Accelerating Graph Mining Systems with Subgraph Morphing

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

Guoqing Harry Xu
Computer Science Department
University of California, Los Angeles
California, USA
harryxu@g.ucla.edu

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. These applications are computationally expensive because finding structural patterns requires checking subgraph isomorphism, which is NP-complete. This paper exploits the *sub-structural similarities* across different patterns by employing Subgraph Morphing to accurately infer the results for a given set of patterns from the results of a completely different set of patterns that are less expensive to compute. To enable Subgraph Morphing in practice, we develop efficient query transformation techniques as well as automatic result conversion strategies for different application scenarios. We have implemented Subgraph Morphing in four state-of-the-art graph mining and subgraph matching systems: Peregrine, AutoMine/-GraphZero, GraphPi, and BigJoin; a thorough evaluation demonstrates that Subgraph Morphing improves the performance of these four systems by 34×, 10×, 18×, and 13×, respectively.

CCS Concepts: • **Information systems** → **Computing platforms**; • **Computing methodologies** → **Parallel programming languages**.

Keywords: subgraph exploration, motifs, frequent subgraph mining, graph system performance

1 Introduction

With growing popularity in graph-structured data, graph mining applications are widely used across a variety of domains including bioinformatics, computer vision, malware detection, and social network analysis [11, 14, 44, 45, 52, 55, 67, 68]. These applications require computing the structural properties of a graph by exploring its subgraphs. To improve efficiency and scalability, a number of graph mining systems have been developed recently [7, 9, 10, 12, 19, 26, 40, 60, 66]. At the core of these systems are an efficient runtime that explores the subgraphs of interest, as well as a high-level programming framework that allows users to express customized logic for exploration and computation.

Recent systems like Peregrine [26] and AutoMine [40] take a pattern-based approach where graph mining applications are formulated as pattern matching sub-tasks—the subgraphs of interest (or *patterns*) are expressed directly via the programming model, and each application is written as a set of operations over the subgraphs that match such patterns. In doing so, the underlying system analyzes the patterns and creates efficient exploration plans. During the matching phase, these exploration plans are used to quickly find the matching subgraphs (or *matches*) and compute results based on them. A pattern-based approach not only simplifies the programming effort, but also provides superior performance compared against traditional approaches that explore subgraphs in breadth/depth-first manner.

We observe that the performance of graph mining applications is not only dependent on the patterns queried by the application, but is also sensitive to *system-level* nuances (*e.g.*, pattern matching strategies and optimizations employed) as well as *application-level* characteristics (*e.g.*, application UDF and aggregation functions).

Instead of developing another custom graph mining system with tailored pattern matching strategies, we take a fundamentally different approach. We aim to take advantage of the performance difference when mining seemingly similar patterns by exploiting the *structural similarities* across different patterns. In general, dramatic performance improvements can be achieved if we could devise a general technique that can infer the results of a pattern for which it is expensive to find matches directly (*i.e.*, hard pattern) from those of other patterns where matching is less expensive (*i.e.*, easy pattern).

We propose Subgraph Morphing for graph mining systems – a generic technique that enables structure-aware algebra over patterns to *morph* the queried patterns into a set of alternative patterns, which can then be used to quickly compute accurate results for the original patterns. We make the following key contributions in this paper.

- We expose key factors that impact the performance of graph mining (Section 3). Our observations from benchmarking various graph mining workloads across multiple

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567489

graph mining systems show that the nature of input workloads (input patterns and data graph from the application), the pattern matching engines in graph mining systems, and the processing requirements of mining applications, all together contribute to the final performance. We envision these observations will be useful in building intuition for future research and development on graph mining systems.

- We develop the Subgraph Morphing algebra that shows how patterns can be morphed with guaranteed correct results (Section 4). Our algebra is general as it natively incorporates morphing with arbitrary aggregation operations from graph mining applications, and it generates multiple alternative solutions (which we call *alternative pattern sets*) to exploit different performance characteristics. With such generality, the system-level and application-level nuances can be incorporated to improve the overall performance, which is a key strength of our technique.
- We develop efficient strategies to enable Subgraph Morphing in practice. A major challenge is the exponential search space of alternative pattern sets with different benefits. We generate and navigate through different alternatives methodically using a novel data structure called S-DAG and a greedy algorithm to select *efficient* alternative pattern sets (Section 5). Our approach is backed by cost models that incorporate all the factors discussed above while estimating the performance of alternative patterns.
  Another challenge is that the results for queried patterns must be inferred from the results of alternative patterns, which can be tedious for users to perform in application logic. We develop strategies to seamlessly convert the results by operating on patterns and their matches only, hence removing the need to modify the application logic (Section 6). Our conversion strategies operate efficiently across both the common output modes in graph mining systems: batched mode where aggregation results are output at the end, and streaming mode where matching subgraphs are returned as a continuous output stream.
- We demonstrate the generality and effectiveness of Subgraph Morphing by integrating it into four state-of-the-art graph mining and subgraph matching systems: Peregrine [26], AutoMine/GraphZero [39, 40], GraphPi [57], and BigJoin [4]. Our extensive evaluation on a variety of graph datasets and graph mining applications demonstrates that Subgraph Morphing accelerates these systems by 34× on Peregrine, 10× on AutoMine/-GraphZero, 18× on GraphPi, as well as 13× on BigJoin (Section 7).

To the best of our knowledge, this paper provides the first treatment of exploiting structural similarities across patterns in a general manner in order to improve graph mining systems across various graph mining workloads. A detailed discussion of related works is available in Section 8.

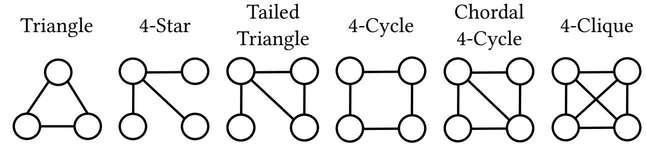


Figure 1. Common pattern names.

2 Background

This section provides an introduction to graph mining concepts necessary for this paper.

Pattern-Centric Graph Mining. Graph mining applications find subgraphs of interest in an input graph G. Such subgraphs are represented as *patterns*. A pattern p is a user-provided graph (or a shape), possibly with labels on its vertices. Apart from regular edges between vertices, a pattern can contain special edges called *anti-edges* [26]. An anti-edge indicates *disconnections* between pairs of vertices. They are used to quickly filter out a subgraph during exploration—a subgraph containing an edge that can match an anti-edge in the pattern is immediately disqualified. An anti-edge is visually represented using a dashed edge in the pattern.

The subgraphs of interest (called matches or *embeddings*) are subgraphs in G that are isomorphic to pattern p, taking both regular edges as well as anti-edges into account.

The isomorphism relation also exists between patterns, but it does not consider anti-edges (*i.e.*, isomorphism between patterns is *w.r.t.* regular edges only). 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.

There are two main strategies to explore subgraphs: *vertex-induced exploration* and *edge-induced exploration*. Vertex-induced exploration explores subgraphs induced by the vertices in the data graph. On the contrary, edge-induced exploration explores subgraphs induced by the edges in the data graph. These two strategies are often used to implement different applications. For example, motif counting uses vertex-induced exploration because a match for a given motif must be *edge-exclusive*—none of its vertices can have an edge in g that does not belong to the motif. On the other hand, frequent subgraph mining uses edge-induced exploration because any subgraph can potentially be frequent.

The exploration strategy can be encoded in the input patterns using anti-edges, controlling which edges can participate in a match. This leads to two types of patterns. A vertex-induced pattern (denoted as p^V) contains anti-edges between each pair of vertices that are not connected by a regular edge. Such a pattern explores subgraphs in the same way as vertex-induced exploration. An edge-induced pattern (denoted as p^E) is a pattern without any anti-edge, and hence, it explores subgraphs in the same way as edge-induced exploration. Note that fully connected patterns (*i.e.*, cliques) are simultaneously edge- and vertex-induced since there

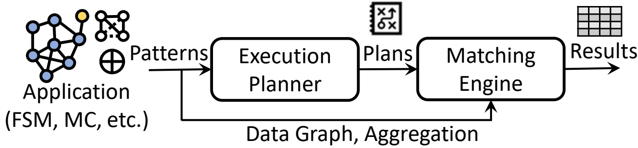


Figure 2. Graph mining workflow.

exists an edge between any pair of vertices (and hence no anti-edge). For any given pattern p, we refer to p^E and p^V as variants of each other. Throughout the paper, we omit the superscript on patterns when the discussion applies to both vertex-induced and edge-induced patterns.

Several patterns can be easily described by their names (*e.g.*, triangle, clique, *etc.*). Figure 1 summarizes the common pattern names that we use in our paper.

Graph Mining Workflow & Applications. Graph mining applications find subgraphs that match patterns of interest and either enumerate (*i.e.*, list out) the individual matches, or perform aggregation operations like counting, computing support, *etc.*, over these instances. Figure 2 shows the high-level workflow of recent pattern-based graph mining systems like Peregrine [26]. At a high level, these systems contain two key modules: an execution planner and a matching engine. The execution planner analyzes the input patterns from the mining application (and perhaps, the data graph) to generate an efficient exploration plan for finding matching subgraphs. This plan is heavily dependent on the structure of the input patterns (*i.e.*, edges and anti-edges), and describes how to break the data graph into tasks, and how to compute each task independently. Then, the matching engine processes the data graph, distributing exploration tasks among worker threads to efficiently explore the matches in the input graph. Based on the requirements of the graph mining application, the pattern matching module outputs either streams of matches (for enumeration) or aggregation results (*e.g.*, match counts).

There are several graph mining applications like Clique Finding, Subgraph Enumeration **(SE)**, Subgraph Counting **(SC)**, Frequent Subgraph Mining (**FSM**) and Motif Counting (**MC**). FSM and MC are two popular applications that perform aggregation over matches. Below we elaborate on important details for these two applications.

Figure 3 shows the difference in the patterns for MC and FSM. MC counts the *vertex-induced* matches of a certain size (*e.g.*, patterns containing 4 vertices). There are six such patterns (shown on the right side of Figure 3), most containing *anti-edges*. Since the aggregation operation is counting (*i.e.*, *sum*), MC requires constant time for each match. FSM, on the other hand, does not have a set of known patterns *a priori*. Instead, it computes frequencies for all subgraphs up to a certain size, based on *edge-induced* matches. The left side of Figure 3 shows a set of 5 patterns, each of which is simply represented by a set of edges. After matches are found, FSM

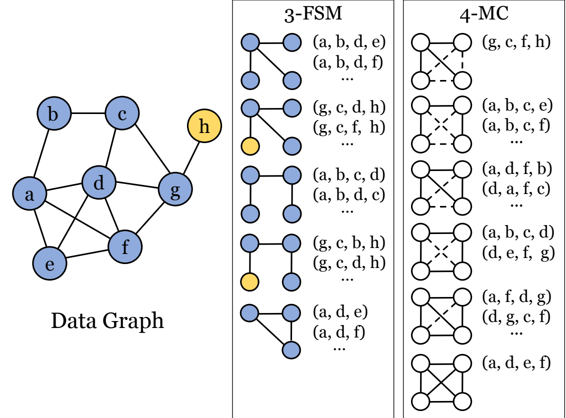


Figure 3. Examples of vertex- and edge-induced patterns in graph mining applications. 3-FSM typically explores (labeled) edge-induced subgraphs; there are three size-3 pattern topologies which form five patterns with distinct labelings. 4-MC typically explores (unlabeled) vertex-induced subgraphs, and there are six size-4 patterns.

computes the *minimum node image (MNI)* [8] of their corresponding patterns, which is used to check whether a pattern is frequent enough.

