A Deeper Dive into Pattern-Aware Subgraph Exploration with PEREGRINE

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

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

Abstract

Graph mining workloads aim to extract structural properties of a graph by exploring its subgraph structures. PEREGRINE is a general-purpose graph mining system that provides a generic runtime to efficiently explore subgraph structures of interest and perform various graph mining analyses. It takes a 'pattern-aware' approach by incorporating a pattern-based programming model along with efficient pattern matching strategies. The programming model enables easier expression of complex graph mining use cases and enables PEREGRINE to extract the semantics of patterns. By analyzing the patterns, PEREGRINE generates efficient exploration plans which it uses to guide its subgraph exploration.

In this paper, we present an in-depth view of the pattern-analysis techniques powering the matching engine of PEREGRINE. Beyond the theoretical foundations from prior research, we expose opportunities based on how the exploration plans are evaluated, and develop key techniques for computation reuse, enumeration depth reduction, and branch elimination. Our experiments show the importance of pattern-awareness for scalable and performant graph mining where the presented new techniques speed up the performance by up to two orders of magnitude on top of the benefits achieved from the prior theoretical foundations that generate the initial exploration plans.

1 Introduction

Graph mining based analytics has become popular across various important domains including bioinformatics, computer vision, and social network analysis [3, 9, 27, 35, 37, 53]. These tasks mainly involve computing structural properties of the graph, i.e., exploring and understanding the substructures within the graph. Finding specific subgraphs (also called *patterns*) of a graph is the *subgraph isomorphism* problem, which is NP-complete. 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.

PEREGRINE [1] is a recent graph mining system that takes a 'pattern-aware' approach to efficiently perform mining tasks on large graphs [23]. 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 presents the subgraphs back to the user space. By doing so, PEREGRINE directly mines the subgraphs of interest while avoiding exploration of unnecessary subgraphs, and simultaneously bypassing expensive computations (e.g., isomorphism and uniqueness checks) throughout the mining process.

PEREGRINE's 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, PEREGRINE introduces 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 (Figure 1 shows Frequent Subgraph Mining [6] implemented as a pattern program). Moreover, it enables PEREGRINE to extract the semantics of patterns which it uses to generate efficient exploration plans for its pattern-aware processing model.

PEREGRINE's efficient subgraph exploration runtime stems from two sources: first, theoretical foundations from existing subgraph matching research [5, 16] that generate exploration plans; and second, advanced matching strategies aimed at exploiting pattern structures for practical system-level benefits, chiefly by architecting the hot paths in the matching process to avoid performance pitfalls and maximize efficiency.

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

```
Void updateSupport(Match m) {
  mapPattern(m.domain());
}

Bool isFrequent(Pattern p, Domain d) {
  return (d.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);
}
```

Figure 1: Frequent Subgraph Mining implementation using PEREGRINE's pattern-based programming model.

In this paper, we focus on the pattern-aware subgraph exploration runtime of PEREGRINE. Specifically, we discuss key techniques including candidate set sharing, enumeration depth reduction, and branch elimination. Our advanced matching strategies include novel concepts of *neighborhood groups* and *match groups*. Neighborhood groups enable computation reuse by identifying pattern vertices which can share sets of candidate matching data vertices. Match groups further improve performance by exploiting the symmetries of pattern vertices to avoid branches in inner loops of the matching code and reducing the exploration depth. We taxonomize patterns into different classes based on the number of match groups they contain, and develop fast paths in PEREGRINE that completely eliminate branches in the final stages of matching for different pattern types.

Since PEREGRINE directly finds the subgraphs of interest, it does not incur additional processing over those subgraphs throughout its exploration process. Moreover, PEREGRINE does not maintain intermediate partial subgraphs in memory, resulting in a limited memory footprint throughout the mining process. PEREGRINE runs on a single machine and is highly concurrent. Evaluation conducted in [23] across several graph mining use cases on real-world graphs shows that PEREGRINE running on a single 16-core machine outperforms distributed graph mining systems [7, 12, 48] running on a cluster with eight 16-core machines. Furthermore, PEREGRINE could easily scale to large graphs and complex mining tasks which could not be handled by other systems.

To understand the benefits of our matching strategies that utilize neighborhood groups and match groups, we evaluate them across different pattern matching and motif counting tasks. Our results show that these techniques are crucial in retaining high performance as they speed up the exploration by up to two orders of magnitude on top of the benefits achieved from the prior theoretical foundations that generate the initial exploration plans.

