q^V) \circ \phi(p^E, q^E)$$

*Proof.* Let the set  $\bigcup_{q^E \supset_n p^E} M(q^V) \circ \phi(p^E, q^E)$  be denoted as  $B$ . From Theorem 3.1, we have  $M(p^E) = M(p^V) \cup B$ . Notice that if  $M(p^V)$  is disjoint from all  $M(q^V) \circ \phi(p^E, q^E)$  where  $q^E \supset_n p^E$ , then we could take the set difference on both sides of the equation from Theorem 3.1 with  $B$  to prove the corollary, simply because no element of  $M(p^V)$  would be removed by the set difference operation.

It remains to prove that  $M(p^V)$  is disjoint from the various  $M(q^V) \circ \phi(p^E, q^E)$  where  $q^E \supset_n p^E$ . We prove this by contradiction.

Let  $m$  be a match in  $M(p^V) \cap M(q^V) \circ \phi(p^E, q^E)$  for any  $p^V$  and some  $q^V$  where  $q^E \supset_n p^E$ . First note that if  $p^V$  is a clique, there is no  $q^E \supset_n p^E$ , and  $m$  cannot exist, so we are done. Instead, suppose  $p^V$  is not a clique, and thus has at least one anti-edge. Since  $q^E \supset_n p^E$ , for each  $f \in \phi(p^E, q^E)$  there is an edge  $(f(u), f(v)) \in E(q^E)$  such that  $(u, v) \notin E(p^E)$ . This means  $(u, v)$  forms an anti-edge constraint in  $p^V$ , and  $(f(u), f(v))$  forms an edge constraint in  $q^V$ . As  $m \in M(q^V) \circ \phi(p^E, q^E)$ , it can be written in the form  $m = m' \circ f$  where  $m' \in M(q^V)$  and  $f \in \phi(p^E, q^E)$ . But then  $((m' \circ f)(u), (m' \circ f)(v))$

must be an edge in  $G$ , since  $(f(u), f(v)) \in E(q^V)$  and  $m'$  is a match for  $q^V$ . This directly contradicts the anti-edge constraint in  $p^V$  that we established above.

Hence  $M(p^V) \cap M(q^V) \circ \phi(p^E, q^E) = \emptyset$  for all  $q^E \supset_n p^E$ , as desired.  $\square$

By recursively substituting in the equation in Corollary 3.1, we can express the matches of a vertex-induced pattern in terms of the matches for edge-induced patterns. Since the final superpattern of any non-clique pattern is a clique, this recursive substitution terminates.

**Discussion.** Using Theorem 3.1, we can easily materialize edge-induced matches from vertex-induced matches. Similarly, we can convert edge-induced matches to vertex-induced ones using Corollary 3.1; however, materializing these vertex-induced matches would require first materializing  $M(p^E)$  for computing the set difference, which may not always be feasible. Nevertheless, the ability to accurately convert edge-induced matches to vertex-induced ones (and vice-versa) enables directly computing results for any given pattern from its alternative pattern sets, as shown next.

**3.2.3 Aggregation with Pattern Morphing.** Graph mining applications often compute aggregated statistics (e.g., counts, support, etc.) based on the matches that get explored from the data graph. We capture how Pattern Morphing naturally enables efficient computation of such aggregated statistics.

Let  $a = (\lambda, \oplus)$  be the aggregation in graph mining applications where  $\lambda$  is a map from matches to an aggregation value and  $\oplus$  is a commutative operator for combining aggregation values. For a set of matches  $M(p)$ , we write  $a(M(p))$  as a shorthand for  $\bigoplus_{m \in M(p)} \lambda(m)$ .