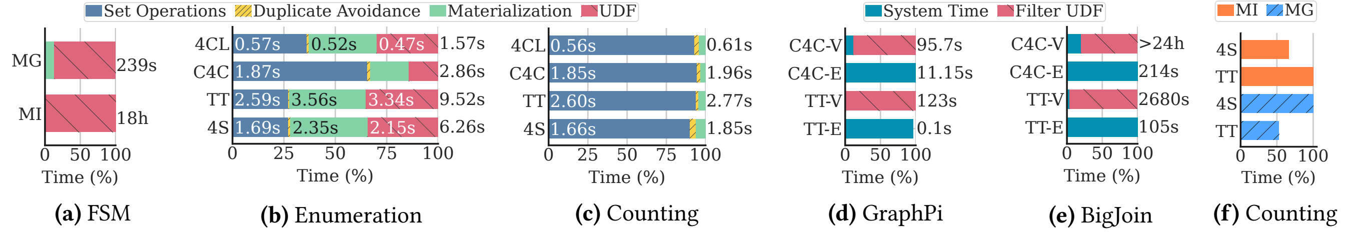
The MNI computation consists of a table with a column for each group of symmetric vertices in p. The data vertices for each match are maintained in respective columns, and the *support* is computed as the size of the smallest column in the table. Hence, to compute this support, the aggregation operation is to join tables by concatenating their respective columns. Each column contains $O(|V|)$ vertices which are merged during aggregation. As a result, while each match requires constant time (to append a value to the table), joining the tables can take $O(|V|)$ time.

3 Performance Analysis

This section identifies key factors that impact the performance of graph mining workloads in order to motivate the need for a generic technique to accelerate arbitrary graph mining workloads across different graph mining systems. To understand the performance bottlenecks in existing systems, we profiled various graph mining workloads on different open-source systems: Peregrine [26], GraphPi [57], and BigJoin [4] for subgraph matching. Figure 4 shows the profiling results, and we summarize our observations below.

3.1 Graph Mining Applications

Figure 4a, Figure 4b and Figure 4c show the performance of three graph mining applications on Peregrine: Frequent Subgraph Mining (FSM), Subgraph Counting (SC) and Subgraph Enumeration (SE). These applications differ widely from each other. FSM invokes a user-defined function (UDF) on each match to compute MNI of patterns, whereas SC does not invoke any UDF since the system natively performs counting using set optimizations. SE is between FSM and SC



(a) FSM **(b)** Enumeration **(c)** Counting **(d)** GraphPi **(e)** BigJoin **(f)** Counting

Figure 4. Profiling graph mining systems. Figures (a-c) show performance breakdown of FSM, SE and SC on Peregrine [26]; (d-e) show performance breakdown of enumerating matches in GraphPi [57] and BigJoin [4]; (f) shows the relative performance of mining patterns on different data graphs in Peregrine (relative *w.r.t.* longer execution for each data graph). MG and MI are MAG and MiCo data graphs (see Figure 11b). 4CL, C4C, TT and 4S are patterns 4-clique, chordal 4-cycle, tailed triangle and 4-star respectively (see Figure 1). The suffixes "-V" and "-E" indicate vertex-induced and edge-induced patterns (*e.g.*, TT-V is vertex-induced tailed triangle).

where it lists out the matches using a UDF, but the UDF is simpler than in FSM.

Observation 1. *Since the number of matches grows exponentially with graph size, invoking UDF on each match impacts the end-to-end processing time. UDFs become the main performance bottleneck when they are expensive (as seen for FSM), while simpler UDFs also influence the processing time by non-trivial amount (as seen for SE).*

The above observation is also valid when mining vertex-induced subgraphs using systems like GraphPi and BigJoin that only perform edge-induced exploration. For these cases, the edge-induced matches mined by the system are processed using a Filter UDF to prune out invalid matches (*i.e.*, matches that do not contain all edges induced by their vertices). As shown in Figure 4[d-e], the Filter UDFs are the main performance bottlenecks, and they significantly slow down the overall processing compared to when matching edge-induced subgraphs (which does not require any UDF).

3.2 Structure of Patterns

Next, we study the performance of SC and SE (Figure 4b and Figure 4c). As expected, the set operations on adjacency lists (set intersections and differences) take most of the time for SC, while SE also spends time on materializing matches by merging sets of candidate vertices.

Since graph mining systems analyze the pattern structure to generate customized pattern-specific matching plans, the structure of the pattern dictates the effectiveness of the sub-techniques involved in the matching plan (*e.g.*, pruning strategies to account for pattern symmetries, or different join fast-paths [28]). Hence, different patterns incur different amount of set operation and materialization time. While one would expect similar looking patterns (*e.g.*, same number of vertices) to have similar performance trends, or denser patterns to be more expensive than sparser patterns with same number of vertices, no such performance trends are guaranteed. For instance, in Figure 4[b-c] we can see:

- A 4-star takes more set operation and materialization time than a 4-clique, even though latter has twice the number of edges than the former (see pattern structures in Figure 1).
- A chordal 4-cycle has only one additional edge over a tailed triangle, but the latter consumes more time for set operation and materialization compared to the former.

Observation 2. *As graph mining systems generate pattern-specific matching plans, even similar-looking patterns incur different amounts of set operations and materialization time, which results in unexpected performance trends across those patterns that are difficult to justify.*

3.3 Structure of Data Graphs

Next, we study the performance of mining different patterns on different data graphs. While mining in larger graphs takes more time (which is expected), the structure of the data graph impacts the relative performance of mining different patterns. This is visible in Figure 4f where mining 4-stars is 25% faster than tailed triangles in MiCo graph, but it is 125% slower in MAG graph. This is because the graph structure influences how different explorations proceed or get pruned (*e.g.*, matching order violation), which in turn impacts the amount of work performed to mine all matches.

Observation 3. *The structure of the data graph also influences the mining performance since it dictates which explorations proceed while others get pruned out.*

3.4 Graph Mining Systems

Finally, we study the relative performance between graph mining workloads across different graph mining systems. Since graph mining systems employ different kinds of pattern matching techniques (*e.g.*, matching algorithms) and are implemented and optimized in different manner (*e.g.*, parallelization strategies), the performance relationships across workloads varies across the systems as well. This is observed when comparing the performance numbers in Figure 4[b-d] for Peregrine and GraphPi: while the chordal 4-cycle is faster than a tailed triangle in Peregrine, the performance relation is reverse for GraphPi where tailed triangle is faster.

Observation 4. *The design and implementation choices incorporated in graph mining systems impacts the relative performance between different graph mining workloads.*

3.5 Motivation Summary

In summary, the end-to-end processing time is influenced by: (a) the structure of patterns and the data graph, (b) the matching strategies and optimizations employed by the mining system, and (c) the processing requirements of the graph mining application (*i.e.*, UDF calls). More importantly, none of these factors are a clear single variable that should be optimized over the other, making it difficult to improve the performance of mining systems.

This motivates the need for a general technique that graph mining systems can employ across various graph mining workloads. Subgraph Morphing is our general technique. It first methodically exposes the space of performance opportunities (Section 4) and then considers the system-level and application-specific nuances to deliver high performance across different scenarios (Section 5 and Section 6).

4 Subgraph Morphing

This section describes principles of subgraph morphing with illustrative examples, and develops its semantics.

4.1 Overview

Subgraph Morphing primarily exploits the structural similarities across different patterns. The key idea is to morph the input patterns from graph mining applications into *alternative patterns* that are less expensive to compute, and then convert the results for those alternative patterns into results for the original input patterns.

When Subgraph Morphing is integrated in graph mining systems, their workflow gets enhanced with two new steps (shown in Figure 5): pattern transformation and result transformation. Instead of being directly passed to the execution planner, the input patterns first undergo a pattern transformation step resulting in alternative patterns. Then, matching plans are computed and followed to explore matches for the alternative patterns from the input graph. Finally, the results for alternative patterns are sent through the result transformation step to compute the results for the original input patterns. Details of pattern transformation and result transformation will be explained in Section 5 and Section 6 respectively, and Appendix A illustrates the key steps in Subgraph Morphing using two graph mining applications. In this section, we will focus on the semantics of Subgraph Morphing.

4.2 Intuition & Example

The intuition behind Subgraph Morphing can be summarized with the following two key observations.

[P1] A match for an edge-induced pattern p^E on n vertices is also a match for all of its subpatterns on these vertices.

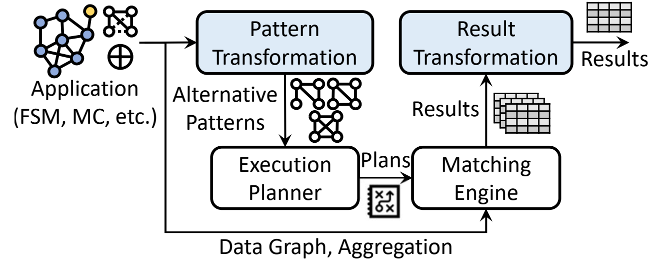


Figure 5. Graph mining with Subgraph Morphing.

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 observation does *not* apply to vertex-induced patterns—although a vertex-induced 4-cycle contains the same four (regular) edges, the additional two anti-edges exclude matches for 4-cliques.

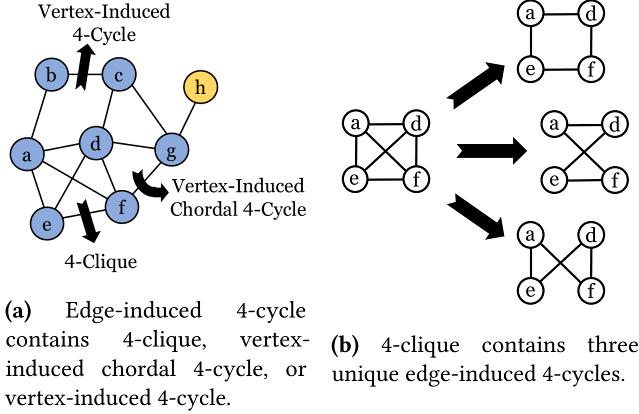
[P2] A match for a vertex-induced pattern p^V is always a match for the corresponding edge-induced pattern p^E, but *not* vice versa—p^V matches exactly the edges in p^E but a subgraph that matches p^E may contain additional edges that are against the anti-edges in p^V.

Example. These observations indicate 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 (*i.e.*, subgraph) for the edge-induced 4-cycle. The vertices in this match may have edges that are in the graph but not present in the pattern. If these edges do not exist, this is also a match for the vertex-induced 4-cycle (*e.g.*, a-b-c-d in Figure 6a). If there exists exactly one extra edge, it is a match for the vertex-induced chordal 4-cycle (*e.g.*, d-c-g-f in Figure 6a). Finally, if there are two extra edges in the match, it is a match for the 4-clique (*e.g.*, a-d-f-e in Figure 6a). These situations are *mutually exclusive* since they depend on specific edges being present or absent.

