PEREGRINE: A Pattern-Aware Graph Mining System

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

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

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

Abstract

Graph mining workloads aim to extract structural properties of a graph by exploring its subgraph structures. General purpose graph mining systems provide a generic runtime to explore subgraph structures of interest with the help of user-defined functions that guide the overall exploration process. However, the state-of-the-art graph mining systems remain largely oblivious to the shape (or pattern) of the subgraphs that they mine. This causes them to: (a) explore unnecessary subgraphs; (b) perform expensive computations on the explored subgraphs; and, (c) hold intermediate partial subgraphs in memory; all of which affect their overall performance. Furthermore, their programming models are often tied to their underlying exploration strategies, which makes it difficult for domain users to express complex mining tasks.

In this paper, we develop PEREGRINE, a pattern-aware graph mining system that directly explores the subgraphs of interest while avoiding exploration of unnecessary subgraphs, and simultaneously bypassing expensive computations throughout the mining process. We design a pattern-based programming model that treats *graph patterns* as first class constructs and enables PEREGRINE to extract the semantics of patterns, which it uses to guide its exploration. Our evaluation shows that PEREGRINE outperforms state-of-the-art distributed and single machine graph mining systems, and scales to complex mining tasks on larger graphs, while retaining simplicity and expressivity with its 'pattern-first' programming approach.

1 Introduction

Graph mining based analytics has become popular across various important domains including bioinformatics, computer vision, and social network analysis [3, 9, 28, 36, 41, 59].

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387548

	Arabesque	Fractal	G-Miner	RStream	PRG-U
PEREGRINE	2-1317×	1.1-737×	3-131×	2-2016×	2-42×

Table 1. PEREGRINE performance summary. PRG-U indicates PEREGRINE without symmetry breaking, to model systems that are not fully pattern-aware (e.g., AutoMine).

These tasks mainly involve computing structural properties of the graph, i.e., exploring and understanding the substructures within the graph. Since the search space is exponential, graph mining problems are computationally intensive and their solutions are often difficult to program in a parallel or distributed setting.

To address these challenges, general-purpose graph mining systems like Arabesque [52], RStream [57], Fractal [12], G-Miner [8] and AutoMine [34] provide a generalized exploration framework and allow user programs to guide the overall exploration process. At the heart of these graph mining systems is an exploration engine that exhaustively searches subgraphs of the graph, and a series of filters that prune the search space to continue exploration for only those subgraphs that are of interest (e.g., ones that match a specific pattern) and are unique (to avoid redundancies coming from structural symmetries). The exploration happens in a step-by-step fashion where small subgraphs are iteratively extended based on their connections in the graph. As these subgraphs are explored, they get verified via *canonicality checks* to guarantee uniqueness, and get analyzed via *isomorphism computations* to understand their structure (or pattern). After that, the subgraphs either get pruned out because they don't match the pattern of interest, or are forwarded down the pipeline where their information is aggregated at the pattern level.

While such an exploration process is general enough to compute different mining use cases including Frequent Subgraph Mining and Motif Counting, we observe that it remains largely oblivious to the patterns that are being mined. Hence, state-of-the-art graph mining systems face three main issues, as described next: (1) These systems perform a large number of unnecessary computations; specifically, every subgraph explored from the graph, even in intermediate steps, is processed to ensure canonicality, and is analyzed to either extract its pattern or to verify whether it is isomorphic to another pattern. Since the exploration space for graph mining use cases is very large, performing those computations on every explored subgraph severely limits the performance of these

systems. (2) The exhaustive exploration in these systems ends up generating a large amount of intermediate subgraphs that need to be held (either in memory or on disk) so that they can be extended. While systems based on breadth-first exploration [52, 57] demand high memory capacity, other systems like Fractal [12] and AutoMine [34] use guided exploration strategies to reduce this impact; however, because they are not fully pattern-aware, they process a large number of intermediate subgraphs which severely limits their scalability as graphs grow large. (3) The programming model in these systems is strongly tied to the underlying exploration strategy, which makes it difficult for domain experts to express complex mining use cases. For example, subgraphs containing certain pairs of strictly disconnected vertices (i.e., absence of edges) are useful for providing recommendations based on missing edges; mining such subgraphs with constraints on their substructure cannot be directly expressed in any of the existing systems.

In this paper, we take a 'pattern-first' approach towards building an efficient graph mining system. We develop PEREGRINE [1], a pattern-aware graph mining system that directly explores the subgraphs of interest while avoiding exploration of unnecessary subgraphs, and simultaneously bypassing expensive computations (isomorphism and canonicality checks) throughout the mining process. PEREGRINE incorporates a pattern-based programming model that enables easier expression of complex graph mining use cases, and reveals patterns of interest to the underlying system. Using the pattern information, PEREGRINE efficiently mines relevant subgraphs by performing two key steps. First, it analyzes the patterns to be mined in order to understand their substructures and to generate an exploration plan describing how to efficiently find those patterns. And then, it explores the data graph using the exploration plan to guide its search and extract the subgraphs back to the user space.

Our pattern-based programming model treats *graph patterns* as first class constructs: it provides basic mechanisms to load, generate and modify patterns along with interfaces to query patterns in the data graph. Furthermore, we introduce two novel abstractions, an ANTI-EDGE and an ANTI-VERTEX, that express advanced structural constraints on patterns to be matched. This allows users to directly operate on patterns and express their analysis as 'pattern programs' on PEREGRINE. Moreover, it enables PEREGRINE to extract the semantics of patterns which it uses to generate efficient exploration plans for its pattern-aware processing model.

We rely on theoretical foundations from existing subgraph matching research [5, 16] to generate our exploration plans. Since PEREGRINE directly finds the subgraphs of interest, it does not incur additional processing over those subgraphs throughout its exploration process; this directly results in much lesser computation compared to the state-of-the-art graph mining systems. Moreover, PEREGRINE does not maintain intermediate partial subgraphs in memory, resulting in much lesser memory consumption compared to other systems.

PEREGRINE runs on a single machine and is highly concurrent. We demonstrate the efficacy of PEREGRINE by evaluating it on several graph mining use cases including frequent subgraph mining, motif counting, clique finding, pattern matching (with and without structural constraints), and existence queries. Our evaluation on real-world graphs shows that PEREGRINE running on a single 16-core machine outperforms state-of-the-art distributed graph mining systems including Arabesque [52], Fractal [12] and G-Miner [8] running on a cluster with eight 16-core machines; and significantly outperforms RStream [57] running on the same machine. Furthermore, PEREGRINE could easily scale to large graphs and complex mining tasks which could not be handled by other systems. Table 1 summarizes PEREGRINE's performance.

2 Background & Motivation

We first briefly review graph mining fundamentals, and then discuss performance and programmability issues in state-of-the-art graph mining systems. In the end, we give an overview of PEREGRINE's pattern-aware mining techniques.

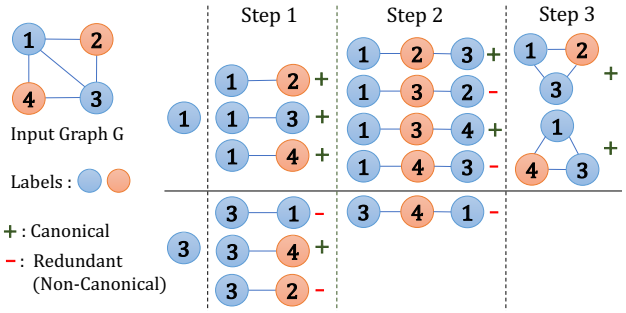
2.1 Graph Mining Overview

Graph Terminology. Given a graph g, we use $V(g)$ and $E(g)$ to denote its set of vertices and edges respectively. If the graph is labeled, we use $L(g)$ to denote its set of labels. A *subgraph* s of g is a graph containing a subset of edges in g and their endpoints.

Graph Mining Model. Graph mining problems involve finding subgraphs of interest in a given input graph. We use P to denote the *pattern graph* (representing structure of interest) and G to denote the input *data graph*. We define a *match* M as a subgraph of G that is *isomorphic* to P, where isomorphism is defined as a one-to-one mapping between $V(P)$ and $V(M)$ such that if two vertices are adjacent in P, then their corresponding vertices are adjacent in M. There are two kinds of matches depending on how vertices and edges from G are extracted in M. An *edge-induced match* is any subgraph of G that is isomorphic to P. A *vertex-induced match* is a subgraph of G that is isomorphic to P while containing all the edges in $E(G)$ that are incident on $V(M)$.

Since there can be sub-structure symmetries within P (e.g., a triangle structure looks the same when it is rotated), the same subgraph of G can result in multiple different matches each with a different one-to-one mapping with $V(P)$. These matches are *automorphisms* of each other, where automorphic matches are defined as two matches M_1 and M_2 such that $V(M_1) = V(M_2)$. A *canonical match* is the unique representative of a set of automorphic matches. Hence, uniqueness is ensured by choosing the canonical match from every set of automorphic matches in G.

[1] PEREGRINE source code: https://github.com/pdclab/peregrine



(a) Step-by-step exploration in graph mining systems starting at vertex 1 and vertex 3. In total, 13 partial matches get explored and 13 canonicality checks are performed that prune out 5 partial matches. Isomorphism checks are performed on the remaining 8 matches for applications like FSM.