Note that our definition of aggregation captures functions whose images can be summed. The simplest example is counting, where  $\lambda(M) = |M|$  for a set of matches  $M$  and the  $\oplus$  operator is the traditional integer sum. For more complicated applications like FSM,  $\lambda$  computes the MNI table of a set of matches and the  $\oplus$  operator joins tables on columns. In addition, we define a *permute* operator  $\circ_*$  for aggregation values to account for the permutations according to  $\phi(q, p)$  (similar to  $\circ$  defined on matches in Section 3.2.1). In the counting example, for a match  $m$  and a permutation  $f$ ,  $\circ_*$  can be defined as  $a(m) \circ_* f = a(m)$ . In FSM, on the other hand,  $a(m) \circ_* f$  permutes the columns of the MNI table of a match  $m$  according to permutation  $f$ .

Using this abstraction, we prove the following critical theorem that enables directly computing aggregation values using matches for morphed patterns.

**Theorem 3.2 (Aggregation Conversion Theorem).** *Let  $a = (\lambda, \oplus)$  be the aggregation in graph mining applications. For any match  $m$ , permutation  $f$  on  $n$  vertices, and permute*

operator  $\circ_*$ , if  $a(m \circ f) = a(m) \circ_* f$ , then:

$$a(M(p^E)) = a(M(p^V)) \oplus \left( \bigoplus_{q^E \supset_n p^E} \bigoplus_{f \in \phi(p^E, q^E)} a(M(q^V)) \circ_* f \right)$$

*Proof.* The equation follows immediately from Theorem 3.1.

$$\begin{aligned} a(M(p^E)) &= a \left( M(p^V) \cup \bigcup_{q^E \supset_n p^E} M(q^V) \circ \phi(p^E, q^E) \right) \\ &= a(M(p^V)) \oplus \left( \bigoplus_{q^E \supset_n p^E} a(M(q^V) \circ \phi(p^E, q^E)) \right) \\ &= a(M(p^V)) \oplus \left( \bigoplus_{q^E \supset_n p^E} \bigoplus_{f \in \phi(p^E, q^E)} a(M(q^V)) \circ_* f \right) \end{aligned}$$

□

Theorem 3.2 demonstrates the benefit of pattern morphing as a significant amount of matches no longer need to be materialized to compute the aggregation values. In other words, we can obtain the results of an aggregation over a pattern by using the aggregations over a completely different set of patterns. The performance gains achieved due to this can be captured as follows.

**Corollary 3.2.** *Let  $T(p)$  be the time it takes to find all matches of a pattern  $p$  with  $n$  vertices and apply aggregation  $a = (\lambda, \oplus)$  that satisfies Theorem 3.2. Then for patterns  $p^E$  and  $q^E \supset_n p^E$ , the aggregation  $a(M(q^V) \circ \phi(p^E, q^E))$  can be computed in time  $O(T(q^V)) + O(|\phi(p^E, q^E)|)$ .*

*Proof.* By definition, we already know:

$$a(M(q^V) \circ \phi(p^E, q^E)) = \bigoplus_{f \in \phi(p^E, q^E)} a(M(q^V)) \circ_* f$$

This means that we can compute  $a(M(q^V) \circ \phi(p^E, q^E))$  by first computing  $a(M(q^V))$  and then applying the permutations  $f \in \phi(p^E, q^E)$ . Hence,  $O(T(q^V))$  time spent computing  $a(M(q^V))$  and  $O(|\phi(p^E, q^E)|)$  time spent adjusting the results. □

Corollary. 3.2 shows how pattern morphing boosts performance by reducing the number of matches we need to produce for each  $q^E \supset_n p^E$  by a factor of  $|\phi(p^E, q^E)|$ .

Composing Theorem 3.2 and Corollary. 3.1 leads to the expected result that aggregations of a vertex-induced pattern can be computed using the aggregations of edge-induced patterns, with the added restriction that the aggregation function's image must also be additive. We skip the proof since it is nearly identical to that of Corollary. 3.2. Additionally, the alternative pattern set can contain a combination of vertex-induced and edge-induced patterns by converting the intermediate aggregations through recursive applications of the theorem.