While the above partitioning enables converting matches, a match for a given pattern can potentially contain multiple matches for another pattern. For example, a match for the 4-clique contains 3 unique matches for the edge-induced 4-cycle, as shown in Figure 6b. Hence, in order to convert a match for the 4-clique into a match for the 4-cycle, we must correctly map the 4-clique vertices, using the subgraph isomorphisms, to those of the 4-cycle.

4.3 Semantics

Subgraph Morphing performs structure-aware algebra over patterns (and hence, their matches) to capture all matches for a given pattern in the input graph by converting the matches of different (alternative) patterns. One way to find alternative patterns for a given pattern is to consider its *superpatterns* because matches of a pattern are guaranteed to contain valid matches for its subpatterns. Hence, our first idea is to derive the matches of the pattern using its superpatterns.



(a) Edge-induced 4-cycle contains 4-clique, vertex-induced chordal 4-cycle, or vertex-induced 4-cycle.

(b) 4-clique contains three unique edge-induced 4-cycles.

Figure 6. Identifying matches for different patterns.

Let M be the set of all matches for a given pattern p. Then,

$$M(p^E) \;=\; M(p^V) \;\; \cup \bigcup_{q^E \supset_n p^E} M(q^V) \circ \phi(p^E, q^E) \qquad (1)$$

where $q^E \supset_n p^E$ indicates the superpatterns q^E of pattern p^E containing same number of vertices (n), $\phi(p^E, q^E)$ captures the set of all subgraph isomorphisms from p^E to q^E, and $M(q^V) \circ \phi(p^E, q^E)$ permutes the vertices in each match $m \in M(q^V)$ according to subgraph isomorphism from p^E to q^E.

In simple words, given an edge-induced pattern p^E, we start with using the matches of its vertex-induced variant p^V (observation [P2]). However, since p^V contains anti-edges that eliminate some valid matches for p^E, we compensate by using matches for additional superpatterns, each obtained by replacing anti-edges in p^V one-by-one with true edges (observation [P1]). This ends up generating an ***alternative pattern set*** for p^E that contains all of its vertex-induced superpatterns with the same number of vertices. Since a match for a superpattern can contain multiple matches for a subpattern (*e.g.*, 4-clique contains three edge-induced 4-cycles in Figure 6b), we use ***permutation functions*** to generate all the matches of the subpattern. A permutation function converts matches of a superpattern into matches for the subpattern based on isomorphic mappings from the subpattern to the superpattern.

Due to space limitation, we skip the theory of Subgraph Morphing and its preservation of full pattern equivalency (*e.g.*, proof for Eq. 1). Next, we focus on the two key aspects that make our Subgraph Morphing strategy generic: directly converting arbitrary aggregations results and multiple alternatives for converting matches.

Converting Aggregation Results. Subgraph Morphing can be applied directly on aggregation results instead of converting the individual matches. By doing so, we can prevent materialization of a significant number of matches, and reduce UDF overheads while computing aggregations by invoking them on fewer match results.

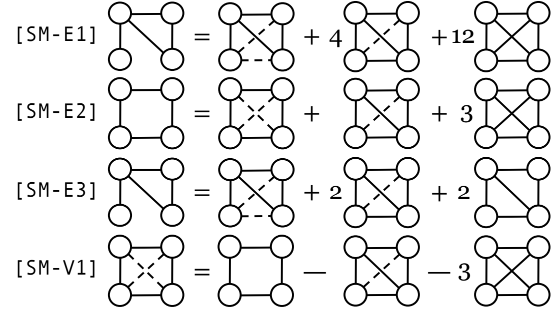


Figure 7. Sample equations resulting from subgraph morphing. [SM-V1] morphs vertex-induced pattern (left) whereas other equations morph edge-induced patterns. [SM-E1] and [SM-E2] are directly obtained from Eq. 1, [SM-E3] by recursively substituting in [SM-E1], and [SM-V1] by adjusting [SM-E2]. The coefficients indicate the numbers of unique matches resulting from subgraph isomorphism.

Let $a = (\lambda, \oplus)$ be the aggregation in graph mining applications where λ is a map from a set of matches to an aggregation value and $\oplus$ is a commutative operator for combining aggregation values. For example, in counting aggregation λ is the number of matches in the set (*i.e.*, $\lambda(M) = |M|$) and $\oplus$ is the traditional integer sum. In FSM application on the other hand, λ computes the MNI table of a set of matches and $\oplus$ joins tables on columns. For a set of matches $M(p)$, we write $a(M(p))$ as a shorthand for $\bigoplus_{m \in M(p)} \lambda(m)$. The aggregation results can be directly converted as follows:

$$a(M(p^E)) = a(M(p^V)) \oplus \Bigg(\bigoplus_{q^E \supset_n p^E} \; \bigoplus_{\substack{f \in \\ \phi(p^E, q^E)}} a(M(q^V)) \circ_* f \Bigg) \quad (2)$$

where $\circ_*$ is a *permute* operator for aggregation values to account for the permutations according to $\phi(p, q)$ (similar to $\circ$ defined on matches in Eq. 1). The $\circ_*$ operator adjusts the aggregation value based on a given permutation f in $\phi(p, q)$. The definition of the permutation function depends on the aggregation operation performed on the matches. For instance, the permutation function for *counting* accounts for all unique isomorphic mappings, which results in multiplying the number of matches of a superpattern by the number of unique isomorphic mappings. In FSM, the permutation function permutes the columns of the MNI table of the match, in a similar manner, based on subgraph isomorphism.

Multiple Alternative Pattern Sets. While Eq. 1 identifies alternative patterns for edge-induced patterns, we can move in the other direction as well to compute results for vertex-induced patterns (achieved by rearranging the terms in Eq. 1 to bring $M(p^V)$ on left-hand side). More importantly, the patterns in the alternative pattern set can be iteratively substituted with their conversion equations to obtain different alternative pattern sets. By recursively substituting the patterns with their alternative patterns sets, we can generate a system of equations representing different alternative

pattern sets that can be used to compute the results for a given query pattern. Additionally, the alternative pattern sets can contain a combination of vertex-induced and edge-induced patterns by converting the intermediate aggregations through recursive substitution.

Figure 7 shows a few samples of how patterns can be morphed to a given pattern. The coefficient associated with a pattern indicates the number of unique matches resulting from subgraph isomorphism (*e.g.*, 4-clique has three 4-cycles, and hence the 4-clique in Equation [SM-E2] has a coefficient 3). As we can see, [SM-E1] and [SM-E3] are two different equations to compute results for the tailed triangle. With different choices for alternative pattern sets, the pattern query can be optimized by selecting the alternative set for which match results can be efficiently computed (discussed in Section 5).

4.4 Significance of Generic Subgraph Morphing

The generic nature of Subgraph Morphing enables accelerating arbitrary graph mining applications while also incorporating system-level nuances and application-level characteristics. For example, system-level nuances were shown to impact the mining workloads differently in Section 3, causing certain patterns to be faster than others on one system but slower on another system. In such a situation, alternative patterns sets can be optimized differently for each individual system by accounting for their relative performance across different patterns; in our example from observation 4, this would mean choosing tailed triangle over 4-cycle for GraphPi but not for Peregrine. Similarly, application-level characteristics like application UDF and the structure of the data graph were shown to impact the query performance in Section 3. Since Subgraph Morphing enables direct conversion of aggregation results, the impact of these application-level characteristics can be directly accounted in choosing the right set of alternative patterns. For example, expensive UDF calls like filtering each match or computing MNI tables for FSM can be reduced by using alternative patterns that are expected to generate fewer match results. But on the other hand, applications that employ inexpensive aggregation operations like counting (system-native) can benefit from alternative patterns that are expected to incur fewer set operations.

In comparison, counting techniques developed in prior works like [22, 35, 42, 47, 72] are inapplicable for general-purpose graph mining systems since they focus on count conversion rules that are customized for specific types of equations on specific patterns. Hence, they cannot systematically generate multiple alternative sets for a given query pattern, which renders them useless as they cannot account for various system-level and application-level nuances. For instance, blindly converting results from vertex-induced to edge-induced (or vice-versa) would often be slower than the original query, depending on the system and the application.

5 Generating Alternative Pattern Sets

This section describes the pattern transformation step (see Figure 5) to compute alternative patterns.

To fully exploit Subgraph Morphing, our goal is to generate alternative pattern sets that would potentially compute the final results efficiently compared to the original query patterns. This cannot be achieved statically because of two main reasons. First, the query patterns in graph mining applications can change dynamically during runtime. For instance in FSM, only those patterns that have enough matches in the data graph (*i.e.*, cross a support threshold) are extended to generate the new set of patterns to be explored in the next step. And second, the input data graph itself impacts the performance of matching (observation 3 in Section 3). Hence, we dynamically generate the alternative patterns for the query patterns as they become available at runtime.

Since the possible alternative pattern sets grow recursively, the search space for identifying *efficient* alternative pattern sets grows exponentially. For a single query pattern, the choice may appear simple. However, when the input contains multiple query patterns, the number of choices increases exponentially as the alternative sets for different patterns overlap, making it hard to estimate benefits. For instance, two patterns may have a lower cost (*i.e.*, faster to compute) compared to the cost of their individual alternative pattern sets; however, the cost of the combined alternative pattern sets can be lower (due to overlapping patterns) than that of the two patterns.

Exhaustively searching all possible combinations of alternative pattern sets is impractical. Instead, we develop a greedy exploration strategy backed by a cost model that actively prunes the search space. Our approach is to first generate a single alternative set, and then use a cost-based selection strategy to iteratively improve the alternative set by substituting better (low cost) alternatives.

5.1 Initial Alternative Patterns

Given a set of input patterns, we generate the initial alternative pattern set for each pattern in the input set using Eq. 1. This primarily involves generating superpatterns of the input pattern (second term in Eq. 1). While the final alternative pattern set contains a mix of vertex-induced and edge-induced patterns, the choice of each individual pattern being either vertex-induced or edge-induced is independent of the other patterns in the set. Hence, we generate edge-induced superpatterns, and later optimize the choice for each pattern when constructing the efficient alternative pattern set. By doing so, we maximize the overlap between the alternative patterns generated across different patterns in the input set, which is beneficial since the same superpattern (or its alternatives) covers multiple input patterns.

The superpatterns of the input pattern are generated by extending them recursively, adding edges between disconnected vertices. Naïvely extending the input patterns can

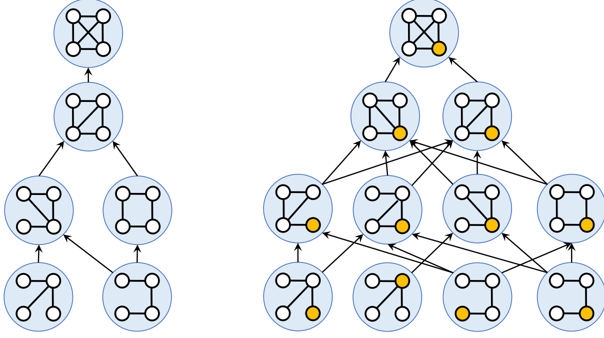


Figure 8. S-DAG for unlabeled patterns (on left), and for patterns with one yellow labeled vertex (on right).

end up generating duplicate patterns due to two reasons. First, adding edges between automorphic (or symmetric) sub-components of a given pattern can result in the same superpatterns; for instance, adding an edge between any pair of disconnected vertices in a 4-cycle would result in the same chordal 4-cycle pattern. Second, different patterns with the same number of vertices have a common subset of superpatterns; for instance, a 4-cycle and a tailed triangle both have the 4-clique and chordal 4-cycle as their superpatterns.