System	Total Matches	Canonicality Computations	Isomorphism Computations
RStream	1.2B (342×)	33.0M	0
Arabesque	1.4B (400×)	1.4B	3.5M
Fractal	659.0M (188×)	599.6M	0

(b) Profiling results for 4-Clique Counting on Patents [17] which contains ~3.5M cliques of size 4. Isomorphism counts are 0 for RStream and Fractal because they have native support for clique computation.

System	Total Matches	Canonicality Computations	Isomorphism Computations
RStream	40.1B (125×)	40.1B	343.3M
Arabesque	685.8M (2.1×)	685.8M	320.7M
Fractal	665.6M (2.1×)	649.1M	320.7M

(c) Profiling results for 3-Motif Counting on Patents [17] which contains ~320M 3-sized motifs.

Figure 1. Left: Example illustrating step-by-step exploration; Right: Number of matches explored (partial and full), canonicality checks performed, and isomorphism checks performed by RStream [57], Arabesque [52] and Fractal [12]. Numbers in brackets indicate the magnitude of matches explored relative to result size.

While the techniques presented in this paper work for both directed and undirected graphs, for easier exposition, we assume that P and G are undirected.

Graph Mining Problems. We briefly describe common graph mining problems. While the problems listed below focus on *counting* subgraphs of interest, they are often generalized to *listing* (or *enumerating*) as well.

— *Motif Counting.* A motif is any connected, unlabeled graph pattern. The problem involves counting the occurrences of all motifs in G up to a certain size.

— *Frequent Subgraph Mining (FSM).* The problem involves listing all labeled patterns with k edges that are frequent in G (i.e., frequency of their matches in G exceed a threshold τ). The frequency of a pattern (also called support) is measured in a variety of ways [30, 38, 39, 58], but most systems choose the *minimum node image (MNI)* [6] support measure since it can be computed efficiently. MNI is anti-monotonic, i.e., given two patterns p and p' such that p is a subgraph of p', support of p will be at least as high as that of p'.

— *Clique Counting.* A k-clique is a fully-connected graph with k vertices. The problem involves counting the number of k-cliques in G. Variations of this problem include counting *pseudo-cliques*, i.e., patterns whose edges exceed some density threshold; *maximal cliques*, i.e., cliques that are not contained in any other clique; and, *frequent cliques*, i.e., cliques that are frequent (exceeding a frequency threshold).

— *Pattern Matching.* The problem involves matching (counting) the number of subgraphs in G that are isomorphic to a given pattern. A variation of this problem is counting *constrained subgraphs*, i.e., subgraphs with structural constraints (e.g., certain vertices in the subgraph must not be adjacent).

All of the above graph mining use cases can be modelled in 3 steps: pattern selection, pattern matching, and aggregation.

2.2 Issues with Graph Mining Systems

While several graph mining systems have been developed [8, 12, 34, 52, 57], they are not pattern-aware. Hence, they demand high computation power and require large memory (or storage) capacity, while also lacking the ability to easily express mining programs at a high level.

2.2.1 Performance.

(A) High Computation Demand. Graph mining systems explore subgraphs in a step-by-step fashion by starting with an edge and iteratively extending matches depending on the structure of the data graph. Since they do not analyze the structure of the pattern to guide their exploration, they perform a large number of: (a) unnecessary explorations; (b) canonicality checks; and, (c) isomorphism checks.

Figure 1a shows an example of step-by-step exploration starting from vertex 1 and vertex 3. In step 1, both the vertices get extended generating 6 partial matches each of size 1 (edges). These are tested for canonicality which prunes out $(3, 1)$ and $(3, 2)$ (non-canonical matches are marked with $-$). For applications like FSM, isomorphism checks are performed on each of the canonical matches to identify their structure and compute metrics. Then, the remaining 4 matches progress to the next step and the entire process repeats. While explorations get pruned via both canonicality and isomorphism checks, every valid partial match is extended to multiple matches which may no longer be valid; generation of intermediate matches which do not result into valid final matches is unnecessary. Furthermore, all intermediate partial matches (unnecessary and valid matches) are operated upon to identify their structure (i.e., isomorphism check) and to verify their uniqueness (i.e., canonicality check). In our example, 13 intermediate matches get generated, 5 of which are unnecessary; 13 canonicality checks and 8 isomorphism

checks are performed. If these checks are not performed at every step (as done in Fractal [12] by delaying its `filter` step), a massive amount of partial and complete matches that do not contribute to final result would get generated.

We verified the above behavior by profiling graph mining systems on clique counting and motif counting applications. As shown in Figure 1b and Figure 1c, on Patents [17] (a real-world graph dataset), RStream [57] and Arabesque [52] generate over a billion partial matches for clique counting while the total number of cliques is only ~3.5M (~99.7% matches were unnecessary); similarly for motif counting, RStream generates over 40 billion partial matches (~99.2% unnecessary) and Arabesque generates over 685 million partial matches (~52% unnecessary). They also perform a large number (hundreds of millions to billions) of canonicality checks and isomorphism checks. Since Fractal [12] explores in depth-first fashion, its numbers are better than RStream and Arabesque; however, they are still very high.

(B) High Memory Demand. Graph mining systems often hold massive amounts of (partial and complete) matches in memory and/or in external storage. Systems based on step-by-step exploration require valid partial matches so that they can be extended in subsequent steps; the total size (in bytes) required by all matches (partial and complete) quickly grows (often beyond main memory capacity) as the size of the pattern or data graph increases. Such a memory demand is lower in DFS-based exploration (as done in Fractal [12]). For clique and motif counting in Figure 1, Arabesque consumes ~101GB main memory while Fractal requires ~32GB memory.

2.2.2 Programmability. Programming in graph mining systems is done at vertex and edge level, with semantics of constructing the required matches defined explicitly by user's mining program. This means, mining programs expressed in those systems contain the logic for: (a) validating partial and complete matches; (b) extending matches via edges and/or vertices; and, (c) processing the final valid matches. As the size of subgraph structure to be mined grows, the complexity of validating partial matches increases, making mining programs difficult to write. For example, the multiplicity algorithm to avoid over-counting in AutoMine [34] cannot be used if the user wants to enumerate patterns, which leaves the responsibility of identifying unique matches to the user. Furthermore, complicated structural constraints beyond the presence of vertices, edges and labels cannot be easily expressed in any of the existing systems.

2.3 Overview of PEREGRINE

We develop a pattern-aware graph mining system that directly finds subgraphs of interest without exploring unnecessary matches while simultaneously avoiding expensive isomorphism and canonicality checks throughout the mining process. We do so by designing a pattern-based programming model that treats *graph patterns* as first class constructs, and by developing a processing model that uses the pattern's substructure to guide the exploration process.

Pattern-based Programming. In PEREGRINE, graph mining tasks are directly expressed in terms of subgraph structures (i.e., graph patterns). Our pattern-aware programming model allows declaring (statically and dynamically generated) patterns, modifying patterns, and performing user-defined operations over matches explored by the runtime. This allows concisely expressing mining programs by abstracting out the underlying runtime details, and focusing only on the substructures to be explored. Moreover, we introduce two novel abstractions, *anti-edges* and *anti-vertices*: an anti-edge enforces strict disconnection between two vertices in the match whereas an anti-vertex captures strict absence of a common neighbor among vertices in the match. These abstractions allow users to easily express advanced structural constraints on patterns to be mined.

Automatic Generation of Exploration Plan. With patterns of interest directly expressed, PEREGRINE analyzes the patterns and computes an *exploration plan* which is later used to guide the exploration in the data graph. Specifically, the pattern is first analyzed to eliminate symmetries within itself so that expensive canonicality checks during exploration can be avoided. Then the pattern is reduced to its *core substructure* that enables identifying matches using simple graph traversals and adjacency list intersection operations without performing explicit isomorphism checks.

Guided Pattern Exploration. After the exploration plan is generated, PEREGRINE starts the exploration process using our pattern-aware processing model. The exploration process matches the core substructure of the pattern to generate partial matches using recursive graph traversals in the data graph. As partial matches get generated, they are extended to form final complete matches by intersecting the adjacency lists of vertices in the partial matches. Since the entire exploration is guided by the plan generated from the pattern of interest, the exploration does not require intermediate isomorphism and canonicality checks for any of the partial and complete matches that it generates. This reduces the amount of computation done in PEREGRINE compared to state-of-the-art graph mining systems. Moreover, since matches are recursively explored and instantly extended to generate complete final results, partial state is not maintained in memory throughout the exploration process which significantly reduces the memory requirement for PEREGRINE.

Finally, we reduce load imbalance in PEREGRINE by enforcing a strict matching order based on vertex degrees. Furthermore, we incorporate on-the-fly aggregation and early termination features to provide global updates as mining progresses so that exploration can be stopped once the conditions required to compute final results are met.

```
[L1] Set<Pattern> loadPatterns(String filename);
[G1] Set<Pattern> generateAllEdgeInduced(Int size);
[G2] Set<Pattern> generateAllVertexInduced(Int size);
[S1] Pattern generateClique(Int size);
[S2] Pattern generateStar(Int size);
[S3] Pattern generateChain(Int size);
[C1] Set<Pattern> extendByEdge(Set<Pattern> patterns);
[C2] Set<Pattern> extendByVertex(Set<Pattern> patterns);
```

```
class Pattern {
  Set<Vertex> getNeighbors(Vertex u);
  Label getLabel(Vertex u);
  Bool areConnected(Vertex src, Vertex dst);
  Void addEdge(Vertex src, Vertex dst);
  Void addAntiEdge(Vertex src, Vertex dst);
  Void removeEdge(Vertex src, Vertex dst);
  Void addLabel(Vertex u, Label l);
  . . .
};
```

Figure 2. PEREGRINE Pattern Interface.