## 4 Implementation & Evaluation

### 4.1 Automatic Pattern Morphing Implementation

The pattern morphing theory developed in Section 3 can be applied in practice to accelerate graph mining tasks in any general-purpose graph mining system. Since pattern morphing guides conversion of results across different patterns, it can be added as an 'external module' to any graph mining system (i.e., without changing the implementation of the system's subgraph exploration strategies). Such a module would: (a) generate alternative pattern sets for the input pattern sets; and, (b) transform the aggregation/operation results after matches get explored.

For this evaluation, we developed an automatic pattern morphing engine for Peregrine, which is the state-of-the-art pattern-aware graph mining system. Peregrine, being pattern-aware, already exploits efficient matching algorithms to generate optimized matching plans for different patterns. Adding pattern morphing as an external module enables leveraging the optimized plans across different patterns.

There can be multiple alternative pattern sets for any given set of patterns, and those different alternative pattern sets can deliver different performance depending on the specific patterns they contain. Hence, selecting the right alternative pattern sets for a given set of patterns is crucial to deliver high performance. This can be addressed by developing a 'pattern morphing optimizer' that minimizes the cost of pattern sets to construct the best alternative pattern sets. Here, the cost of a given pattern set must capture three main factors:

1. The nuances of exploration strategies used by the underlying system. For instance, on Peregrine this would include the work done in adjacency list intersection and difference operations, positive effects of symmetry breaking, and the custom pattern-specific optimizations it employs. On other systems that may not be fully pattern-aware (e.g., due to lack of symmetry breaking), the cost of exploring patterns would be different.
2. The details of performing application-specific operations on matches. For instance, counting patterns takes  $O(1)$  for groups of matches, whereas computing MNI support for FSM takes  $O(|V(G)|)$  time. With pattern morphing, the conversion costs for these operations must also be considered.
3. The details of the data graph. This would not only include the structural details of the data graph like degree distribution and connectivity information, but also include application-specific properties in the data graph like label distributions (required for FSM).

We evaluate the effectiveness of such a cost-model based pattern morphing optimizer by incorporating it with our pattern morphing engine for Peregrine.



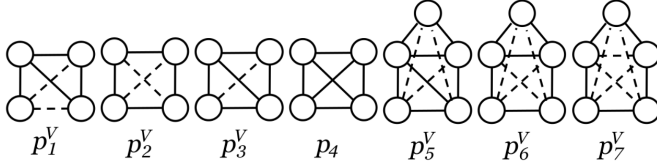


Figure 7. Patterns used in evaluation.

$G$	$ V(G) $	$ E(G) $	$ L(G) $	Max. Deg.	Avg. Deg.
(MI) Mico [13]	100K	1M	29	1359	22
(PA) Patents [18]	3.7M	16M	37	789	10
(YT) YouTube [9]	6.9M	44M	38	4039	12
(OK) Orkut [51]	3M	117M	—	33133	76

Table 2. Real-world graphs used in evaluation.  
'—' indicates unlabeled graph.

## 4.2 Experimental Setup

**System.** We evaluate the pattern morphing engine in Peregrine on a Google Cloud n2-highcpu-32 instance which consists of a 2.8GHz Intel Cascade Lake processor with 32 logical cores and 32GB of RAM.

**Datasets.** Table 2 lists the data graphs used in our evaluation. Mico (MI) is a co-authorship graph labeled with each author’s research field. Patents (PA) is a patent citation graph, where each patent is labeled with the year it was granted. Youtube (YT) consists of videos crawled by [9] from 2007–2008, with edges between related videos. Videos are labeled according to their ratings, like in [12]. Orkut (OK) is an unlabeled social network graph where edges represent friendships between users. All these datasets have been used to evaluate previous systems [12, 24, 31, 44].