We avoid generating redundant superpatterns by maintaining them in the S-DAG data structure, as described next.

S-DAG for Superpattern Sets. As superpatterns get generated, we memoize them and their superpattern-subpattern relationships in form of a directed acyclic graph, called S-DAG. The S-DAG is queried each time before recursively extending any pattern and adding new superpatterns in order to avoid redundant pattern alternatives.

Each vertex of the S-DAG represents a pattern (either one of the input patterns or one of their superpatterns), and we draw directed edges from each pattern with k edges to its superpatterns with $k+1$ edges. Figure 8 shows two S-DAG examples: one where patterns are unlabeled, and other where patterns are labeled. Depending on the number of labels in the pattern, the number of possible patterns increases compared to the number of unlabeled patterns, and each labeled pattern can have many more superpatterns than an unlabeled one.

For fast comparison and lookup operations on S-DAG, we represent the patterns using 64-bit pattern IDs which uniquely correspond to the pattern structures. Each pattern is first *canonicalized* (using Bliss library [29]) so that its vertices have consistent vertex IDs. Then, the edges of the canonicalized pattern are hashed consistently to compute its pattern ID that uniquely identifies the pattern structure. While pattern IDs can be computed quickly (in milliseconds) as patterns get generated, they can also be computed offline.

5.2 Selecting Efficient Alternative Patterns

Once the S-DAG is generated, the final alternative pattern set is constructed by carefully selecting the set of patterns

Algorithm 1 Efficient Alternative Patterns

```
    Input: Set of query patterns P and their S-DAG^P
    Output: Low-cost alternative pattern set S
 1: INITIALIZEPATTERNCOSTS(S-DAG^P)
 2: procedure SELECTPATTERNS(P, S-DAG^P)
 3:   S ← P
 4:   while S not converged do
 5:     for each p ∈ ∪_{s∈S} PARENTS(S-DAG^P, s) do
 6:       for each C ∈ ℙ(CHILDREN(S-DAG^P, p)) where C ⊆ S do
 7:         cost_C ← Σ_{c∈C} INITIAL_COST(c)
 8:         SPC ← ∪_{c∈C} SUPERPATTERNS(c)
 9:         cost_SPC ← Σ_{spc∈SPC} COST(spc)
10:         if cost_SPC < cost_C then
11:           S ← (S \ C) ∪ SPC
12:           for each c ∈ C ∪ SPC do
13:             SETCOST(S-DAG^P, c, 0)
14:           end for
15:         end if
16:       end for
17:     end for
18:   end while
19:   return S
20: end procedure
```

based on the potential performance benefits they can bring. To avoid evaluating every possible alternative set in the exponential search space, we develop a greedy algorithm that iteratively finds better alternatives using the S-DAG. Instead of searching for the optimal alternative pattern set, our goal is to construct an *efficient* alternative set that promises faster execution compared to the original query patterns.

Algorithm 1 shows the selection algorithm. Initially, each node in S-DAG is assigned a cost that estimates the time taken to match that pattern. Since either variant of superpatterns can be used in the alternative pattern set, the nodes are assigned the minimum cost between the two variants of the patterns they represent. Then the algorithm proceeds iteratively to replace patterns with lower cost alternatives.

For each pattern we check whether any subsets of the pattern's children in the S-DAG benefit from morphing. If the combined cost of the children is more than the cost of their combined superpatterns, then the superpatterns are selected for the alternative pattern sets. When the alternative patterns are selected, the S-DAG is re-weighted to reflect that those patterns are free, *i.e.*, their cost is set to 0, and then the patterns are traversed again. The algorithm incrementally reduces the cost of the alternative pattern set until it can no longer be improved. By considering only the subsets of each pattern's children, we reduce the exponential search space to the number of unique subpatterns for each pattern.

Estimating Relative Pattern Costs. In order to identify cheaper pattern alternatives, the selection algorithm requires pattern costs that represent the estimated relative times to match different patterns. The cost of pattern-matching depends not only on the input patterns and the data graph,

but also on the system-specific properties such as the underlying matching algorithm used by the system and on application-specific details like aggregation functions.

Modern graph mining and pattern matching systems like [19, 39, 40, 57] often incorporate a cost model to compute efficient exploration plans for the given patterns. While these models are useful, they do not account for any application-specific detail since it is irrelevant to their matching plans. Since we are interested in costs that estimate the performance of different patterns on a given application workload, for these systems we piggyback on their existing cost model by enhancing them to include cost of result aggregation.

The data graph is modeled by an abstract probabilistic graph, where two vertices are connected by an edge with a fixed probability. The pattern matching process is modeled as a series of $V(P)$ nested loops over this abstract graph. After computing the expected number of iterations in each loop, keeping in mind previous loops, the final cost is simply the number of iterations in the innermost loop.

As aggregations are functions on individual matches, their costs are modeled as the number of estimated matches multiplied by the amount of work for the aggregation. The number of estimated matches is already available from the cost model of the underlying system. The amount of work for the aggregation can be estimated by profiling the application-specific UDF to identify how aggregation time scales with the number of matches, or by analyzing the aggregation operations. For profiling, a set of n dummy matches can be generated by randomly selecting $|V(P)|$ vertices n times, and then the time required to apply the UDF to these n matches gets measured. Repeating this for varying n and integrating the resulting curve yields an approximation of how the aggregation scales. Alternatively, the cost of aggregation for simple or well-known operations can be directly provided as hints to the system. For instance, the counting aggregation is performed using a constant operation per match, and hence incurs no additional cost. On the other hand, FSM application incurs $O(|V(G)|)$ cost to approximate the overheads of merging MNI tables.

Finally, for systems like [4, 26] that do not use any cost model, we compute pattern costs based on their pattern matching details using a similar approach as [40]. In addition, we improve the cost estimation using two novel enhancements.

- From profiling, we observed that highest degree data vertices (those in the 95th percentile) contribute the majority (66–99%) of the matches and the majority of the execution time. Hence, the graph model is restricted to the portion of the data graph comprising the highest degree vertices.
- Since partial orders for symmetry breaking [18] impact the input size (*e.g.*, adjacency lists or indexed tuples) for the set operations or joins performed during matching, the neighborhood size is estimated in terms of the expected number of smaller or higher vertex id neighbors.

Algorithm 2 Converting Aggregation Results

```
Input: Set of query patterns P, their alternative pattern set S,
       and aggregation store A holding results of S
Output: Aggregation store R for P
 1: procedure convertResults(P, S, A)
 2:   for each p ∈ P do
 3:     for each q ∈ alternative(S, p) do
 4:       for each q_key ∈ AggrKeyMap(q) do
 5:         for each f ∈ φ(p, q) do
 6:           p_key ← permute(q_key, f)
 7:           R[p_key] ← reduce(R[p_key], A[q_key])
 8:         end for
 9:       end for
10:     end for
11:   end for
12:   return R
13: end procedure
```

6 Transforming Results

The efficient alternative pattern set is given as input to the pattern matching engine. The next step is to convert the match results generated for these alternative patterns into results for the original query patterns (result transformation in Figure 5). As discussed in Section 4.3, we use *permutation functions* (*i.e.*, $\phi(p,q)$ in Eq. 1 and Eq. 2) that convert the results based on isomorphic mappings from the query pattern to the alternative pattern. For seamless conversion, our key insight is to permute the vertex ids in the pattern instead of modifying the results so that the results get mapped for original patterns. We will describe our result conversion strategies in Section 6.1 and Section 6.2.

Output Modes. Graph mining systems often employ different output modes for returning match results that are suitable for different applications. For example, applications like FSM and MC return the final aggregation values (counts, support values) in the end after all required matches are found. On the other hand, applications like SE return each individual match to the user function for further processing (*e.g.*, filtering) as the matches get explored. To handle these output requirements, our permutation functions can be employed to convert the results either on-the-fly or after matching finishes.

6.1 Post-Matching Conversion

In this case, results get converted after matching for the alternative patterns completes. Then the converted results are returned to output.

Since the final results are often computed by application-specific aggregation functions, one way to convert the results would be by directly modifying the aggregation functions (*e.g.*, map-reduce UDF) to simply change the mapping between results and patterns. While such a change is easy, it still requires capturing the relationship between the query patterns and their alternative patterns into aggregation functions.

```
void process(Pattern p, Match m) {
  for (PatternVertex u : p.getVertices())
    map(u, m(u));
}
MNIColumn reduce(MNIColumn accumulator,
                 MNIColumn new_value) {
  return accumulator.merge(new_value);
}
```

Figure 9. FSM Application.

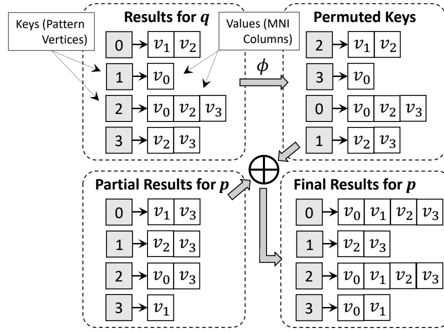


Figure 10. Converting MNI aggregation for FSM.

To make the conversion process seamless to the application, we instead modify the vertex ids in the patterns, which does not require modifying application-specific logic. This is achieved by applying the permutation function on vertex ids of the alternative patterns. By doing so, we simply invoke the aggregation operation on the query pattern, but with permuted ids, which ends up correctly routing the aggregation results from alternative patterns to the query patterns.

Algorithm 2 shows the result conversion process. The results for alternative patterns are maintained in the aggregation store A as key-value pairs, where keys are of different types depending on the application (*e.g.*, patterns for SC, pattern vertices for MNI). To convert the results (lines 5-8), the aggregation keys for the alternative pattern are permuted based on the permutation function to obtain the key for input pattern, and then the aggregation values for those keys are combined using the application's aggregation function (reduce on line 7).

Example. We illustrate how conversion happens for the MNI aggregation in FSM. To help understand the context, the FSM application code is shown in Figure 9. In the process function, the individual vertices are mapped to the respective pattern vertices, and the reduce operation merges the set of pattern vertices (*i.e.*, MNI columns).

Figure 10 shows an example of result conversion. The permutation function permutes the vertex ids for pattern q which results in a change in the mappings between keys (pattern vertices) and values (MNI columns). Hence, for the results on top right, the column for vertex 0 gets remapped to the third column. Then, the aggregation function merges the columns for p with the permuted aggregation.

Algorithm 3 Converting Matches On-the-Fly

1: // Send alternative pattern q to matching engine
2: // Matching engine starts finding matches
3: // Matching engine generates match m for pattern q
4: **for** *each* $f \in \phi(p, q)$ **do**
5: $\quad m_permuted \leftarrow \text{PERMUTE}(m, f)$
6: $\quad \text{PROCESS}(p, m_permuted)$ /* Call UDF */
7: **end for**

6.2 On-the-Fly Conversion

Here, the results are converted as they get generated by the matching engine, and then the converted results are sent down the application's processing pipeline.