2 Graph Mining Fundamentals

Graph mining involves exploring connected subgraphs of interest in a large data graph. Graph mining queries are diverse, ranging from simple pattern matching and motif counting queries where the structure of the subgraphs to explore is fixed and known ahead of time, to more complex use cases like frequent subgraph mining where exploration is guided by previous results. We represent subgraphs of interest with graph templates called *patterns*.

We define a *match* m of a pattern p as a subgraph of the data graph that is *isomorphic* to p, where isomorphism is a one-to-one mapping between the vertices of p and m such that if two vertices are adjacent in p, then their corresponding vertices are adjacent in m.

Since there can be sub-structural symmetries within p (e.g., a triangle structure looks the same when it is rotated), the same subgraph of the data graph can be represented by multiple different mappings from p into G. For both efficiency and ease-of-use, graph mining systems usually explore each match only once.

In PEREGRINE, users express graph mining tasks directly in terms of patterns. For example, k-Motif Counting is a matter of matching all patterns with k vertices. Frequent Subgraph Mining, on the other hand, lists all patterns that are frequent in the data graph, and it requires iteratively matching and extending patterns by an edge while filtering patterns that do not meet a frequency threshold (see Figure 1). PEREGRINE's pattern-aware runtime extracts the patterns involved in the graph mining use cases, finds all of their unique matches in the data graph, and processes the matches according to the user's instructions.

3 Directly Matching A Given Pattern

As patterns of interest are expressed by the graph mining program, the runtime traverses the data graph vertices in parallel,

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

Figure 2: Computing exploration plan.

finding all matches for the input patterns originating from each vertex. Since patterns are much smaller than the data graph, PEREGRINE analyzes the input patterns to develop an exploration plan. In addition to well-established techniques from the literature [5, 16, 24], PEREGRINE performs novel pattern analysis to optimize the exploration plan.

Figure 2 shows how the exploration plan is computed from a given pattern p, and Figure 3 shows the exploration plans for example patterns p_a and p_b. First, to avoid duplicate matches we break the symmetries of p by enforcing a partial ordering on matched vertices [16]. For p_a we obtain the partial ordering $u_1 < u_3$ and $u_2 < u_4$, and for p_b we obtain $u_1 < u_2$.

In the next step, we compute the core of p (called p_C) as the subgraph induced by its minimum connected vertex cover [2]. In our example, p_C for p_a is the subgraph induced by u_2 and u_4. 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 via set operations. The candidate set for u_1 in p_a is $\text{adj}(m(u_2)) \cap \text{adj}(m(u_4))$, for instance.

PEREGRINE performs further analysis to optimize the matching process for both core and non-core vertices beyond a naïve depth-first traversal of the pattern vertices. It is important to note that this analysis is performed on the pattern graph only, i.e., all the computations are applied on p (and its derivatives). Hence, exploration plans are computed quickly (often in less than half a millisecond).

3.1 Matching Orders

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 p restricted to 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

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

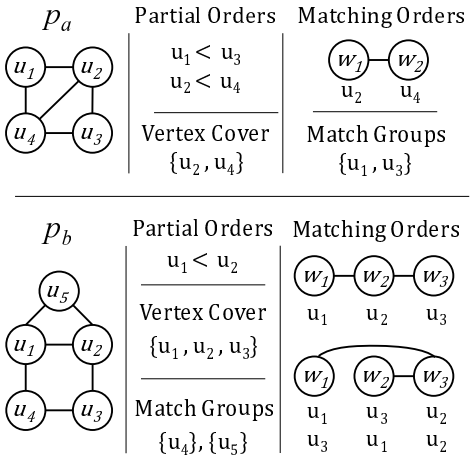

Figure 3: Examples of pattern graphs and their analysis. Neighborhood groups and match groups are equal in unlabeled patterns.

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 p_a, 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. For example, since u_3 in p_b is not ordered with respect to the other core vertices, there are three valid vertex sequences, which are reduced to two matching orders. We call the i^{th} matching order p_{Mi}.