3 PEREGRINE Programming Model

Since graph mining fundamentally involves finding subgraphs that satisfy certain structural properties, we design our programming model around *graph patterns* as first class constructs. This allows users to easily express the subgraph structures of interest, without worrying about the underlying mechanisms of how to explore the graph and find those structures. With such a declarative style of expressing patterns, PEREGRINE enables users to program complex mining queries as operations over the matches. The clear separation of *what to find* and *what to do with the results* helps users to quickly reason about correctness of their mining logic, and develop advanced mining-based analytics.

We first present how patterns are directly expressed in PEREGRINE, and then show how common graph mining use cases can be programmed with patterns in PEREGRINE.

3.1 PEREGRINE Patterns

Figure 2 shows our API to directly express, construct and modify connected graph patterns. Patterns can be constructed statically and loaded using `[L1]`, or can be constructed dynamically `[G1-G2, C1-C2, S1-S3]`. `[G1]` and `[G2]` generate all unique patterns that can be induced by certain number of edges and vertices respectively. `[S1-S3]` generate special well-known patterns. `[C1-C2]` take a group of patterns as input, and extend one of them by an edge or a vertex, to return all of the unique new patterns that result from these extensions. This allows constructing patterns step-by-step which is useful to perform guided exploration. The `Pattern` class provides a standard interface to access and modify the pattern graph structure.

In most common applications, the edges and vertices in the pattern graph are sufficient for PEREGRINE to find subgraph structures that match the pattern. For advanced mining use cases that require structural constraints within the pattern, we introduce *anti-edges* and *anti-vertices*.

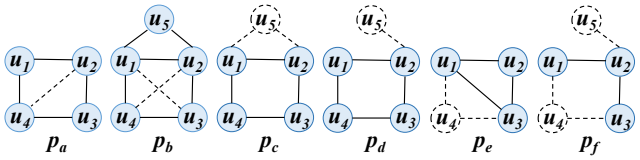


Figure 3. Example patterns with Anti-Edges and Anti-Vertices.

3.1.1 Anti-Edges. Anti-edges are used to model constraints between vertex pairs in the pattern. They are special edges indicating disconnections between pairs of vertices. For example, in a social network graph where vertices model people and edges model their friendships, extracting unrelated people with at least two mutual friends can be achieved using p_a in Figure 3 where (u_2, u_4) is an anti-edge.

An anti-edge ensures that the two vertices in the match do not have an edge between them in the data graph. If two vertices u_1 and u_2 are connected via an anti-edge in a pattern p, then any match for p guarantees the anti-edge constraint:

$$m(u_1) = v_1 \wedge m(u_2) = v_2 \implies (v_1, v_2) \notin E(G)$$

where m is the function mapping vertices in p to vertices in G. The two vertices connected by an anti-edge are called *anti-adjacent*. Figure 3 shows another example pattern (p_b) with anti-adjacent vertices u_1-u_3 and u_2-u_4, and their corresponding anti-edges (u_1, u_3) and (u_2, u_4). PEREGRINE natively supports anti-edges (discussed in §4.2), and hence, it directly matches only those subgraphs that do *not* contain an edge between the two anti-adjacent vertices.

3.1.2 Anti-Vertices. Anti-vertices are used to model constraints among shared neighborhoods of vertices in the pattern. They are special vertices that are only connected to other vertices via anti-edges. For example, extracting pairs of friends with only one mutual friend in a social network graph can be achieved using p_e in Figure 3 where u_4 is an anti-vertex. Such a query cannot be directly expressed using anti-edges alone.

Anti-vertices represent absence of a vertex. So a match of a pattern with an anti-vertex will not contain a data vertex matching the anti-vertex. If $\overline{u}$ is an anti-vertex in a pattern p and S is the set of data vertices matching the neighbors of $\overline{u}$, then any match for p guarantees the anti-vertex constraint:

$$S = m(\text{adj}(\overline{u})) \implies \bigcap_{v \in S} \text{adj}(v) \setminus m(\text{adj}(m^{-1}(v))) = \varnothing$$

where m is the function mapping vertices in p to vertices in G, and m^{-1} is its inverse.

To distinguish anti-vertices, we call a vertex with at least one regular edge (i.e., not anti-edge) a regular vertex. Figure 3 shows different patterns with anti-vertices: In a match for p_c, the matches for u_1 and u_2 have no common neighbors. On the other hand, in a match for p_d, the match for u_2 has no neighbors other than the matches for u_1 and u_3. Finally, p_f has two anti-vertices which combines constraints from p_c and p_d. PEREGRINE natively supports anti-vertices (discussed in

```
Void updateSupport(Match m) { mapPattern(m.getDomain()); }
Bool isFrequent(Pattern p, Domain d) {
  return (d[p].support() >= threshold);
}
DataGraph G = loadDataGraph("labeledInput.graph");
Set<Pattern> patterns = generateAllEdgeInduced(2);
while (patterns not empty) {
  Map<Pattern, Domain> results = match(G, patterns, updateSupport);
  Set<Pattern> frequentPatterns = results.filter(isFrequent).keys();
  patterns = extendByEdge(frequentPatterns);
}
```

(a) Frequent Subgraph Mining

```
Int numTriplets = 0;
Void countAndCheck(Match m) {
  Int numTriangles = loadAggregatedValue(m);
  if (numTriangles*3/numTriplets > bound)
    stopExploration();
  else
    mapPattern(m, 1);
}

DataGraph G = loadDataGraph("input.graph");
Pattern wedge = generateStar(3);
numTriplets = 2*count(G, wedge);

Pattern triangle = generateClique(3);
Map<Pattern, Int> result = match(G, triangle, countAndCheck);
```

(b) Global Clustering Coefficient Bound

```
DataGraph G = loadDataGraph("input.graph");
Void output(Match m) { write(m); }
Pattern p = loadPattern("pattern.txt");
match(G, p, output);
```

(c) Pattern Matching

```
DataGraph G = loadDataGraph("input.graph");
Pattern p = generateClique(desiredSize);
Int result = count(G, p);
```

(d) Clique Counting

```
DataGraph G = loadDataGraph("input.graph");
Set<Pattern> patterns =
    generateAllVertexInduced(size);
Map<Pattern, Int> result = count(G, patterns);
```

(e) Motif Counting

```
Void found(Match m) {
  mapPattern(m, True);
  stopExploration();
}
DataGraph G = loadDataGraph("input.graph");
Pattern p = generateClique(desiredSize);
Map<Pattern, Bool> result = match(G, p, found);
```

(f) Clique Existence

Figure 4. Graph mining use cases in PEREGRINE's pattern-aware programming model.

§4.3), and hence, it directly matches only those subgraphs that satisfy absence of vertices across neighborhoods as defined by anti-vertex constraint.

3.1.3 Edge-Induced and Vertex-Induced Patterns.

Depending on the mining use case, the matches for a given pattern must be either *edge-induced* or *vertex-induced*. For example, Frequent Subgraph Mining (FSM) relies on edge-induced matches, whereas Motif Counting requires vertex-induced matches (programs shown in §3.2). Our pattern-based programming model allows exploring subgraphs in both edge-based and vertex-based fashion as discussed next.

An edge-induced match is a subgraph s_e of G such that the subgraph of G induced by $E(s_e)$ is isomorphic to p. Note that, by definition, the subgraph induced by $E(s_e)$ is equal to s_e. Hence, edge-induced matches are directly expressed by the pattern.

A *vertex-induced* match, on the other hand, is a subgraph s_v of G such that the subgraph of G induced by $V(s_v)$ is isomorphic to p. Hence, the sets of vertex-induced and edge-induced matches of p are not equal mainly because the vertices of s_e can have more edges between them in G than are present in p (in general, the subgraph induced by $V(s_e)$ is not isomorphic to p). In our pattern-based programming approach, the vertex-induced requirement gets directly expressed using anti-edges. Specifically, to find vertex-induced matches of a pattern p, we use the following result:

Theorem 3.1. *Let p be a pattern, and p' be another pattern such that p' has the same vertices and edges as p, and every pair of vertices in p that are not adjacent are anti-adjacent in p'. The set of vertex-induced matches of p is equal to the set of edge-induced matches of p'.*

Proof. We skip the proof due to limited space. An edge-induced match m' for p' shares the same edges and vertices as a vertex-induced match m for p, and does not have edges between any vertices which are not adjacent in p. □

Hence, our pattern-based programming doesn't need to separately define the exploration strategy, as done in other pattern-unaware systems [52, 57].