While we can employ the same strategy of converting patterns instead of converting the results, it is possible to directly convert the match generated by the matching engine since it is yet not been modified by the application-specific functions (*i.e.*, converting the match does not require application details). Algorithm 3 shows on-the-fly conversion of matches. Instead of directly calling the application function with the alternative pattern q and its match m, the permutation function is applied on m which generates the match for the query pattern p. These are then supplied to the application function (process on line 6).

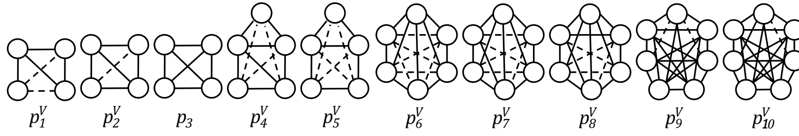
7 Evaluation

Mining Systems and Implementation Details. We integrated Subgraph Morphing in four state-of-the-art graph mining and subgraph matching systems: Peregrine [26], AutoZero (an implementation of techniques in both AutoMine [40] and GraphZero [39]), GraphPi [57], and BigJoin [4]. Subgraph Morphing is generally applicable to other mining and pattern matching systems like [19, 38] as well. Systems like Arabesque [60], Fractal [12] and others [9, 10, 66] perform generic BFS or DFS explorations that do not exploit the pattern structure in order to optimize exploration. Hence, they deliver similar performance across different patterns of same size, providing little or no opportunity to exploit performance difference across patterns.

We evaluate using the available mining capabilities of each of the four systems to cover all cases. Hence, we use Peregrine and AutoZero for counting motifs and patterns, GraphPi and BigJoin when counting vertex-induced patterns with a UDF to filter, and Peregrine for subgraph enumeration and FSM.

Subgraph Morphing was added in form of two modules external to the pattern matching engines in these systems (see Figure 5), *i.e.*, the pattern matching strategies and optimizations in these systems were left untouched.

AutoMine [40] uses a compilation-based approach to generate matching schedules and GraphZero [39] enhances the schedules using symmetry breaking (similar to [18]). Since neither AutoMine nor GraphZero have source code available, we developed an in-house version by faithfully implementing the symmetry breaking restrictions and performance

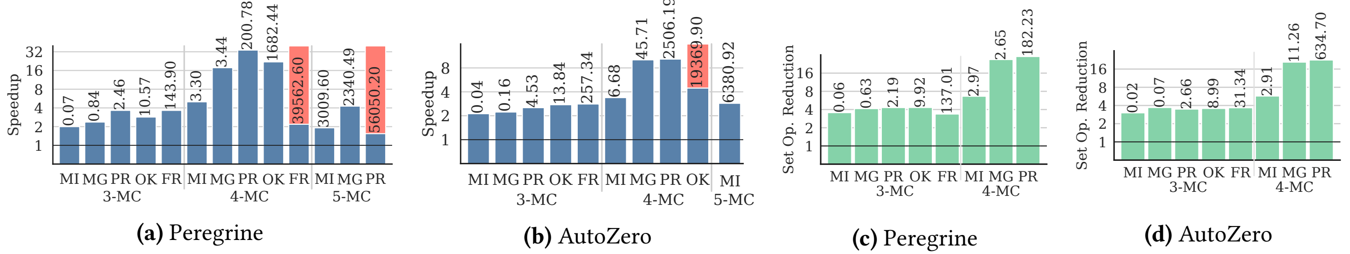


(a) Vertex-induced patterns used in evaluation. The edge-induced variants do not contain anti-edges.

G	$\|V(G)\|$	$\|E(G)\|$	Num. Labels	Max. Deg.	Avg. Deg.
(MI) MiCo [13]	100K	1M	29	1359	22
(MG) MAG [24]	726K	5.4M	349	4779	14
(PR) Products [24]	2.4M	61M	47	17481	52
(OK) Orkut [70]	3M	117M	—	33133	76
(FR) Friendster [70]	65M	1.8B	—	5214	55

(b) Real-world graphs used in evaluation.

Figure 11



(a) Peregrine **(b)** AutoZero **(c)** Peregrine **(d)** AutoZero

Figure 12. Performance improvements from Subgraph Morphing in Peregrine & AutoZero for Motif Counting. (a-b): speedups are *w.r.t.* baseline system (without morphing); absolute times (in seconds) for when Subgraph Morphing is enabled are shown on top of the bars. Red bars indicate the cases where baseline did not finish within 24 hours (*i.e.*, speedups for those cases are underestimated). (c-d): reductions in set operation times are *w.r.t.* baseline system; absolute times (in seconds) are shown.

model for choosing individual pattern schedules from [39]. Unlike AutoMine however, GraphZero does not merge the schedules of multiple input patterns. Hence we augmented our in-house implementation with schedule-merging, so that overlapping loops in different pattern schedules are merged together, and conflicting restrictions are applied separately to avoid under-counting. We name this augmented implementation AutoZero. AutoZero directly generates C++ code for pattern matching schedules, and invokes g++ version 10 to compile it. Across all the experiments, we did not measure the C++ code generation and compilation time for AutoZero.

Experimental Setup. We investigate the impact of Subgraph Morphing on the performance bottlenecks identified in Section 3 through experiments on a wide array of applications: Motif Counting (MC) of size 3 to 5 vertices, FSM with size-3 and -4, as well as Subgraph Counting (SC) and Enumeration (SE) with patterns in Figure 11a. Most of these patterns have been used in state-of-the-art evaluations [12, 57], and we have also included some larger and denser patterns in order to stress the systems.

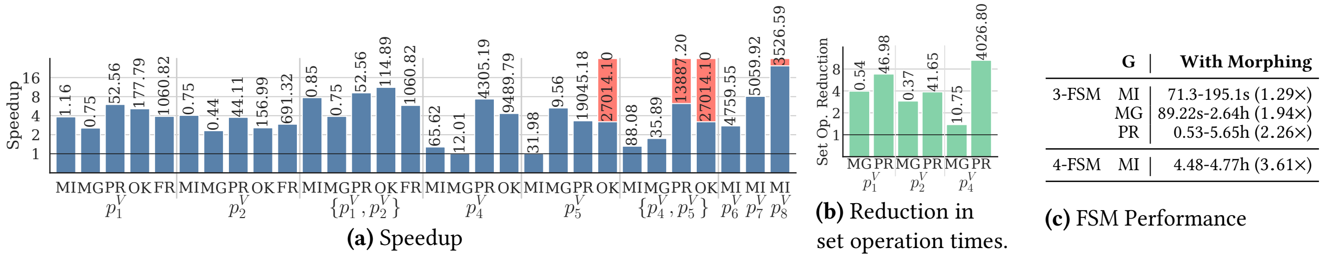
Figure 11b lists the data graphs used in our evaluation. MiCo (MI) is a co-authorship graph labeled with each author's research field. MAG (MG) is an academic graph composed of several vertex types. We use the portion representing citations between papers, where papers are labeled by the venue they were published in. Products (PR) is a co-purchasing network, where vertices represent products, labeled by their category, and edges indicate two products are purchased together. Orkut (OK) and Friendster (FR) are unlabeled social network graphs where edges represent friendships between users. MiCo, Orkut and Friendster have been used to evaluate previous systems [12, 19, 26, 40, 60], while MAG and Products are recent graph datasets designed to evaluate data mining tasks [24].

All our experiments were run on a Google Cloud `n2-highcpu-32` instance, equipped with a 2.8GHz Intel Cascade Lake processor with 32 logical cores and 32GB of RAM. Across all experiments, we measured the end-to-end execution time, which includes input pattern transformation, mining computation, as well as result transformation. Since pattern transformation is done on the input patterns, we observed this phase took little time—for instance, transforming patterns of size 4 and 5 took at most 0.7 ms and 7.2 ms, respectively, whereas finding matches for those patterns on large graphs often takes 10s-1000s of seconds.

7.1 Morphing for Reducing Set Operation Time

Since counting is heavily bottlenecked by set operations in both Peregrine and AutoZero, we use **Motif Counting (MC)** as a representative benchmark. Figure 12 summarizes the results. Figure 12c and Figure 12d show that execution time for motif counting is dominated by set operations.

Compared to the baseline systems, applying Subgraph Morphing reduced set operation time in AutoZero and Peregrine by $3-22\times$ and $3.5-30\times$, respectively. This is because morphing the vertex-induced patterns in Motif Counting results in alternative pattern sets which contain fewer anti-edges. While anti-edges actively prune the search space and reduce the number of matches generated, each anti-edge necessitates an additional set operation (set difference) in the matching plan. Our selection algorithm identifies that the additional time required for set operations is not justified by the reduction in the number of matches since the counting aggregation is inexpensive.



(a) Speedup

(b) Reduction in set operation times.

	G	With Morphing
3-FSM	MI	71.3-195.1s (1.29×)
	MG	89.22s-2.64h (1.94×)
	PR	0.53-5.65h (2.26×)
4-FSM	MI	4.48-4.77h (3.61×)

(c) FSM Performance

Figure 13. Performance improvements from Subgraph Morphing in Peregrine for Subgraph Counting (Figures a-b) and Frequent Subgraph Mining (Figure c). For FSM, *minimum* speedups are reported in brackets. Note that 4-FSM on larger graphs did not finish in 24 hours since the complexity grows exponentially, requiring more resources (more machines and time).

The reduced set operation times translate to significant speedups in overall execution times, as seen in Figure 12a and Figure 12b. Subgraph Morphing yields a maximum 34× speedup in Peregrine for 4-MC on PR. The smallest speedup with Peregrine, 1.5× for 5-MC on PR, is an underestimation since the baseline Peregrine (without Subgraph Morphing) did not finish counting even half the patterns in 24 hours. In AutoZero, Subgraph Morphing yields 2 − 10× speedups for motif counting, including a conservative 5× speedup in the OK 4-MC case, which the baseline system could not complete in 24 hours.

Subgraph Counting (SC). Motif Counting represents a best-case scenario for Subgraph Morphing, since all superpatterns are already contained in the input pattern set. Here we examine the converse situation in Figure 13(a-b), matching single patterns and pairs of patterns from Figure 11a, such that few or no superpatterns are part of the input set. We use Peregrine for these experiments, since it matches patterns one by one, further exacerbating the cost of extra superpatterns. Due to limited space, we skip AutoZero, which gives the best case for Subgraph Morphing since merged matching plans significantly reduce the cost of extra superpatterns. Even with higher costs for superpatterns, Figure 13a shows that Subgraph Morphing speeds up Peregrine executions by 1.2 − 24×. The greatest speedup came from the large p_8^V pattern, which Peregrine could not mine without morphing. As expected, set operations are still the main bottleneck when matching individual patterns, and Subgraph Morphing reduces the time spent on them by 1.3 − 10× (shown in Figure 13b), despite having to match extra patterns.

7.2 Morphing for Reducing UDF Overheads

We evaluate the effectiveness of Subgraph Morphing to address the key bottleneck in **Frequent Subgraph Mining (FSM)** and filter-based mining.