**Applications.** We run experiments on a wide array of applications: counting motifs with 3 and 4 vertices, size 3 FSM, and matching vertex-induced versions of the patterns in Figure 7. Throughout the evaluation, we use the following notations for different variants:

- **No PMR:** this is Peregrine without pattern morphing, where the query patterns for each of the applications are directly used to explore the data graph.
- **Naïve PMR:** this is Peregrine with pattern morphing, where the query patterns are morphed so that edge-induced input patterns are morphed into vertex-induced patterns, and vertex-induced input patterns are morphed into edge-induced patterns.
- **Cost-Based PMR:** this is Peregrine with the pattern-morphing optimizer that constructs the best alternative pattern sets by minimizing their costs.

## 4.3 Overview of Performance Results

Table 3 shows the performance of pattern morphing for all applications across different data graphs. Overall, naively

App	$G$	No PMR	Naïve PMR	Cost-Based PMR
3-MC	MI	0.14	<b>0.07</b>	<b>0.07</b> (2×)
	PA	0.97	<b>0.52</b>	<b>0.52</b> (1.87×)
	YT	6.83	<b>2.06</b>	<b>2.06</b> (3.32×)
	OK	30.15	<b>10.57</b>	<b>10.57</b> (2.85×)
4-MC	MI	16.53	<b>3.30</b>	<b>3.30</b> (5.01×)
	PA	21.27	<b>6.10</b>	<b>6.10</b> (3.49×)
	YT	178.04	<b>41.78</b>	<b>41.78</b> (4.26×)
	OK	37017.93	<b>3138.76</b>	<b>3138.76</b> (11.79×)
$p_1^V$	MI	4.44	<b>1.16</b>	<b>1.16</b> (3.82×)
	PA	<b>1.11</b>	1.17	<b>1.11</b>
	YT	10.70	<b>9.27</b>	<b>9.27</b> (1.15×)
	OK	913.70	<b>177.79</b>	<b>177.79</b> (5.14×)
$p_2^V$	MI	<b>1.89</b>	3.15	<b>1.89</b>
	PA	<b>3.97</b>	4.35	<b>3.97</b>
	YT	<b>31.91</b>	34.18	<b>31.91</b>
	OK	<b>1730.37</b>	3010.42	<b>1730.37</b>
$p_3^V$	MI	3.04	<b>1.08</b>	<b>1.08</b> (2.81×)
	PA	<b>0.82</b>	0.95	<b>0.82</b>
	YT	8.07	<b>6.79</b>	<b>6.79</b> (1.16×)
	OK	403.41	<b>156.99</b>	<b>156.99</b> (2.57×)
$p_5^V$	MI	84.08	<b>65.62</b>	<b>65.62</b> (1.28×)
	PA	<b>8.32</b>	10.77	<b>8.32</b>
	YT	<b>42.50</b>	83.03	<b>42.50</b>
	OK	41130.03	<b>9489.79</b>	<b>9489.79</b> (4.33×)
$p_6^V$	MI	<b>31.98</b>	88.08	<b>31.98</b>
	PA	35.23	<b>25.34</b>	<b>25.34</b> (1.39×)
	YT	<b>109.48</b>	182.31	<b>109.48</b>
	OK	—	<b>27014.10</b>	<b>27014.10</b>
$p_7^V$	MI	<b>23.56</b>	423.87	<b>23.56</b>
	PA	<b>14.95</b>	58.37	<b>14.95</b>
	YT	<b>67.03</b>	429.97	<b>67.03</b>
	OK	—	—	—
$p_2^E$	MI	2.09	4.75	<b>1.93</b> (1.08×)
	PA	<b>3.29</b>	4.16	<b>3.29</b>
	YT	27.84	21.22	<b>16.69</b> (1.67×)
	OK	2896.95	2331.59	<b>1841.42</b> (1.57×)
$\{p_2^E, p_3^E\}$	MI	3.92	2.70	<b>1.93</b> (2.03×)
	PA	<b>3.29</b>	3.62	<b>3.29</b>
	YT	18.82	17.95	<b>16.69</b> (1.28×)
	OK	2908.70	1997.15	<b>1845.09</b> (1.58×)
$\{p_5^V, p_6^V\}$	MI	116.06	<b>88.08</b>	<b>88.08</b> (1.32×)
	PA	43.55	<b>25.34</b>	<b>25.34</b> (1.56×)
	YT	<b>151.98</b>	182.31	<b>151.98</b>
	OK	—	<b>27014.10</b>	<b>27014.10</b>
3-FSM	MI	127.16	101.60	<b>71.31</b> (1.78×)
	PA	644.74	3772.41	<b>639.75</b> (1.01×)
	YT	<b>491.07</b>	550.29	<b>491.07</b>