3.2 Pattern-Aware Mining Programs in PEREGRINE

Figure 4 shows PEREGRINE programs for motif counting, frequent subgraph mining (FSM), clique counting, pattern matching, an existence query for global clustering coefficient bound, and an existence query for k-sized clique. All the programs first express patterns by dynamically generating them or by loading them from external source. Then they invoke PEREGRINE engine to find (`match()`) and process matches of those patterns. For every match for the pattern, user-defined function (e.g., `updateSupport()`, `countAndCheck()`, `found()`, etc.) gets invoked to perform desired analysis. The `count()` function is a syntactic

```
ExplorationPlan generatePlan(Pattern p) {
  partialOrders = breakSymmetries(p);
  vc = minConnectedVertexCover(p);
  pC = vertexInducedSubgraph(vc, p);
  matchingOrders = computeMatchingOrders(pC,
      partialOrders);
  return (pC, partialOrders, matchingOrders);
}
```

Figure 5. Computing exploration plan.

sugar and is equivalent to `match()` with a function that increments a counter. Most of the programs are straightforward; we discuss FSM and existence queries in more detail.

3.2.1 FSM: Anti-Monotonicity & Label Discovery. FSM leverages anti-monotonicity in support measures (discussed in §2.1). PEREGRINE natively provides MNI support computation where it internally constructs the *domain* of patterns, i.e., a table mapping vertices in G to those in p (similar to [1]). After exploration ends for a single iteration, the support measure maintained by PEREGRINE can be directly used to prune infrequent patterns using a threshold, as shown in Figure 4a, and only the remaining frequent patterns are then programmatically extended to be explored.

Before finding the first small frequent labeled patterns, the FSM program has no information about which labelings are frequent. PEREGRINE provides dynamic label discovery by starting with unlabeled (or partially labeled) patterns as input and returning labeled matches. Hence, the FSM program in Figure 4a first starts with unlabeled patterns of size 2, and discovers frequent labeled patterns. It then iteratively extends the frequent labeled patterns with unlabeled vertices to discover frequent labeled patterns of larger sizes.

3.2.2 Existence Queries. Existence queries allow quickly verifying whether certain structural properties hold within a given data graph. PEREGRINE allows dynamically stopping exploration when the required conditions get satisfied.

Figure 4b shows a PEREGRINE program to verify if the global clustering coefficient [27] of graph G is above a certain bound. The global clustering coefficient is the ratio of three times the number of triangles and the number of triplets (all connected subgraphs with three vertices, including duplicates) in G. The number of triplets is equal to twice the number of edge-induced 3-star matches since the endpoints of a 3-star are symmetric. Hence, the program quickly computes the number of 3-stars, and then starts counting triangles. During exploration, if the number of triangles reaches the requisite number to exceed the bound, exploration stops immediately.

Figure 4f shows PEREGRINE program to check whether a clique of a certain size is present in G. As soon as the exploration finds at least one match, it stops and returns `True`.

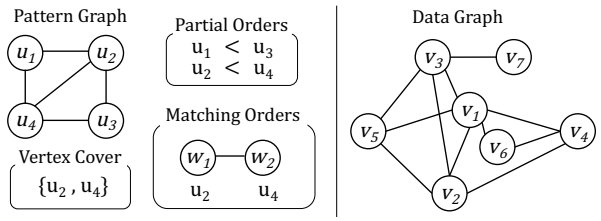


Figure 6. Example of a pattern graph and a data graph.

4 Pattern-Aware Matching Engine

PEREGRINE is pattern-aware, and hence, it directly finds patterns in any given data graph. In this section, we discuss our core pattern matching engine that directly finds canonical subgraphs from a given vertex in the data graph. In §5, we will use this engine to build PEREGRINE. For simplicity, we assume the data graph and the pattern are unlabeled.

4.1 Directly Matching A Given Pattern

To avoid the overheads of a straightforward exhaustive search, we develop our pattern matching solution based on well-established techniques [5, 16, 24]. Since patterns are much smaller than the data graph, we analyze the given pattern to develop an exploration plan. This plan guides the data graph exploration to ensure generated matches are unique.

Figure 5 shows how the exploration plan is computed from a given pattern p. First, to avoid non-canonical matches we break the symmetries of p by enforcing a partial ordering on matched vertices [16]. For our example pattern in Figure 6, we obtain the partial ordering $u_1 < u_3$ and $u_2 < u_4$.

In the next step, we compute the core of p (called p_C) as the subgraph induced by its minimum connected vertex cover [2]. Given a match m for p_C, all matches of p which contain m can be computed from the adjacency lists of vertices in m. In our example, p_C is the subgraph induced by u_2 and u_4.

To simplify the problem of matching p_C, we generate matching orders to direct our exploration in the data graph. A matching order is a graph representing an ordered view of p_C. The vertices of the matching order are totally-ordered such that the partial ordering of $V(p)$ restricted to $V(p_C)$ is maintained. This allows matching p_C by traversing vertices with increasing vertex ids without canonicality checks.

We compute matching orders by enumerating all sequences of vertices in p_C that meet the partial ordering, and for each sequence we create a copy of p_C where the id of each vertex is remapped to its position in the sequence. Then, we discard duplicate matching orders. For our example pattern (Figure 6), its core substructure has only one valid vertex sequence, $\{u_2, u_4\}$, so we obtain only one matching order. Note that there can be multiple matching orders for a given p_C depending on the partial orders. We call the i^{th} matching order p_{Mi}.

[2] A connected vertex cover is a subset of connected vertices that covers all edges.

```
Void matchFrom(Match m, Pattern p, Func f,
    MatchingOrder mo, PartialOrder po, Int i) {
  if (i > |V(pC)|) {
    // remaps m as in §4.1, before completing it
    // and invoking the user's callback f()
    completeMatch(m, p, f, po, 1);
  } else {
    for (v in getExtensionCandidates(mo, po, i)) {
      matchFrom(m+v, p, f, mo, po, i+1);
    }
  }
}

AggregationVal match(Graph G, Pattern p, Func f) {
  Aggregator a;
  (pC, partialOrder, matchingOrders) =
      generatePlan(p);
  parallel for (v in G) {
    for (matchingOrder in matchingOrders) {
      matchFrom({v}, p, f, matchingOrder,
                partialOrders, 1);
    }
  }
  return a.result();
}
```

Figure 7. Pattern-Aware Processing Model.

Thus, to match p_C it suffices to match its matching orders p_{Mi}. A match for p_{Mi} results in 1 match for p_C per valid vertex sequence. In our example, a match for p_{M1}, say $\{v_2, v_3\}$, is converted to a single match for p_C, $v_2 \rightarrow w_1 \rightarrow u_1, v_3 \rightarrow w_2 \rightarrow u_2$.

It is important to note that the exploration plan is generated by analyzing the pattern graph only, i.e., all the computations explained above are applied on p (and its derivatives). Hence, exploration plans are computed quickly (often in less than half a millisecond).

4.2 Matching Anti-Edges

To enforce an anti-edge constraint, we perform a set difference between the adjacency lists of its endpoints. For example, if v_i, v_j match u_1, u_2 of p_a in Figure 3, the candidates for u_4 are the elements of $\text{adj}(v_i) \setminus \text{adj}(v_j)$.

To perform the set difference, we need to ensure that one of the vertices of the anti-edge is already matched so that its adjacency list is available. Hence, when computing the vertex cover we also cover the anti-edge by including one of its endpoints. When computing partial orders, however, we do not need to consider anti-edges since they don't generate automorphic matches and we only verify absence of edges (i.e., we never traverse through anti-edges).

4.3 Matching Anti-Vertices

The anti-vertex semantics offer flexibility to match them differently compared to anti-edges. Anti-vertices break symmetries and do not impact the core graph (i.e., vertex cover computation) as described next.

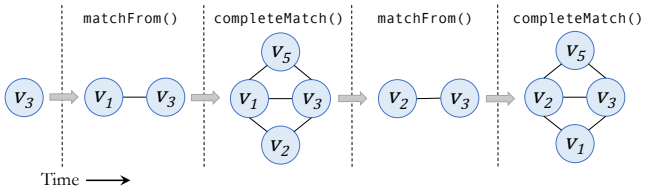


Figure 8. Pattern-guided exploration in PEREGRINE for pattern and data graph in Figure 6 with matching order high-to-low.

Checking Anti-Vertices. The anti-vertex constraint can only be verified after the common neighbors of an anti-vertex's neighbor have been matched. Thus, we perform the check after all true vertices are already matched.

For example, consider p_e in Figure 3, with anti-vertex u_4. If v_i, v_j, v_k in the data graph match u_1, u_2, u_3 respectively, then we verify the anti-vertex constraint for u_4 as follows:

$$(\text{adj}(v_i) \setminus \{v_j\}) \cap (\text{adj}(v_k) \setminus \{v_j\}) = \varnothing$$

Since anti-vertices do not participate while matching regular vertices and edges, we keep the pattern core p_C the same as the core pattern when anti-vertices are removed.

Breaking Symmetries with Anti-Vertices. We expose the asymmetry introduced by anti-vertices to the symmetry-breaking algorithm, which treats the anti-edges of an anti-vertex differently than regular edges.

5 PEREGRINE: Pattern-Aware Mining

We will now discuss how PEREGRINE performs pattern-aware mining using the matching engine presented in §4.

5.1 Pattern-Aware Processing Model

Mining in PEREGRINE is achieved by matching patterns starting from each vertex and invoking the user function to process those patterns. Hence, a task in PEREGRINE is defined as the data vertex where the matching process begins. As shown in Figure 7, each mining task takes a start vertex and the exploration plan generated in §4 (matching orders, partial orders, pattern core p_C). From the starting vertex, we recursively match vertices in the matching order. At each recursion level, a data vertex is matched to a matching order vertex. To avoid non-canonical matches, we maintain sorted adjacency lists and use binary search to generate candidate sets comprised only of vertices that meet the total ordering.