Reducing UDF overheads in FSM. Figure 13c summarizes the performance results when employing Subgraph Morphing in Peregrine for FSM. Subgraph Morphing alleviates the UDF bottleneck in 4-FSM on MiCo by morphing the patterns predicted to be most frequent into vertex-induced variants which will have fewer matches. For example, the edge-induced 4-Star pattern with all of its vertices sharing the most frequent label in the data graph is one of the most expensive patterns to mine in MiCo FSM due to the few constraints in the pattern combined with the frequent labeling, leading to over 5.7B matches. Morphing it generates 3 additional superpatterns (Tailed Triangle, Chordal 4-Cycle, and 4-Clique, all vertex-induced and with the same labeling), but results in 1.4B fewer matches, and as many fewer UDF calls (a reduction of 24%).

Similarly for other expensive patterns, the morphed patterns saved over 13 hours of time spent on UDFs while spending only 1 additional hour on set operations, yielding 3.6× speedup. 3-FSM on MiCo shows the least improvement, as both the patterns and the data graph are small, making the input patterns easy to match. Note that FSM operates on labeled patterns, which generally require more superpatterns during morphing.

Eliminating Filter UDFs. We apply Subgraph Morphing to vertex-induced pattern matching with GraphPi [57] and BigJoin [4]. These systems lack native support for mining vertex-induced patterns; hence, extracting vertex-induced results requires matching the edge-induced variants and using a Filter UDF to remove matches with extra edges. With Subgraph Morphing, we compute vertex-induced results with edge-induced patterns without invoking any UDFs.

As shown in Figure 14, Subgraph Morphing significantly speeds up GraphPi and BigJoin, by 1.4 − 18× and 6.3 − 13.3× respectively. This is because the native matching capabilities in these systems outweigh the expensive edge lookups in Filter UDFs even when multiple patterns must be matched.

We observed that 98% of execution time in the baseline system (without Subgraph Morphing) was spent in UDF calls. Drilling deeper reveals that the poor performance is primarily due to branches incurred on every match by the Filter UDF. Figure 14c and Figure 14d show that eliminating the UDFs using Subgraph Morphing reduces the number of branch misses by 30× on average (1.7 − 88×).

7.3 On-the-Fly Conversion

We evaluate the benefits of Subgraph Morphing for **Subgraph Enumeration (SE)** where on-the-fly conversion is employed to handle a stream of matches. We used Peregrine to enumerate matches comprised of vertices whose average weight is within a standard deviation of the distribution

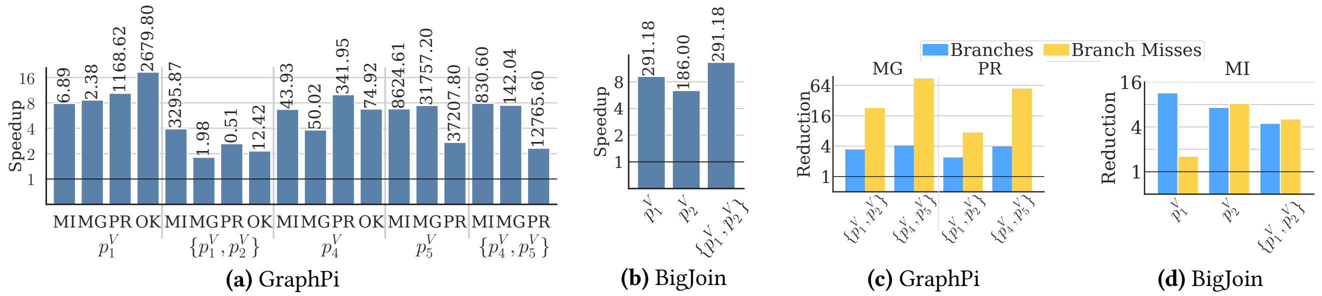


Figure 14. Performance improvements from Subgraph Morphing in GraphPi and BigJoin.

mean (vertex weights were assigned from a normal distribution). All edge-induced 4-vertex patterns (4V^E) on MiCo and Products, as well as p_4^E on MiCo were used in this experiment. Neither p_4^E nor its alternative pattern sets could be matched on Products within 24 hours.

Figure 15a and Figure 15b summarize the results. Since the filter is only dependent on the matched vertices, Subgraph Morphing trades the time spent filtering matches for additional set operations by morphing into vertex-induced patterns which have fewer matches, and then converting the matches that pass the filter on-the-fly. This reduces the time spent in UDFs by $5-16\times$, and as a result, speeds up the execution by $2.6-4\times$.

7.4 Scaling to Large Patterns

We use patterns p_9 and p_{10} that contain 7 vertices. Such large patterns are uncommon in evaluations of graph mining systems. This is because graph mining workloads scale exponentially with pattern size (theoretical bottleneck due to NP-complete nature), making large patterns difficult to mine on single machine systems even for medium-sized data graphs. Since our goal is to show the effectiveness of Subgraph Morphing on large pattern workloads, we control the data graph size to limit the workload size so that executions can finish on a single machine in reasonable time. We do so by partitioning the Products and Orkut graphs using METIS [30], and using Peregrine and GraphPi to mine p_9^V and p_{10}^V within the partitions. This way, the edges between partitions are dropped out, which reduces the workload size.

As shown in Figure 15c and Figure 15d, morphing speeds up enumeration on Peregrine by $4-7\times$, while it also improves enumeration on GraphPi by $2-5\times$. We observed that the analyses from Section 7.1 and Section 7.2 apply to large patterns as well. With Subgraph Morphing incorporated, Peregrine spent $3-11\times$ less time on set operations, while GraphPi incurred $2.2-46\times$ fewer branches.

7.5 Cost Model Effectiveness

A given input pattern can have exponentially many alternative sets, leading to potentially large gaps in performance. We study the effectiveness of our cost model in identifying the right alternative pattern set that delivers performance. Figure 15e shows the performance of 250 alternative pattern sets for 5-motif counting on MiCo, including the query pattern set and the set chosen by the cost model. The optimal set is over 3× faster than the slowest. The cost model chose an alternative pattern set which performs within 10% of the optimal one.

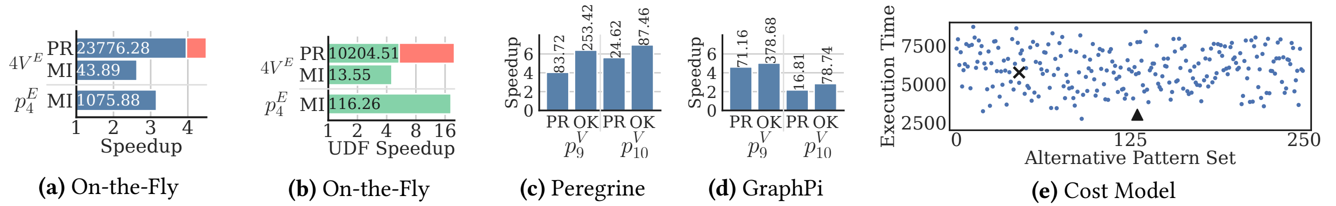
Several alternative sets perform worse than the query set; while this is visible in Figure 15e for 5-motif counting, it is especially clear in FSM which involves many patterns. For 3-FSM on PR with support threshold 140K, blindly morphing all input patterns leads to an execution time of over 22 hours, whereas the query pattern set takes 14 hours and the cost model selects a set taking only 5.65 hours.

8 Related Work

To the best of our knowledge, this paper provides the first treatment of exploiting structure-based query transformations to address bottlenecks in graph mining systems.

General-Purpose Graph Mining Systems. General-purpose graph mining systems [7, 10, 12, 19, 26, 40, 60, 66, 73] incorporate efficient subgraph exploration strategies as well as expressive programming models that enable users to express a wide range of graph mining applications. Arabesque [60], Fractal [12], RStream [66], Kaleido [73] and Pangolin [10] are exploration-based systems which mine subgraphs through iterative extensions by edges or vertices. ASAP [25] is an approximate system allowing users to navigate the tradeoff between error and performance. Tesseract [7] is a mining system for dynamic graphs.

Peregrine [26] introduced the concept of pattern awareness in graph mining systems where it exploits the structural (and label) properties of input patterns. Peregrine incorporates a pattern-based programming model that enables easier expression of complex graph mining use cases, and employs efficient pattern matching strategies [28] to deliver high performance. Being a practical end-to-end system, Peregrine also includes techniques for dynamic load balancing, early termination and on-the-fly aggregation that enable it to retain high efficiency across various graph mining applications. SumPA [19] enhances batching in pattern-aware matching plans by combining the input patterns into abstract patterns in order to eliminate redundancies during exploration. AutoMine [40] compiles input patterns into exploration programs consisting of set operation schedules. While AutoMine batches the schedules of multiple input patterns, the schedules remain oblivious to the pattern substructures and symmetries, and hence end up exploring redundant matches [26].



(a) On-the-Fly **(b)** On-the-Fly **(c)** Peregrine **(d)** GraphPi **(e)** Cost Model

Figure 15. (a-b): performance improvements from Subgraph Morphing in Peregrine for Subgraph Enumeration with On-the-Fly conversion (absolute times in seconds). (c-d): performance improvements from Subgraph Morphing for large patterns. (e): the space of alternative pattern sets for 5-motifs and their performance (in seconds) using Peregrine on MiCo graph. The input pattern set is marked by the cross and the set selected by the cost model is marked by the triangle.

All these works focus on efficiently processing graph mining applications as expressed by the user. Subgraph Morphing can be integrated as an add-on in these systems to exploit performance opportunities across different pattern applications.

Counting Subgraphs. A myriad of research has been conducted on algorithms for counting motifs [3, 18, 22, 35, 41–43, 47, 48, 72]. [3] uses combinatorial identities for counting size 3 and 4 motifs. RAGE [35] provides a method for computing edge-induced size-4 motifs, and for converting the results to those for vertex-induced motifs. [22] uses automorphism groups of pattern vertices to compute counts for motifs with 2-5 vertices, while [42, 43, 72] optimize orbit-local counting using equations for arbitrary pattern sizes. [47] computes counts for all size 5 motifs using global and local counts for smaller patterns.

As discussed in Section 4.4, none of these works are applicable for general-purpose graph mining systems since they focus (a) only on converting counts, (b) only for certain specific patterns, and (c) only on certain specific way to convert counts. Hence for instance, their combinatorial strategies (*e.g.*, scalar möbius function in [72]) cannot be generalized to arbitrary aggregations, and they cannot generate multiple alternatives, which is crucial. In comparison, Subgraph Morphing is general and captures system-level nuances and application-level characteristics, making it practical for graph mining systems.

Subgraph Matching. Several works devise efficient subgraph isomorphism solutions using sophisticated analysis of data and query graphs [4–6, 20, 21, 31, 32, 38, 39, 49, 50, 56, 57, 71]. None of these works infer matches from one pattern to another. We refer readers to a recent study [59], which evaluates state-of-the-art subgraph matching techniques. GraphZero [39] enhances the schedules from AutoMine using the standard symmetry breaking technique [18], but the individual schedules remain independent unlike in AutoMine (i.e., no schedule batching). [38] leverages an intermediate computation tree structure to generate efficient code for distributed pattern matching. GraphPi [57] uses a performance model to select efficient matching orders for subgraph matching. Finally, works like [15, 27, 51, 62] develop graph query language abstractions for easier expression of graph workloads while enabling automatic reasoning and optimization in the underlying graph runtime systems.