Table 3. Execution times in seconds (including time spent morphing patterns) with and without pattern morphing. '—' represents executions that did not finish within 24 hours.

using pattern morphing shows a benefit of 1.16-11.79 $\times$ , with potential slowdowns in only few cases. The cost-based pattern morphing correctly captures those cases and retains high performance benefits across all cases. Below we analyze how pattern morphing improves each of the applications.

#### 4.4 Pattern Morphing for Motif Counting

The greatest benefits for pattern morphing occur in the motif counting application, since all superpatterns are already included in the input pattern set. As a result, the morphing engine always generates an optimal morphing set. Furthermore, counting is a very cheap aggregation, which makes morphing vertex-induced patterns very profitable in terms of the set operation versus aggregation tradeoff.

Pattern morphing yielded a better execution time in all the motif counting experiments. For 4-motif counting on the large Orkut graph, pattern morphing yields 11.79 $\times$  speedup over the baseline that doesn't use pattern morphing.

#### 4.5 Pattern Morphing for Pattern Matching

Matching individual patterns is a test of the worst case for pattern morphing, since several extra superpatterns need to be matched in order to morph a given single pattern. Despite this, our cost-based pattern morphing optimizer finds opportunities for a 5 $\times$  speedup when matching  $p_1^V$  on Orkut, and 4.33 $\times$  when matching  $p_5^V$  on Orkut. Additionally, with pattern morphing, Peregrine was able to match  $p_6^V$  on Orkut in 7.5 hours, whereas without morphing the execution did not finish within 24 hours.

We also match groups of patterns to observe whether it amortizes the cost of the extra superpatterns. Interestingly, when matching only  $p_6^V$ , morphing it would lead to a slowdown on the MiCo graph, but when the input pattern set consists of both  $p_5^V$  and  $p_6^V$ , pattern morphing yields a 1.32 $\times$  speedup.

Table 4 lists the alternate pattern sets selected by Cost-Based PMR for some of the input patterns across different data graphs. As we can see, the cost-based pattern morphing optimizer selects different outcomes that minimize the pattern set costs. For instance, it correctly detects that morphing  $p_3^V$  is beneficial for all graphs except the Patents graph. Similarly, while Naïve PMR ends up morphing both patterns in  $\{p_2^E, p_3^E\}$ , Cost-Based PMR correctly morphs only  $p_2^E$  for all graphs except the Patents graph.

#### 4.6 Pattern Morphing for Frequent Subgraph Mining

We ran 3-FSM on the labeled datasets MiCo, Patents, and YouTube with support measures of 4000, 23000, and 300000, respectively (similar to [12, 24, 44]). On the MiCo graph, pattern morphing reduces the execution time by 43%, but the benefit is marginal on Patents, and none on YouTube since the cost-based pattern morphing optimizer ends up choosing not to morph the input pattern set. These performance