Once a matching order is fully matched, it is converted to matches for p_C. Matches for p_C are then completed by performing set intersections (for true edges) and set differences (for anti-edges) on sections of adjacency lists that satisfy the partial orders. Each completed match is passed to a user-defined callback for further processing. Figure 8 shows a complete exploration example.

Note that our processing model doesn't incur expensive isomorphism and canonicality checks for every match in the

data graph, while simultaneously avoiding mis-matches and only exploring subgraphs that match the given pattern. Furthermore, tasks in our processing model are independent of each other since explorations starting from two different vertices do not require any coordination. Threads dynamically pick up new tasks when they finish their current ones.

5.2 Early Pruning for Dynamic Load Balancing

While a matching order enforces a total ordering on the data vertices matching p_C, there is flexibility in the order in which its vertices are matched. To reduce the load imbalance across our matching tasks, we: (a) follow matching orders high-to-low, e.g. in our example in Figure 6 we match w_2 before w_1; and, (b) order vertices by their degree such that $v_i < v_j$ in the data graph if and only if $degree(v_i) \leq degree(v_j)$.

High-degree vertices have fewer neighbors with degrees higher than or equal to their own, so the degree-based ordering ensures that when a high-degree vertex is matched to w_2, only those few neighbors can be matched to w_1. Thus, explorations of neighbors with lower degrees are pruned. Note that the total number of matches generated remains the same; the high-to-low matching order traversal, along with degree-based vertex ordering, reduces the workload imbalance of matching across high-degree and low-degree vertices by dynamically pruning more explorations from high-degree tasks while enabling those explorations in low-degree tasks.

Finally, it is important to note that this process does not 'eliminate' workload imbalance simply because the mining workload is dynamic and depends on the pattern and data graphs. Hence, to avoid stragglers and maximize parallelism, we process tasks in the order defined by the degree of the starting vertex, beginning with the highest-degree vertices.

5.3 Early Termination for Existence Queries

For existence queries, PEREGRINE allows actively monitoring the required conditions so that the exploration process terminates as quickly as possible. When the matching thread observes the required conditions, the user function calls `stopExploration()` to notify other matching threads. Threads monitor their notifications periodically while matching, and when a notification is observed, the thread-local values computed up to that point are aggregated and returned to the user.

5.4 On-the-fly Aggregation

PEREGRINE performs on-the-fly aggregation to provide global updates as mining progresses. This is useful for early termination and for use cases like FSM where patterns that meet the support threshold can be deemed frequent while matching continues for other patterns.

We achieve this using an asynchronous aggregator thread that periodically performs aggregation as values arrive from threads. The matching threads swap the global aggregation value with the local aggregation value and set a flag to indicate that new thread-local aggregation values are available for aggregation. The aggregator thread blocks until all thread-local aggregation values become available, after which it performs the aggregation and resets the flag to indicate that the global aggregation value is available. With this design, our matching threads remain non-blocking to retain high matching throughput.

5.5 Implementation Details

PEREGRINE is implemented in C++ where concurrent threads operate on exploration tasks, each starting at a different vertex in the data graph. The data graph is represented using adjacency lists, and the tasks are distributed dynamically using a shared atomic counter indicating the next vertex to be processed. To minimize coordination, threads maintain information regarding their exploration tasks, including candidate sets for each pattern vertex as exploration proceeds.

PEREGRINE provides native computation of support values for frequency-based mining tasks like FSM. Domains are implemented as a vector of bitmaps representing the data vertices that can be mapped to each pattern vertex. They are aggregated by merging their contents via logical-or. To scale to large datasets, we use compressed Roaring bitmaps [7], which are more memory efficient than dense bitmaps.

6 Evaluation

We evaluate the performance of PEREGRINE on a wide variety of graph mining applications and compare the results with the state-of-the-art general purpose graph mining systems [3]: Fractal [12], Arabesque [52], RStream [57] and G-Miner [8].

6.1 Experimental Setup

System. All experiments were conducted on `c5.4xlarge` and `c5.metal` Amazon EC2 instances. Most experiments use `c5.4xlarge`, with an Intel Xeon Platinum 8124M CPU containing 8 physical cores (16 logical cores with hyper-threading), 32GB RAM, and 30GB SSD. Fractal (FCL), Arabesque (ABQ) and G-Miner (GM) were evaluated using both a cluster of 8 nodes (denoted by the suffix '-8'), as well as in single node configuration (denoted by the suffix '-1').

RStream was evaluated on a `c5.4xlarge` (RS-16) as well as a `c5.metal` (RS-96) equipped with an Intel Xeon Scalable Processor containing 48 physical cores (96 logical cores with hyper-threading), and 192GB RAM. Both instances were provisioned with a 500GB SSD.

In all performance comparisons, we ran PEREGRINE on a `c5.4xlarge`, and we used `c5.metal` to study PEREGRINE's scalability and resource utilization.

Datasets. Table 2 lists the data graphs used in our evaluation. Mico (MI) is a co-authorship graph labeled with each

[3] We could not evaluate AutoMine [34] since its source code is not available.

G	$\|V(G)\|$	$\|E(G)\|$	$\|L(G)\|$	Max. Deg.	Avg. Deg.
(MI) Mico [13]	100K	1M	29	96637	21.6
(PA) Patents [17]					
└ Unlabeled	3.7M	16M	—	793	10
└ Labeled	2.7M	13M	37	789	10
(OK) Orkut [60]	3M	117M	—	33133	76
(FR) Friendster [60]	65M	1.8B	—	5214	55

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

author's research field. Patents (PA) is a patent citation graph. In the labeled version, each patent is labeled with the year it was granted. Orkut (OK) and Friendster (FR) are unlabeled social network graphs where edges represent friendships between users. Mico and labeled Patents have been used by previous systems [12, 52, 57] to evaluate FSM while Orkut and Friendster were used by [8]. Except for FSM and labeled pattern matching, all experiments on Patents use its larger, unlabeled version.

Applications. We evaluated PEREGRINE on a wide array of applications: counting motifs with 3 and 4 vertices, labeled 3- and 4-motifs; counting k-cliques, for k ranging from 3 to 5; FSM with patterns of 3 edges on labeled datasets using various supports; matching the patterns shown in Figure 9; and checking the existence of 14-cliques. We selected the patterns in Figure 9 to cover all the patterns used in [12] and [8]; note that patterns like triangles and empty squares are covered via applications like cliques and motifs. Since G-Miner's pattern matching program is specific to labeled p_2 (in Figure 9), we used labels on p_2 for all the systems to enable direct comparison. To match it on Orkut and Friendster graphs, which are unlabeled, we added synthetic labels (integers 1-6 as done in [8]) with uniform probability.

6.2 Comparison with Breadth-First Enumeration

Table 3 compares PEREGRINE's performance with Arabesque and RStream on motif-counting, clique-counting, and FSM (these systems do not support pattern matching). As we can see, PEREGRINE outperforms the breadth-first systems by at least an order of magnitude on every application except FSM. RStream, despite being an out-of-core system, runs out of memory during FSM computations because of the massive amounts of aggregation information, as well as during 4- and 5-cliques on Mico where it could not handle the size of a single expansion step.

It was interesting to observe that Arabesque performed better in single-node mode compared to 8-node configuration across all experiments except FSM on Patents, where it ran out of memory. This is because its breadth-first exploration generates large amounts of partial matches which must be synchronized across the entire cluster between supersteps, incurring high communication costs that impact its scalability.

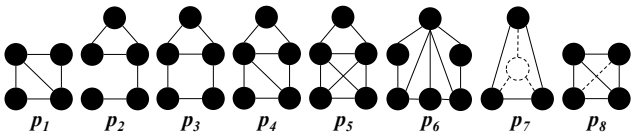


Figure 9. Patterns used in evaluation.

When support thresholds are high, Arabesque on 8 nodes computes FSM more quickly than PEREGRINE. This is because its breadth-first strategy leverages high parallelism when there are few frequent patterns to explore and aggregate. However, this approach is sensitive to the support threshold, which stops Arabesque from scaling to lower threshold values where there are more frequent patterns. In these scenarios Arabesque simply fails due to the memory burden of maintaining the vast amount of intermediate matches and aggregation values. We suspect that even with more main memory per node, the intermediate computations (canonicality, isomorphism, etc.) for each individual match in Arabesque would significantly limit its performance. Since PEREGRINE is pattern-aware, it only needs to maintain aggregation values for the patterns it is currently matching, allowing it to scale to inputs that yield many frequent patterns.

6.3 Comparison with Depth-First Enumeration

Table 4 compares PEREGRINE's performance with Fractal on motif-counting, clique-counting, FSM, and pattern matching. As we can see, PEREGRINE is faster than Fractal by at least an order of magnitude across most of the applications. For instance, 4-cliques on Patents finished in less than a second on PEREGRINE whereas Fractal took over 200 seconds in both cluster and single-node configurations.