Frequent Subgraphs. Works like [1, 2, 13] develop solutions for mining frequent patterns, however none of these are pattern-based and they instead view the FSM computation in terms of arbitrary subgraphs of the data graph.

Graph Processing. [16, 17, 23, 34, 36, 37, 46, 53, 54, 58, 63–65, 69, 74] and others primarily focus on graph processing problems that compute values on vertices and edges, as opposed to graph mining problems that are concerned with subgraph structures. aDFS [61] enhances the graph processing system PGX.D [23] with a hybrid depth-first/breadth-first graph exploration strategy for pattern matching queries. On the other hand, techniques like [33] develop custom transformations for specific subgraphs in the data graph in order to speed up value propagation.

9 Conclusion

We presented Subgraph Morphing, a general technique to accelerate graph mining workloads across various graph mining systems. We exposed key factors that impact the performance of graph mining workloads, and observed there is no singular bottleneck that is common across the different workloads running on different graph mining systems. Subgraph Morphing exploits performance differences across pattern structures while also incorporating key system-level and application-level characteristics to deliver high performance. We formalized Subgraph Morphing and developed efficient strategies to enable it in practice. Our extensive evaluation showed promising results.

Acknowledgments

We would like to thank our shepherd Haifeng Yu and the anonymous reviewers for their valuable and thorough feedback. This work is supported by the Natural Sciences and Engineering Research Council of Canada, as well as US NSF grants CNS-1703598, CNS-1763172, CNS-1907352, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CNS-2147909, and CNS-2153449, ONR grant N00014-18-1-2037, and research grants from Cisco.

References

[1] 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 1–12, 2016.

[2] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. Incremental Frequent Subgraph Mining on Large Evolving Graphs. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1767–1768, 2018.

[3] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient Graphlet Counting for Large Networks. In *2015 IEEE International Conference on Data Mining*, ICDM 2015, pages 1–10, 2015.

[4] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, February 2018.

[5] Bibek Bhattarai, 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, pages 1447–1462, 2019.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1199–1214, 2016.

[7] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 458–473, 2021.

[8] Björn 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 of *Lecture Notes in Computer Science*, pages 858–863, 2008.

[9] 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 Thirteenth EuroSys Conference*, EuroSys '18, pages 1–12, 2018.

[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proceedings of the VLDB Endowment*, 13(10):1190–1205, April 2020.

[11] Wei-Ta Chu and Ming-Hung Tsai. Visual Pattern Discovery for Architecture Image Classification and Product Image Search. In *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*, ICMR '12, pages 1–8, 2012.

[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 2019 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. *Proceedings of the VLDB Endowment*, 7(7):517–528, March 2014.

[14] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, Qinghua Zheng, and Ting Liu. Frequent Subgraph Based Familial Classification of Android Malware. In *27th IEEE International Symposium on Software Reliability Engineering*, ISSRE 2016, pages 24–35, 2016.

[15] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, 2018.

[16] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 17–30, 2012.

[17] 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 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 599–613, 2014.

[18] 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.

[19] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. SumPA: Efficient Pattern-Centric Graph Mining with Pattern Abstraction. In *30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, pages 318–330, 2021.

[20] 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 2019 International Conference on Management of Data*, SIGMOD '19, pages 1429–1446, 2019.

[21] 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 2013 International Conference on Management of Data*, SIGMOD '13, pages 337–348, 2013.

[22] Tomaz Hocevar and Janez Demsar. A Combinatorial Approach to Graphlet Counting. *Bioinformatics*, 30(4):559–565, 2014.

[23] 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.

[24] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR*, abs/2005.00687, 2020.

[25] 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 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 745–761, 2018.

[26] 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, pages 1–16, 2020.

[27] Kasra Jamshidi, Mugilan Mariappan, and Keval Vora. Anti-Vertex for Neighborhood Constraints in Subgraph Queries. In *Proceedings of the ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '22, pages 1–9, 2022.

[28] Kasra Jamshidi and Keval Vora. A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE. *SIGOPS Operating Systems Review*, 55(1):1–10, June 2021.

[29] Tommi Junttila and Petteri Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 135–149, 2007.

[30] George Karypis and Vipin Kumar. Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, pages 35–es, 1996.

[31] 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 2016 International Conference on Management of Data*, SIGMOD '16, pages 1231–1245, 2016.

[32] 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 2018 International Conference on Management of Data*, SIGMOD '18, pages 411–426, 2018.

[33] 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, pages 245–257, 2016.

[34] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[35] Dror Marcus and Yuval Shavitt. RAGE - A Rapid Graphlet Enumerator for Large Networks. *Computer Networks*, 56(2):810–819, February 2012.

[36] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 83–98, 2021.

[37] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–16, 2019.

[38] Daniel Mawhirter, Sam Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, pages 289–303, 2021.

[39] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Operating Systems Review*, 55(1):21–37, June 2021.

[40] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 509–523, 2019.

[41] Daniel Mawhirter, Bo Wu, Dinesh Mehta, and Chao Ai. ApproxG: Fast Approximate Parallel Graphlet Counting through Accuracy Control. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 533–542, 2018.

[42] Ine Melckenbeeck, Pieter Audenaert, Didier Colle, and Mario Pickavet. Efficiently Counting All Orbits of Graphlets of Any Order in a Graph Using Autogenerated Equations. *Bioinformatics*, 34(8):1372–1380, November 2017.

[43] Ine Melckenbeeck, Pieter Audenaert, Thomas Van Parys, Yves Van De Peer, Didier Colle, and Mario Pickavet. Optimising Orbit Counting of Arbitrary Order by Equation Selection. *BMC Bioinformatics*, 20(1), January 2019.

[44] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.

[45] 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.

[46] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[47] Ali Pinar, Comandur Seshadhri, and Vaidyanathan Vishal. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 1431–1440, 2017.

[48] Mahmudur Rahman, Mansurul Alam Bhuiyan, and Mohammad Al Hasan. Graft: An Efficient Graphlet Counting Method for Large Graph Analysis. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2466–2478, October 2014.

[49] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and Robust Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, 2019.

[50] 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 265–281, 2018.

[51] Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, 2015.

[52] 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, pages 983–992, 2017.

[53] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, 2015.

[54] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 1–12, 2013.

[55] Soumajyoti Sarkar, Ruocheng Guo, and Paulo Shakarian. Using Network Motifs to Characterize Temporal Network Evolution Leading to Diffusion Inhibition. *CoRR*, abs/1903.00862, 2019.

[56] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. QFrag: Distributed Graph Search via Subgraph Isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 214–228, 2017.

[57] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, pages 1–14, 2020.

[58] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[59] Shixuan Sun and Qiong Luo. In-Memory Subgraph Matching: An In-Depth Study. In *Proceedings of the 2020 International Conference on Management of Data*, SIGMOD '20, pages 1083–1098, 2020.

[60] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, 2015.

[61] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *2021 USENIX Annual Technical Conference*, ATC '21, pages 209–224, 2021.

[62] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 1–6, 2016.

[63] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference*, ATC '19, pages

429–442, July 2019.

[64] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 237–251, 2017.

[65] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 861–878, 2014.

[66] 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 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 763–782, 2018.

[67] Li Wang, Hongying Zhao, Jing Li, Yingqi Xu, Yujia Lan, Wenkang Yin, Xiaoqin 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.

[68] Xuyun Wen, Wei-Neng Chen, Ying Lin, Tianlong Gu, Huaxiang Zhang, Yun Li, Yilong Yin, and Jun Zhang. A Maximal Clique Based Multiobjective Evolutionary Algorithm for Overlapping Community Detection. *IEEE Transactions on Evolutionary Computation*, 21(3):363–377, June 2017.

[69] Chengshuo Xu, Keval Vora, and Rajiv Gupta. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 587–600, 2019.

[70] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[71] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2049–2062, 2021.

[72] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. Distributed Subgraph Counting: A General Approach. *Proceedings of the VLDB Endowment*, 13(12):2493–2507, July 2020.

[73] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *36th IEEE International Conference on Data Engineering*, ICDE '20, pages 673–684, 2020.

[74] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 301–316, 2016.

A Applications with Subgraph Morphing

We will walk through the main steps in applying Subgraph Morphing (*i.e.*, S-DAG generation, pattern selection, and result conversion) on two graph mining use cases: Frequent Subgraph Mining and Subgraph Counting.

A.1 Frequent Subgraph Mining

Since FSM explores labeled edge-induced patterns, it can end up matching and computing MNI for a large number of patterns. To simplify exposition, we consider a single pattern p_a^E (edge-induced 4-star). Figure 16 summarizes the example.

The S-DAG is constructed by recursively adding the superpatterns of p_a^E. The resulting S-DAG is shown in Figure 16a. Since we are dealing with labeled patterns, some of the superpatterns can have identical structures but different labelings. Patterns p_b and p_c in the S-DAG show this case.

Costs are estimated for both variants of each pattern in the S-DAG. The pattern costs for our example are shown in Figure 16c. Since MNI computations are sensitive to output size, patterns that are estimated to produce more matches have higher costs. For example, p_a^E is the least constrained pattern in the S-DAG, and hence has the highest cost. Similarly, the other superpatterns have lower costs for the vertex-induced variants which cause fewer matches.

Next, the alternative pattern set S is constructed using Algorithm 1. Initially, $S = \{p_a^E\}$. Then we iterate over the direct parents of p_a^E in the S-DAG, beginning with p_b^V. The only child of p_b^V is p_a^E with cost 25 while the superpatterns of p_a^E (including p_a^V) have combined cost 17. As a result, S is updated to contain $\{p_a^V, p_b^V, p_c^V, p_d^V, p_e^V, p_f\}$ and all these patterns have their costs set to 0. The algorithm converges in the next iteration as the alternative pattern set S does not change.

The matching engine explores the subgraphs that match the patterns in S. The final step is to compute the MNI table for p_a^E from the MNI results for patterns in S. To illustrate this, consider the sample data graph shown in Figure 16b. Figure 16d shows the MNI tables for the alternative pattern set S. Note that p_a^V, p_c^V and p_d^V do not have any matches in this example, and hence their MNI tables are empty (not shown). Figure 16e shows how the final MNI table is computed from the tables for alternative patterns. Starting with an empty table, the MNI tables are merged after permuting them using permutation functions. Consider the MNI table for p_e^V. There are two subgraph isomorphisms from p_a^E to p_e^V, which lead to two permutations. The first one is the identity permutation (*i.e.*, unchanged) which results in T_1. The second one sends the first column of the MNI table to the second, the second column to the third, and the third column to the first. Applying this permutation and merging the resulting table with T_1 gives T_2. This process continues with the next alternative pattern p_f and results in T_3. There are two further isomorphisms into p_f, and one into p_b^V, none of which affect the final result, and the process completes with T_6.

A.2 Subgraph Counting

In this application, we are interested in counting the subgraphs that match three unlabeled vertex-induced patterns: a 4-star, a 4-cycle, and a 4-chain. Figure 17 summarizes the example where the three patterns are named p_a, p_b and p_c.

Similar to the previous example, S-DAG is constructed by recursively adding superpatterns of those three input patterns. The resulting S-DAG is shown in Figure 17a and the estimated pattern costs are shown in Figure 17c. In this case, since the patterns are unlabeled and the counting aggregation is a constant time operation, the set operation time is the primary concern. Hence, edge-induced variants of sparse patterns tend to be far cheaper to compute than their vertex-induced variants which require additional set differences.