App	G	Alt. Set
$p_1^V$	MI	$\{p_1^E, p_3^E, p_4\}$
	PA	$\{p_1^E, p_3^E, p_4\}$
	YT	$\{p_1^E, p_3^E, p_4\}$
	OK	$\{p_1^E, p_3^E, p_4\}$
$p_2^V$	MI	$\{p_2^V\}$
	PA	$\{p_2^V\}$
	YT	$\{p_2^V\}$
	OK	$\{p_2^V\}$
$p_2^E$	MI	$\{p_2^V, p_3^E, p_4\}$
	PA	$\{p_2^E\}$
	YT	$\{p_2^V, p_3^E, p_4\}$
	OK	$\{p_2^V, p_3^E, p_4\}$
$p_3^V$	MI	$\{p_3^E, p_4\}$
	PA	$\{p_3^V\}$
	YT	$\{p_3^E, p_4\}$
	OK	$\{p_3^E, p_4\}$
$\{p_2^E, p_3^E\}$	MI	$\{p_2^V, p_3^E, p_4\}$
	PA	$\{p_2^E, p_3^E\}$
	YT	$\{p_2^V, p_3^E, p_4\}$
	OK	$\{p_2^V, p_3^E, p_4\}$

**Table 4.** Alternative pattern sets used in Cost-Based PMR.

differences are inherent to the data graph, and depend on the distribution of labels among vertices.

## 5 Related Work

There has been a variety of research to develop efficient graph mining solutions. To the best of our knowledge, this is the first formal treatment of generic structure-based transformations of input patterns in graph mining systems.

**General-Purpose Graph Mining.** Several general-purpose graph mining systems have recently emerged in the literature [8, 12, 24, 31, 44, 48]. Arabesque [44] and Fractal [12] are exploration-based general-purpose graph mining systems which mine subgraphs through iterative extensions by edges or vertices. RStream [48] is an out-of-core graph mining system which structures computations as relational operations on tables of subgraphs and stores intermediate state on disk. Pangolin [8] is a recent exploration-based graph mining system which supports offloading computation to GPU. ASAP [23] is a general-purpose approximate graph mining system allowing users to navigate the tradeoff between error and performance by analyzing graph mining computations. [31] compiles input patterns into exploration programs for graph mining by applying tournament algorithm to select efficient set operation schedules. Peregrine [24] is a pattern-aware general-purpose graph mining system which

exploits structural and label properties of input patterns to efficiently mine their matches.

All these works focus on efficiently processing graph mining applications as expressed by the user, by optimizing the exploration plans for the subgraph structures of interest. In contrast, we focus on how graph mining applications can be transformed by morphing the subgraph structures of interest to enable more efficient execution. Pattern morphing can be integrated with these systems as an add-on module to improve their performance.

**Motif Counting.** A myriad of research has been conducted on efficient algorithms for mining motifs. [17] developed a technique to break symmetries of patterns by enforcing a partial-ordering on pattern vertices. [3] is a fast, parallel algorithm for counting size 3 and 4 motifs using combinatorial identities. RAGE [29] provides a method for efficiently computing edge-induced size-4 motifs, and equations for converting the results to those for vertex-induced motifs. [21] efficiently lists the automorphism groups of pattern vertices and uses them to compute exact counts for motifs with 2-5 vertices. There is also research on techniques for computing approximate counts of motifs [32, 36].

All of these works focus on counting patterns of fixed size using explicit formulae for those specific patterns. Our work differs by developing a general algebra for morphing arbitrary patterns and transforming arbitrary graph mining applications. Pattern morphing is not restricted by pattern types, pattern size, or application semantics.

**Pattern Matching.** These works aim to devise more efficient subgraph isomorphism solutions using sophisticated analysis of data and query graphs [4, 5, 19, 20, 25, 26, 37, 42]. DualSIM [25] is an out-of-core pattern matching system that uses pattern-core decomposition to efficiently map pattern vertices onto data vertices. CECI [4], a distributed pattern matching system, builds clustered indexes in the data graph based on the query pattern topology and uses these indexes to quickly materialize matches. [5, 19, 20] decompose the query graph into tree and non-tree edges and match them efficiently guided by sophisticated heuristics. These works are focused on algorithms for pattern matching, which is orthogonal to the issue of what patterns to match.