Given equal resources (i.e., on a single node), FSM on Mico is up to 2.6× faster on PEREGRINE compared to that on Fractal. Furthermore, PEREGRINE scales to the larger dataset while Fractal does not. Even with 8 nodes, Fractal only outperforms PEREGRINE on the small Mico graph, and cannot handle the Patents workload except for very high support thresholds, where there is less work to be done; there too, PEREGRINE executes faster than Fractal.

Similar to Arabesque, Fractal's pattern-unawareness requires it to maintain global aggregation values throughout its computation. In FSM, the aggregation values consume $O(|V|)$ memory per vertex in each pattern in the worst case, and thus quickly become a scalability bottleneck. On the other hand, PEREGRINE only needs to maintain aggregation values for the current patterns being matched, which allows it to achieve comparable performance and superior scalability while matching up to 15,817 patterns on Mico and 6,739 patterns on Patents.

App	G	PEREGRINE	Arabesque		RStream	
			ABQ-8	ABQ-1	RS-96	RS-16
3-Motifs	MI	**0.12**	158.05	39.05	51.83	252.74
	PA	**3.10**	870.70	525.49	2685.45	2186.93
	OK	**17.90**	—	—	/	/
	FR	**370.64**	—	—	/	/
4-Motifs	MI	**6.74**	—	—	/	/
	PA	**12.04**	—	—	/	×
	OK	**6156.10**	—	—	/	/
2K-FSM	MI	**380.81**	3418.25	821.60	×	—
3K-FSM	MI	**279.74**	3520.82	784.27	×	—
4K-FSM	MI	**250.68**	3514.97	779.75	×	—
20K-FSM	PA	**859.41**	—	—	1757.69	—
21K-FSM	PA	**647.97**	—	—	1711.87	—
22K-FSM	PA	507.56	**342.63**	—	1626.53	—
23K-FSM	PA	402.57	**299.12**	—	1936.92	—
3-Cliques	MI	**0.05**	18.62	5.98	7.34	11.32
	PA	**0.59**	155.55	87.26	8.40	11.97
	OK	**13.75**	—	—	986.20	1643.10
	FR	**296.99**	—	—	/	/
4-Cliques	MI	**2.02**	1598.09	353.37	266.61	—
	PA	**0.90**	249.38	107.02	105.00	181.30
	OK	**281.47**	—	—	/	/
	FR	**1337.77**	—	—	/	/
5-Cliques	MI	**89.60**	×	—	—	—
	PA	**1.12**	352.64	122.09	145.00	237.90
	OK	**3182.56**	—	—	/	/
	FR	**4214.72**	—	—	/	/

Table 3. Execution times (in seconds) for PEREGRINE, Arabesque [52] and RStream [57].
'×' indicates the execution did not finish within 5 hours.
'—' indicates the system ran out of memory.
'/' indicates the system ran out of disk space.

6.4 Comparison with Purpose-Built Algorithms

G-Miner is a general-purpose subgraph-centric system that targets expert users to implement the mining algorithms using a low-level subgraph data structure. Since expressing common mining algorithms requires domain expertise, we only evaluated the applications that are already implemented in G-Miner: 3-clique counting and pattern matching on p_2 (pattern matching for other patterns is not supported). This experiment serves to showcase how PEREGRINE compares to custom algorithms for matching specific patterns.

Table 5 compares PEREGRINE's performance with G-Miner. As we can see, PEREGRINE is 3× to 77× faster than G-Miner when counting 3-cliques even though G-Miner implements an algorithm designed specifically to count 3-cliques. When matching p_2, PEREGRINE is 6× to 131× faster on Mico and Patents. On Orkut, however, G-Miner performs better on finding p_2; this is because G-Miner indexes vertices by labels when preprocessing the data graph, whereas PEREGRINE discovers labels dynamically. Due to these indexes, G-Miner could not handle Friendster even with 240GB disk space on the cluster.

App	G	PEREGRINE	Fractal	
			FCL-8	FCL-1
3-Motifs	MI	**0.12**	22.13	17.11
	PA	**3.10**	231.95	214.34
	OK	**17.90**	—	—
	FR	**370.64**	—	—
4-Motifs	MI	**6.74**	78.66	420.67
	PA	**12.04**	362.19	742.35
	OK	**6156.10**	—	—
2K-FSM	MI	380.81	**154.47**	675.98
3K-FSM	MI	279.74	**154.74**	680.33
4K-FSM	MI	250.68	**144.34**	663.26
20K-FSM	PA	**851.41**	×	—
21K-FSM	PA	**647.97**	×	—
22K-FSM	PA	**507.56**	×	—
23K-FSM	PA	**402.57**	451.18	—
3-Cliques	MI	**0.05**	18.71	17.21
	PA	**0.59**	232.60	216.76
	OK	**13.75**	—	—
	FR	**296.99**	—	—
4-Cliques	MI	**2.02**	25.77	34.79
	PA	**0.90**	237.64	224.50
	OK	**281.47**	—	—
	FR	**1337.77**	—	—
5-Cliques	MI	**89.60**	181.30	904.65
	PA	**1.12**	266.88	217.30
	OK	**3182.56**	—	—
	FR	**4214.72**	—	—
Match p_1	MI	**0.12**	24.76	36.02
	PA	**0.84**	235.72	189.03
	OK	**38.97**	—	—
	FR	**824.62**	—	—
Match p_2	MI	**0.03**	22.11	16.85
	PA	**1.07**	260.15	202.23
	OK	**474.09**	—	—
	FR	**18.09**	—	—
Match p_3	MI	**19.93**	181.76	1288.94
	PA	**13.41**	30.18	69.33
	OK	**13292.77**	—	—
Match p_4	MI	**12.29**	120.99	789.81
	PA	**2.23**	25.58	21.63
	OK	**1569.73**	—	—
	FR	**7057.40**	—	—
Match p_5	MI	**14.94**	56.51	345.35
	PA	**1.89**	25.30	17.39
	OK	**1381.03**	—	—
	FR	**6726.51**	—	—
Match p_6	MI	**65.26**	×	×
	PA	**27.94**	210.04	205.39

Table 4. Execution times (in seconds) for PEREGRINE and Fractal [12]. '—' indicates the system ran out of memory.
'×' indicates the execution did not finish within 5 hours.

App	G	PEREGRINE	G-Miner	
			GM-8	GM-1
3-Cliques	MI	**0.05**	3.79	3.86
	PA	**0.59**	7.91	8.93
	OK	**13.75**	44.26	62.65
	FR	**296.99**	/	/
Match p_2	MI	**0.03**	3.67	3.95
	PA	**1.07**	6.84	9.80
	OK	474.09	**145.00**	396.72
	FR	**18.09**	/	/

Table 5. Execution times (in seconds) for PEREGRINE and G-Miner [8]. '/' indicates the system ran out of disk space.

G	Existence 14-Clique	Anti-Vertex p_7	Anti-Edge p_8
MI	0.07	0.65	6.92
PA	3.95	0.67	1.69
OK	4.08	56.06	879.01
FR	50.39	470.21	4017.15

Table 6. PEREGRINE execution times (in seconds) for matching with an anti-vertex (p_7), matching with an anti-edge (p_8), and 14-clique existence query.

6.5 Mining with Constraints in PEREGRINE

We evaluate PEREGRINE on mining tasks with structural constraints. We match a pattern containing an anti-vertex (p_7), one containing an anti-edge (p_8), and perform an existence query of a 14-clique. The results are show in Table 6.

Mining with Anti-Vertices. Pattern p_7 expresses a maximal clique of size 3 (triangle) using a fully-connected anti-vertex, i.e., it matches all triangles that are not contained in a 4-clique. While satisfying the anti-vertex constraint requires computing set-intersections across all vertices of the triangle, PEREGRINE takes less than a minute on Orkut, and under eight minutes on the billion scale Friendster graph.

Mining with Anti-Edges. Pattern p_8 represents a vertex-induced chordal square using an anti-edge constraint. Satisfying the anti-edge constraint is computationally demanding, since it requires computing set differences of adjacency lists, which is twice as many operations as the sum of the adjacency list sizes. Nevertheless, PEREGRINE still easily completes it on all the datasets.

Existence Query. The goal of this query is to determine whether a 14-clique exists in the data graph. PEREGRINE stops exploration immediately after finding an instance of 14-clique. We observe that Patents and Orkut performed similarly; this is because the rarer the target pattern for an existence query, the longer it takes to find it. Patents does not contain a 14-clique, so the entire graph was searched, but in the much larger and denser Orkut graph, a 14-clique gets found quickly during exploration. Friendster is both large and sparse, and hence, 14-cliques are rare. Furthermore, since 14-clique is a large pattern, several partial explorations do not lead to a complete 14-clique.

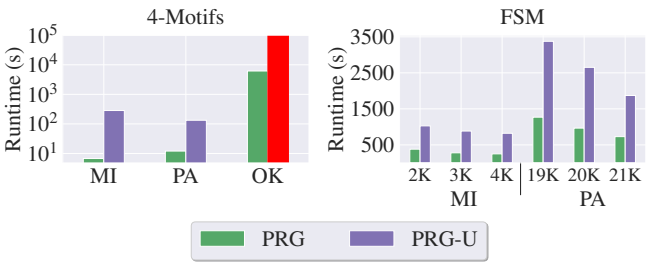


Figure 10. Execution times (in seconds) for PEREGRINE with (PRG) and without (PRG-U) symmetry breaking. PRG-U could not finish matching any of the 4-motif patterns on Orkut within 5 hours.