Using the S-DAG and the pattern costs, Algorithm 1 computes the alternative pattern set S. Initially, S starts with $\{p_a^V, p_b^V, p_c^V\}$. Then, p_a^V is evaluated against its superpatterns (p_a^E, p_c^V, p_d^E, p_e^E, p_f). Since p_a^V costs 20 while its superpatterns cost 30 combined, p_a^V is not morphed in this step. Similarly, p_b^V is not morphed in the next step. However, when $C = \{p_a^V, p_b^V\}$, the cost of C is 50 while the cost of the combined superpatterns (including the variants of the patterns in C) is only 33. Hence, S is updated to replace p_a^V and p_b^V with p_a^E, p_b^E, and the other superpatterns, and the cost of these superpatterns is set to 0. Notice that p_c^V is one of the superpatterns of p_a and p_b. Since the original cost of the superpatterns of p_c^V was greater than the cost of p_c^V, it would not have been morphed. However, since the cost of superpatterns got set to 0, the new cost of superpatterns of p_c^V reduces to 10. Hence, S is updated once again with p_e^E instead of p_e^V. The final alternative pattern set S is $\{p_a^E, p_b^E, p_c^E, p_d^E, p_e^E, p_f\}$.

After matching the alternative patterns, their results are transformed back to counts for p_a, p_b and p_c. We discuss this result conversion process next. Figure 17b shows an example data graph, and Figure 17d shows the number of matches in the data graph for the alternative patterns. The permutation function accounts for the subgraph isomorphisms from the original patterns to the alternative patterns. For example, consider pattern p_c^V whose counts can be computed using [SM-V1] in Figure 7, i.e., $|M(p_c^V)| = |M(p_c^E)| - |M(p_e^V)| - 3 \times |M(p_f)|$. However, our alternative set contains the morphed patterns for p_e^V, and hence, $|M(p_e^V)|$ is computed as $|M(p_e^E)| - 6 \times |M(p_f)|$. Therefore, $|M(p_c^V)| = 7 - 3 - 3 \times 1 = 1$. Counts for p_a^V and p_b^V are computed in a similar manner.

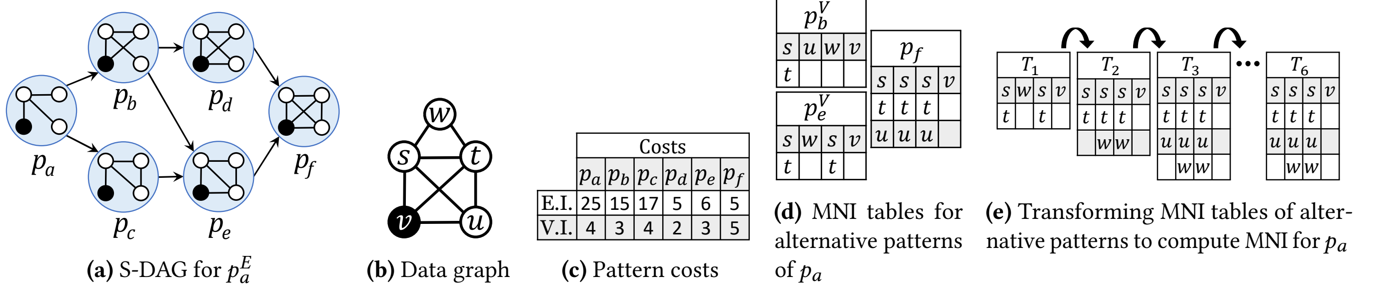


Figure 16. Frequent Subgraph Mining (FSM) with Subgraph Reshaping. Key steps in reshaping are shown for pattern p_a.

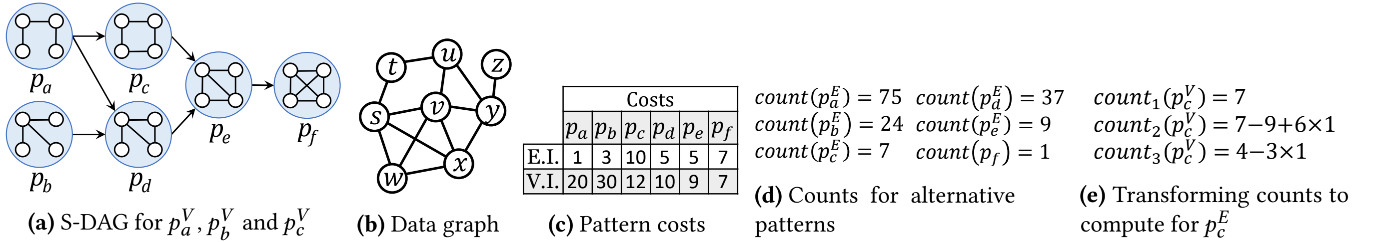


Figure 17. Subgraph Counting (SC) with Subgraph Reshaping. Key steps in reshaping shown for input patterns p_a, p_b and p_c.

B Artifact Appendix

B.1 Abstract

This artifact implements the core Subgraph Morphing techniques, including the SDAG data structure, all algorithms presented in the paper, as well as the experiments supporting our major claims. We have provided scripts to re-create our major benchmarking experiments on the open-source systems evaluated in Section 7.

Note that this artifact does not contain our AutoZero implementation (original code for those systems is not made available by the authors), and hence we have not provided scripts to reproduce the results for AutoZero experiments (*i.e.*, Figure 12b and Figure 12d).

B.2 Description & Requirements

B.2.1 How to access. Source code, datasets, instructions for building the software, and scripts to run the experiments are available in our git repository: `<link-removed>`.

B.2.2 Hardware dependencies. We conducted our experiments in a Google Cloud `n2-highcpu-32` instance, comprising 32 virtual cores and 32 GB memory. Graph mining is CPU intensive and embarrassingly parallel, so we recommend a powerful multi-core machine to run experiments. Additionally, some experiments use the Friendster dataset, which requires approximately 36GB memory to pre-process, and the same amount of disk space, though after pre-processing its space requirement falls to 28GB. The remaining datasets take up around 5GB disk space.

B.2.3 Software dependencies. To re-create our testbed, the following software is required:

- Linux operating system; we conducted our experiments on Ubuntu 22.04
- `g++` (version 11 or newer) and `cmake` (version 3.13 or newer) to build Peregrine
- `OpenMPI` (`libopenmpi-dev`) to build GraphPi
- `cargo` to build BigJoin
- Additionally, the experiment scripts use `datamash` and `bc` to analyze execution results

B.2.4 Benchmarks. Our git repository contains all necessary benchmarking code.

B.3 Set-up

First, clone our git repository on a Ubuntu 22.04 machine. Then for each package mentioned above, run

```
$ apt install -y <package name>
```

From within the repository, you can build the various systems. To build Peregrine (10 minutes), run

```
$ make bliss; make -j
```

To build GraphPi (5 minutes), run

```
$ ./scripts/install-gpi.sh
```

To build BigJoin (15 minutes or more, depending on network speed), run

```
$ ./scripts/install-bigjoin.sh
```

To check if everything is functioning (30 seconds), run

```
$ ./scripts/sanity_check.sh
```

B.4 Evaluation workflow

B.4.1 Major Claims.

- *(C1): Subgraph Morphing algebra (introduced in Section 4) improves performance across a wide range of graph mining applications and inputs without sacrificing correctness. The performance claim is verified by the experiments: (E1-2) described in Section 7.1 and presented in Figure 12a and Figure 13a; (E3-5) described in Section 7.2 and presented in Figure 13c, Figure 14a, and Figure 14b; and (E6) described in Section 7.3 and presented in Figure 15a. Correctness is validated in all these experiments, as the outputs are equal between baseline and morphed executions.*
- *(C2): The Subgraph Morphing module is able to navigate tradeoffs in different pattern sets to select efficient alternative patterns in negligible time compared to the graph mining execution. This is demonstrated in experiment (E7) described in Section 7.5 and presented in Figure 15e.*
- *(C3): Subgraph Morphing can scale to large pattern sizes, as evidenced by experiments (E8-9) described in Section 7.4 and presented in Figure 15c and Figure 15d.*
- *(C4): Subgraph Morphing is system-agnostic and can be integrated in several different state-of-the-art pattern-based graph mining systems to improve performance. This is demonstrated through all experiments (E1-9), as they demonstrate performance improvement in three different graph mining systems.*

B.4.2 Experiments. All experiments reproduce figures in Section 7 in tabular form, using a comma-separated value (CSV) format. Each experiment comes with a single script that reproduces its corresponding figure, with the convention that `./scripts/figXX.sh` will reproduce Figure XX. We provide additional scripts that run on limited versions of experiments which take unreasonably long to run exhaustively. They take the form `./scripts/figXX-quick.sh`.

The results of experiments (E1-2), (E4-6), and (E8-9) are identical in format and interpretation. A CSV file will be written to the terminal as execution progresses, always showing the command currently being run. At the end of execution, the CSV file contains a line for each set of inputs in the format:

```
p,g,morphed time,baseline time,speedup
```

where `p` is the pattern name, `g` is the graph name, and the execution times are in seconds.

(E1) [Motif Counting] [3 compute-days] [limited: 12 compute-hours]: This experiment reproduces Figure 12a and validates claim (C1) and (C4).

(E2) [Subgraph Counting] [5 compute-days] [limited: 1 compute-day]: This experiment reproduces Figure 13a and validates claim (C1).

(E3) [Frequent Subgraph Mining] [1.5 compute-days] [limited: 6 compute-hours]: This experiment reproduces Figure 13c and validates claim (C1).
[Results] The output of this experiment is also a CSV file, with the following format:

```
t,g,morphed time,baseline time,speedup
```

where `t` is the support threshold, `g` is the graph name, and the execution times are in seconds.

(E4) [GraphPi Filter] [1 compute-day] [limited: 3 compute-hours]: This experiment reproduces Figure 14a and validates claims (C1) and (C4).

(E5) [BigJoin Filter] [6 compute-hours]: This experiment reproduces Figure 14b and validates claims (C1) and (C4).

(E6) [On-The-Fly Conversion] [1.5 compute-days] [limited: 1.5 compute hours]: This experiment reproduces Figure 15a and validates claim (C1).

(E7) [Cost Modeling] [10 milliseconds]: This experiment reproduces Figure 15e and validates claim (C2). The provided script uses the Subgraph Reshaper to select an alternative pattern set for the 5-motifs pattern set, then compares its performance to that of every other possible choice of alternative pattern set.
[Preparation] This experiment requires a file containing the execution time for every size 5 pattern both edge-induced and vertex-induced. One is provided in the repository, but users can generate it themselves by simply running Peregrine on every size 5 pattern individually and recording the times.
[Results] The output is the total runtime of the alternative pattern set selected by the Subgraph Reshaper, as well as statistics about this selection relative to other possible alternative pattern sets.

(E8) [Peregrine Pattern Scalability] [1 compute-hour]: This experiment reproduces Figure 15c and validates claim (C3).

(E9) [GraphPi Pattern Scalability] [1 compute-hour]: This experiment reproduces Figure 15d and validates claim (C3).