**Frequent Subgraph Mining.** GraMi [13] performs efficient FSM by enumerating a minimal set of matches to determine frequency of patterns. ScaleMine [2] samples matches of a pattern in the data graph to build a statistical profile of pattern frequency, which guides the computation of exact results. IncGM+[1] is a system for continuous frequent subgraph mining that prunes the search space using a set of infrequent subgraphs which are adjacent to frequent subgraphs. None of these works are pattern-based, and instead view the FSM computation in terms of subgraphs of the data graph.

**Graph Processing.** Many works address computations on static and dynamic graphs [11, 15, 16, 22, 27, 28, 30, 35, 39, 40, 43, 45–47, 52]. Research in this space is typically applicable to computations on vertex and edge values, as opposed to graph mining problems which are concerned with substructures of the data graph. Techniques like [27] develop custom transformations for specific substructures to shrink the data graph and speed up value propagation.

## 6 Conclusion

We presented PATTERN MORPHING to enable structure-aware algebra over patterns. Pattern Morphing *morphs* a given set of patterns into alternative pattern sets that can be used to compute accurate results for the original set of patterns. We developed the theoretical foundations for Pattern Morphing that guarantee accurate conversion of matches across different patterns, and further enable direct conversion of results from application-specific operations over explored matches. To evaluate the effectiveness of pattern morphing, we implemented an automatic pattern morphing engine and incorporated it in the Peregrine system. Our results showed that pattern morphing significantly improved the performance of various graph mining applications. Being a generic technique at pattern-level, pattern morphing can be incorporated in existing graph mining systems, and it can be applied to various problems beyond graph mining.

## References

- [1] E. Abdelhamid, M. Caim, M. Sadoghi, B. Bhattacharjee, Y. Chang, and P. Kalnis. Incremental frequent subgraph mining on large evolving graphs. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2710–2723, 2017.
- [2] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. Scalemine: Scalable parallel frequent subgraph mining in a single large graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, pages 61:1–61:12, 2016.
- [3] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient graphlet counting for large networks. In *IEEE International Conference on Data Mining (ICDM '15)*, pages 1–10, 2015.
- [4] Bibek Bhattacharai, Hang Liu, and H. Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19*, page 1447–1462, New York, NY, USA, 2019. Association for Computing Machinery.
- [5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '16)*, pages 1199–1214, 2016.
- [6] Bjorn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In *Advances in Knowledge Discovery and Data Mining: 12th Pacific-Asia Conference*, volume 5012, pages 858–863, 2008.
- [7] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: An efficient task-oriented graph mining system. In *Proceedings of the European Conference on Computer Systems (EuroSys '18)*, pages 32:1–32:12, 2018.
- [8] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and