6.6 PEREGRINE's Pattern-Aware Runtime

Benefits of Symmetry Breaking. Symmetry breaking is a well-studied technique for subgraph matching that PEREGRINE uses to guide its graph exploration. However, recent systems like Fractal [12] and AutoMine [34] are not fully pattern-aware and do not leverage symmetry breaking for common graph mining use cases. We showcase the importance of symmetry breaking in PEREGRINE by disabling it and running 4-motifs and FSM with low support thresholds. These are expensive subgraph matching workloads: 4-motifs contains complex patterns with many matches and FSM involves a large number of patterns to match. Figure 10 summarizes the results.

We observe that symmetry breaking improves performance by an order of magnitude for 4-motifs on Mico and Patents. Orkut 4-motifs without symmetry breaking did not even finish matching even a single size 4 pattern within 5 hours. This shows the importance of symmetry breaking when scaling to large patterns and large datasets. For instance, Orkut contains over 22 trillion *unique* vertex-induced 4-stars, and so without symmetry breaking, the system must process six times that many matches (a 4-star's automorphisms are the permutations of its 3 endpoints: resulting in 3! = 6 automorphic subgraphs).

FSM achieves 3× performance improvement through symmetry breaking. This is because with symmetry breaking, FSM's expensive aggregation values are only written to once per unique match in the data graph, whereas the naive approach without symmetry breaking would incur dozens of redundant write (and read) accesses per unique match.

Breakdown on Mining Time. Figure 12 shows the ratio of time spent in each stage of matching during 4-motif execution: finding the range of sorted candidate sets that meet the pattern's partial order (PO), performing adjacency list

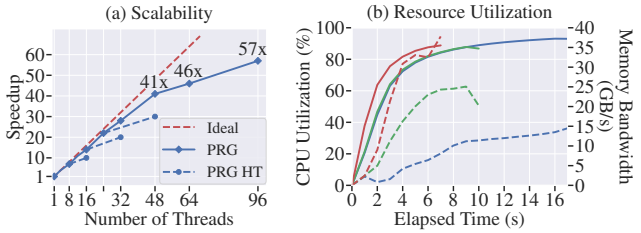


Figure 11. (a) Scalability (PRG HT = hyper-threaded). (b) CPU utilization (solid) and memory bandwidth (dashed) for 24 cores (blue), 47 cores (green) and 94 cores (red).

intersections and differences to match the pattern core (Core) and finally, intersecting the adjacency lists of the pattern core to complete the match (Non-Core). Some time is also spent on the other requirements of matching, for example, fetching adjacency lists and mapping vertices (Other).

We observe that the majority of execution is spent intersecting adjacency lists of candidate vertices to complete matches. In comparison to the overall execution time, matching the core pattern is insignificant. This is because the core pattern is matched according to all valid total orderings of its vertices, and hence, the traversal is fully guided. In contrast, the non-core vertices may or may not be ordered with respect to each other, and with respect to the core vertices; so the runtime usually has less guidance when exploring the graph. Furthermore, in most patterns the core is small and involves fewer intersections than the non-core component.

6.7 System Characteristics

Scalability. We study PEREGRINE's scalability by matching pattern p_1 on Orkut using `c5.metal` instance. Note that we do not perform a COST analysis [37] with this experiment since we already compared PEREGRINE with optimized algorithms in §6.4, and state-of-the-art serial pattern matching solutions like [19, 53] performed much slower than our single threaded execution.

Figure 11a shows how PEREGRINE scales as number of threads increase from 1 to 96. As we can see, PEREGRINE scales linearly until 48 threads, after which speedups increase gradually. This is mainly because `c5.metal` has 48 physical cores, and scheduling beyond 48 threads happens with hyper-threading. We verified this effect by alternating how

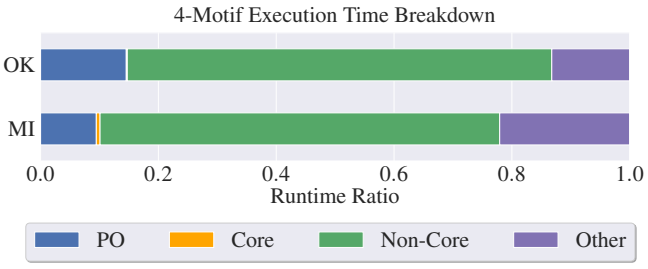


Figure 12. PEREGRINE 4-motif execution time breakdown.

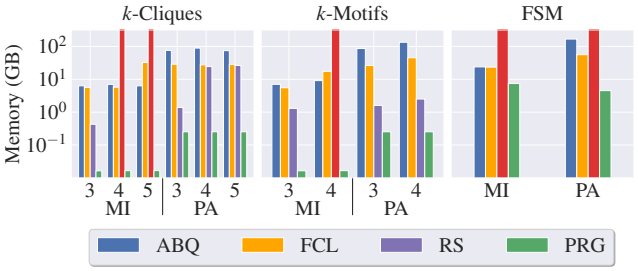


Figure 13. Peak memory usage of different systems across various applications. Tall red bars represent RStream out of memory errors.

threads get scheduled across different cores; the dashed lines in Figure 11a show speedups when every pair of PEREGRINE threads is pinned to two logical CPUs on one physical CPU. As we can see, with 48 threads but only 24 physical cores, PEREGRINE only achieves a 30× speedup, whereas with 48 physical cores it achieves a 41× speedup. Since pattern exploration involves continuous random memory accesses throughout execution, hyper-threading helps in hiding memory latencies only up to an extent. Figure 11b verifies this, as memory bandwidth grows considerably higher when using more cores, though CPU utilization remains similar.

We observe that speedups also decline slightly between 24 cores and 48 cores. This is because `c5.metal` has two NUMA nodes, each allocated to 24 physical cores. We measured remote memory accesses to observe the NUMA effects: when running on 48 cores, cross-numa memory traffic was 86GB as opposed to only 4.9MB when running on 24 cores.

Resource Utilization. Figure 11b shows CPU utilization and memory bandwidth consumed by PEREGRINE while matching p_1 on Orkut on `c5.metal` with 24, 47, and 94 threads. We reserve a core for profiling to avoid its overhead. We observe that PEREGRINE maintains high CPU utilization throughout its execution. The memory bandwidth curve increases over time; as high degree vertices finish processing, low degree vertices do less computation and incur more memory accesses as they get processed.

Figure 13 compares the peak memory usage for PEREGRINE and other systems. For distributed systems we report the sum of all nodes' peak memory. PEREGRINE consistently uses less memory than all the systems, mainly because of its direct pattern-aware exploration strategy. It is interesting to note that changing the pattern size in cliques and motifs does not impact PEREGRINE's memory usage. The usage is high for FSM compared to other applications due to large domain maps for support calculation.

Load Balancing. Since PEREGRINE threads dynamically pick up tasks as they become free, we observe near-zero load imbalance while matching p_1 across all our datasets. The difference between times taken by threads to finish all of their work was only up to 71 ms.

7 Related Work

There has been a variety of research to develop efficient graph mining solutions. To the best of our knowledge, PEREGRINE is the first general-purpose graph mining system to leverage pattern-awareness in its programming and processing models.

General-Purpose Graph Mining Systems. Several general-purpose graph mining systems have been developed [8, 12, 22, 34, 52, 57]. Arabesque [52] is a distributed graph mining system that follows a filter-process model developed on top of map-reduce. It proposed the "Think Like an Embedding" processing model. Fractal [12] extends this to the concept of *fractoids*, which expose parts of the user program to the system; in conjunction with depth-first exploration, fractoids allow the system to more intelligently plan its execution. G-Miner [8] is a task-oriented distributed graph mining system that enables building custom graph mining use cases using a distributed task queue. RStream [57] is a single machine out-of-core graph mining system that leverages SSDs to store intermediate solutions. It uses relational algebra to express mining tasks as table joins. AutoMine [34] is a recent single-machine system that generates efficient code to match patterns for common graph mining tasks. As discussed in §2.2, none of these systems are fully pattern-aware, the way PEREGRINE is: these systems perform unnecessary explorations and computations, require large memory (or storage) capacity, and lack the ability to easily express mining tasks at a high level. While Fractal uses symmetry breaking for pattern matching use case, other applications like FSM and motif counting are not guided by symmetry breaking, and hence they end up performing unnecessary explorations. Similarly, AutoMine also does not employ symmetry breaking for any of the use cases, requiring users to filter duplicate matches by individually examining every single match when enumerating patterns. Lack of full pattern-awareness not only makes these systems slower, but also limits their applicability to more complex mining use cases. Finally, ASAP [22] is a programmable distributed system for approximate graph mining where users write programs based on sampling edges and vertices to reason about the probabilistic counts of patterns.

Purpose-Built Graph Mining Solutions. These works efficiently perform specific graph mining tasks. ApproxG [35] is an efficient system for computing approximate graphlet (motif) counts with accuracy guarantees. [2] uses combinatorial arguments to obtain counts for size 3 and 4 motifs after counting smaller motifs. [10] efficiently lists k-cliques in sparse graphs and [4] is aimed at k-plexes which are clique-like structures. GraMi [13] leverages anti-monotonicity for FSM on a single machine while ScaleMine [1] is a distributed system for FSM that uses efficiently computable approximate stats to inform its graph exploration. [51] is also a distributed system focusing on FSM. [50, 61] are recent works aimed at analyzing small graphs whose edges have large attribute sets.