- gpu. *Proc. VLDB Endow.*, 13(10):1190–1205, April 2020.
- [9] Xu Cheng, C. Dale, and Jiangchuan Liu. Statistics and social network of youtube videos. In Hans van den Berg and Gunnar Karlsson, editors, *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 229–238. IEEE, June 2008.
  - [10] Wei-Ta Chu and Ming-Hung Tsai. Visual pattern discovery for architecture image classification and product image search. In *Proceedings of the ACM International Conference on Multimedia Retrieval (ICMR '12)*, pages 1–8, 2012.
  - [11] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, pages 918–934, 2019.
  - [12] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '19)*, pages 1357–1374, 2019.
  - [13] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. In *Proceedings of the VLDB Endowment (PVLDB '14)*, pages 517–528, 2014.
  - [14] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu. Frequent subgraph based familial classification of android malware. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–35, 2016.
  - [15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, pages 17–30, 2012.
  - [16] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, pages 599–613, 2014.
  - [17] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Research in Computational Molecular Biology*, pages 92–106, 2007.
  - [18] Bronwyn Hall, Adam Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools. *NBER Working Paper 8498*, 2001.
  - [19] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '19)*, pages 1429–1446, 2019.
  - [20] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '13)*, pages 337–348, 2013.
  - [21] Tomaz Hocevar and Janez Demsar. A combinatorial approach to graphlet counting. *Bioinformatics (Oxford, England)*, 30:559–565, 12 2013.
  - [22] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx.d: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*, pages 1–12, 2015.
  - [23] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 745–761, Carlsbad, CA, 2018.
  - [24] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20*, New York, NY, USA, 2020. Association for Computing Machinery.
  - [25] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '16)*, pages 1231–1245, 2016.
  - [26] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '18)*, pages 411–426, 2018.
  - [27] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC '16*, page 245–257, New York, NY, USA, 2016. Association for Computing Machinery.
  - [28] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, and Google Inc. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '10)*, pages 135–146, 2010.
  - [29] D. Marcus and Y. Shavitt. Rage – a rapid graphlet enumerator for large networks. *Computer Networks*, 56(2):810 – 819, 2012.
  - [30] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '19)*, pages 25:1–25:16, 2019.
  - [31] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 509–523, 2019.
  - [32] Daniel Mawhirter, Bo Wu, Dinesh Mehta, and Chao Ai. Approxg: Fast approximate parallel graphlet counting through accuracy control. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '18)*, pages 533–542, 2018.
  - [33] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, N Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298:824–7, 2002.
  - [34] Aida Mrzic, Pieter Meysman, Wout Bittremieux, Pieter Moris, Boris Cule, Bart Goethals, and Kris Laukens. Grasping frequent subgraph mining for bioinformatics applications. *BioData Mining*, 11(1):20, 2018.
  - [35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 456–471, 2013.
  - [36] M. Rahman, M. A. Bhuiyan, and M. Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2466–2478, 2014.
  - [37] Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. Prunejuice: Pruning trillion-edge graphs to a precise pattern-matching solution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, pages 21:1–21:17, 2018.
  - [38] Rahmtin Rotabi, Krishna Kamath, Jon Kleinberg, and Aneesh Sharma. Detecting strong ties using network motifs. In *Proceedings of the 26th International Conference on World Wide Web Companion, WWW '17 Companion*, page 983–992, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.
  - [39] Amitabha Roy, Laurent Bindshaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 410–424, 2015.
  - [40] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM '13)*, 2013.

- [41] Soumajyoti Sarkar, Ruocheng Guo, and Paulo Shakarian. Using network motifs to characterize temporal network evolution leading to diffusion inhibition. *Social Network Analysis and Mining*, 9(1), April 2019.
- [42] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. Qfrag: Distributed graph search via subgraph isomorphism. In *Proceedings of the Symposium on Cloud Computing (SoCC '17)*, pages 214–228, 2017.
- [43] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, pages 135–146, 2013.
- [44] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulhaga. Arabesque: A system for distributed graph mining. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 425–440, 2015.
- [45] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, pages 237–251, 2017.
- [46] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *Proceedings of SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*, page 861–878, 2014.
- [47] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, page 223–236, 2017.
- [48] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, pages 763–782, 2018.
- [49] Li Wang, Hongying Zhao, Jing Li, Yingqi Xu, Yujia Lan, Wenkang Yin, Xiaolin Liu, Lei Yu, Shihua Lin, Michael Yifei Du, Xia Li, Yun Xiao, and Yunpeng Zhang. Identifying functions and prognostic biomarkers of network motifs marked by diverse chromatin states in human cell lines. *Oncogene*, 39(3):677–689, September 2019.
- [50] X. Wen, W. Chen, Y. Lin, T. Gu, H. Zhang, Y. Li, Y. Yin, and J. Zhang. A maximal clique based multiobjective evolutionary algorithm for overlapping community detection. *IEEE Transactions on Evolutionary Computation*, 21(3):363–377, 2017.
- [51] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.
- [52] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 301–316, 2016.