Several systems aim to perform efficient pattern matching. OPT [25] is a fast single-machine out-of-core triangle-counting system whose techniques are generalized by Dual-Sim [24] to match arbitrary patterns. [49] proposes several provably cache-friendly parallel triangle-counting algorithms which provide order-of-magnitude speedups over previous algorithms. DistTC [20] presents a distributed triangle-counting technique that leverages a novel graph partitioning strategy to count triangles with minimal communication overhead.

[31] is a distributed map-reduce based pattern matching system that first finds small patterns and joins them into large ones. QFrag [47] is another map-reduce based distributed pattern matching system that focuses on searching graphs for large patterns using the TurboISO [19] algorithm. PruneJuice [43] is a distributed pattern matching system that focuses on pruning data graph vertices that cannot contribute to a match. [18] is a scalable subgraph isomorphism algorithm while TurboFlux [26] performs pattern matching on dynamically changing data graphs. [40] presents a pattern matching plan optimizer incorporated in Graphflow [23] that uses both binary and multi-way joins. [44] is a resource-aware distributed graph querying system for property graphs.

Graph Processing Systems. Several works enable processing static and dynamic graphs [11, 14, 15, 21, 29, 32, 33, 42, 45, 46, 48, 54–56, 62]. These systems typically compute values on vertices and edges rather than analyzing substructures in graphs. They decompose computation at vertex and edge level, which is not suitable for graph mining use cases.

8 Conclusion

We presented PEREGRINE, a pattern-aware graph mining system that efficiently explores subgraph structures of interest, and scales to complex graph mining tasks on large graphs. PEREGRINE uses 'pattern-based programming' that treats *patterns* as first class constructs. We further introduced two novel abstractions: ANTI-EDGE and ANTI-VERTEX, that express advanced structural constraints on patterns to be matched. This allows users to directly operate on patterns and easily express complex mining use cases as 'pattern programs' on PEREGRINE.

Our extensive evaluation showed that PEREGRINE outperforms the existing state-of-the-art by several orders of magnitude, even when it has access to up to 8× fewer CPU cores. Furthermore, PEREGRINE successfully handles resource-intensive graph mining tasks on billion-scale graphs on a single machine, while the state-of-the-art fails even with a cluster of 8 such machines or access to large SSDs.

Acknowledgments

We would like to thank our shepherd Aleksandar Prokopec and the anonymous reviewers for their valuable and thorough feedback. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

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 61:1–61:12, 2016.

[2] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick Duffield. Efficient Graphlet Counting for Large Networks. In *IEEE International Conference on Data Mining (ICDM '15)*, pages 1–10, 2015.

[3] Peter S. Bearman, James Moody, and Katherine Stovel. Chains of Affection: The Structure of Adolescent Romantic and Sexual Networks. *American Journal of Sociology*, 110(1):44–91, 2004.

[4] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. Efficient Enumeration of Maximal k-Plexes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '15)*, pages 431–444, 2015.

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

[6] Bjorn Bringmann and Siegfried Nijssen. What Is Frequent in a Single Graph? In *Advances in Knowledge Discovery and Data Mining: 12th Pacific-Asia Conference*, volume 5012, pages 858–863, 2008.

[7] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better Bitmap Performance with Roaring Bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016.

[8] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-oriented Graph Mining System. In *Proceedings of the European Conference on Computer Systems (EuroSys '18)*, pages 32:1–32:12, 2018.

[9] Wei-Ta Chu and Ming-Hung Tsai. Visual pattern discovery for architecture image classification and product image search. In *Proceedings of the ACM International Conference on Multimedia Retrieval (ICMR '12)*, pages 1–8, 2012.

[10] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing K-cliques in Sparse Real-World Graphs*. In *Proceedings of the World Wide Web Conference (WWW '18)*, pages 589–598, 2018.

[11] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, pages 918–934, 2019.

[12] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '19)*, pages 1357–1374, 2019.

[13] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. In *Proceedings of the VLDB Endowment (PVLDB '14)*, pages 517–528, 2014.

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

[15] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, pages 599–613, 2014.

[16] 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.

[17] Bronwyn Hall, Adam Jaffe, and Manuel Trajtenberg. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. *NBER Working Paper 8498*, 2001.

[18] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '19)*, pages 1429–1446, 2019.

[19] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. TurboISO: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '13)*, pages 337–348, 2013.

[20] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. DistTC: High Performance Distributed Triangle Counting. In *IEEE High Performance Extreme Computing Conference (HPEC '19)*, pages 1–7, 2019.

[21] 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.

[22] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, pages 745–761, Carlsbad, CA, 2018.

[23] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '17)*, pages 1695–1698, 2017.

[24] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '16)*, pages 1231–1245, 2016.

[25] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. OPT: A New Framework for Overlapped and Parallel Triangulation in Large-scale Graphs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '14)*, pages 637–648, 2014.

[26] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '18)*, pages 411–426, 2018.

[27] Anton Korshunov, Ivan Beloborodov, Nazar Buzun, Valeriy Avanesov, Roman Pastukhov, Kyrylo Chykhradze, Ilya Kozlov, Andrey Gomzin, Ivan Andrianov, Andrey Sysoev, Stepan Ipatov, Ilya Filonenko, Christina Chuprina, Denis Turdakov, and Sergey Kuznetsov. Social Network Analysis: Methods and Applications. In *Proceedings of the Institute for System Programming of RAS*, pages 439–456, 2014.

[28] Frederick S. Kuhl, Gordon M. Crippen, and Donald K. Friesen. A combinatorial algorithm for calculating ligand binding. *Journal of Computational Chemistry*, 5(1):24–34, 1984.

[29] Pradeep Kumar and H Howie Huang. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '19)*, pages 249–263, 2019.

[30] Michihiro Kuramochi and George Karypis. Finding Frequent Patterns in a Large Sparse Graph*. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[31] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable Distributed Subgraph Enumeration. In *Proceedings of the VLDB Endowment (PVLDB '16)*, pages 217–228, 2016.

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

[33] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys '19)*, pages 25:1–25:16, 2019.

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

[35] Daniel Mawhirter, Bo Wu, Dinesh Mehta, and Chao Ai. ApproxG: Fast Approximate Parallel Graphlet Counting Through Accuracy Control. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '18)*, pages 533–542, 2018.

[36] Jean McGloin and David Kirk. An Overview of Social Network Analysis. *Journal of Criminal Justice Education*, 21:169–181, 2010.

[37] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at what COST? In *Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[38] Jinghan Meng and Yi-cheng Tu. Flexible and Feasible Support Measures for Mining Frequent Patterns in Large Labeled Graphs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '17)*, page 391–402, 2017.

[39] Pieter Meysman, Yvan Saeys, Ehsan Sabaghian, Wout Bittremieux, Yves Van de Peer, Bart Goethals, and Kris Laukens. Discovery of Significantly Enriched Subgraphs Associated with Selected Vertices in a Single Graph. In *Proceedings of the International Workshop on Data Mining in Bioinformatics (BIOKDD '15)*, volume 15, pages 1–8, 2015.

[40] Amine Mhedhbi and Semih Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. In *Proceedings of the VLDB Endowment (PVLDB '19)*, page 1692–1704, 2019.

[41] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, N Kashtan, Dmitri Chklovskii, and Uri Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298:824–7, 2002.

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

[43] Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. PruneJuice: Pruning Trillion-edge Graphs to a Precise Pattern-matching Solution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, pages 21:1–21:17, 2018.

[44] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *Proceedings of the International Workshop on Graph Data-Management Experiences & Systems (GRADES '17)*, 2017.

[45] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 410–424, 2015.

[46] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM '13)*, 2013.

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

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

[49] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE '15)*, pages 149–160, 2015.

[50] Qi Song, Mohammad Hossein Namaki, and Yinghui Wu. Answering Why-Questions for Subgraph Queries in Multi-attributed Graphs. In *IEEE International Conference on Data Engineering (ICDE '19)*, pages 40–51, 2019.

[51] Nilothpal Talukder and Mohammed J. Zaki. A Distributed Approach for Graph Mining in Massive Networks. *Data Mining and Knowledge Discovery*, 30(5):1024–1052, 2016.

[52] 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 ACM Symposium on Operating Systems Principles (SOSP '15)*, pages 425–440, 2015.

[53] Julian Ullmann. Bit-vector Algorithms for Binary Constraint Satisfaction and Subgraph Isomorphism. *Journal of Experimental Algorithmics*, 15:1–1, 2011.

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

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

[56] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, page 223–236, 2017.

[57] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, pages 763–782, 2018.

[58] Yuyi Wang and Jan Ramon. An Efficiently Computable Support Measure for Frequent Subgraph Pattern Mining. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD '12)*, pages 362–377, 2012.

[59] Liang Wu and Huan Liu. Tracing Fake-News Footprints: Characterizing Social Media Messages by How They Propagate. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM '18)*, pages 637–645, 2018.

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

[61] Gensheng Zhang, Damian Jimenez, and Chengkai Li. Maverick: Discovering Exceptional Facts from Knowledge Graphs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '18)*, pages 1317–1332, 2018.

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