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. For example, a match for p_{bM2}, say $\{v_1, v_2, v_3\}$, is converted to two matches for p_{bC}:

$$\{v_1 \to w_1 \to u_1, v_2 \to w_2 \to u_3, v_3 \to w_3 \to u_2\}$$
$$\{v_1 \to w_1 \to u_3, v_2 \to w_2 \to u_1, v_3 \to w_3 \to u_2\}$$

3.2 Neighborhood Groups

We observe that sets of non-core vertices with identical neighborhoods exhibit useful properties that further enable PEREGRINE to avoid redundant computation and reduce the match enumeration depth. PEREGRINE collects such vertices into *neighborhood groups*, which it leverages for several important optimizations. For example, p_a has one neighborhood group $\{u_1, u_3\}$, while in p_b there are two neighborhood groups $\{u_4\}$ and $\{u_5\}$, as the non-core vertices are adjacent to different core vertices.

3.2.1 Candidate Sets per Neighborhood Group

Since the vertices in a neighborhood group have the same core neighbors, they also have the same candidate matches. In p_a,

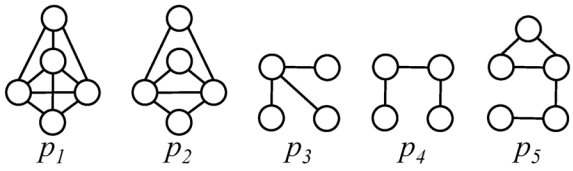

Figure 4: Patterns used in evaluation.

the non-core vertices vertices u_1 and u_3 are both adjacent to u_2 and u_4, and hence have the same candidate set. PEREGRINE computes candidate sets per neighborhood group instead of per non-core vertex to avoid performing duplicate operations for each member of a neighborhood group.

3.2.2 Reducing Match Enumeration Depth

We make an important observation about the vertices within a neighborhood group. Namely, the vertices in a neighborhood group are symmetric to each other. We exploit these symmetries to efficiently enumerate matches instead of the traditional DFS enumeration process that iteratively maps the candidates for each non-core vertex and backtracks.

In unlabeled patterns, the partial ordering of the pattern restricted to a neighborhood group is a total ordering. For example, the neighborhood group $\{u_1, u_3\}$ is totally ordered due to the condition $u_1 < u_3$. In labeled patterns, each subset of the neighborhood group vertices with the same labels will be totally ordered. For ease of explanation, we use the term *match group* to refer to a totally ordered subset of a neighborhood group in labeled patterns, and an entire neighborhood group in unlabeled patterns.

This allows us to break up enumeration into several mostly independent stages. As the elements of a match group are ordered and there are no orderings between match groups, we can map the elements of an individual match group quickly in tight loops with few branches. Importantly, vertices in separate neighborhood groups are not symmetric, so there are no checks for partial orders across groups. The only dependence outside a match group is to avoid re-mapping data vertices that have already been matched in m. We proceed to map one match group at a time in depth-first manner. By using match groups, the depth of exploration is reduced from the number of non-core vertices to the number of match groups.

3.2.3 Performance Results

We measure the performance benefits achieved by neighborhood groups. We run pattern matching queries with patterns p_1 and p_2 from Figure 4, because in both patterns all the non-core vertices are organized in a single neighborhood group. Patterns without multiple vertices in a neighborhood group remain unaffected. We run the queries on the Patents [17] and YouTube [8] graphs. Patents is a sparse graph with 2.7M vertices and 13M edges where each edge represents a citation between US patents. YouTube has 6.9M vertices and 44M edges, where each vertex represents a video and edges link related videos together. The experiments are performed on an Intel Xeon Gold 3.10GHz processor, using 16 physical cores with hyperthreading enabled and 32GB RAM.

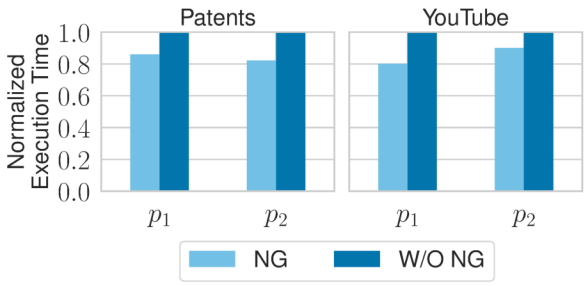

Figure 5: Execution times for pattern matching queries with neighborhood groups (NG) and without neighborhood groups (W/O NG). All times are normalized w.r.t. W/O NG.

Figure 5 shows the execution time for PEREGRINE with and without neighborhood groups. We observe that PEREGRINE achieves 11-25% better performance when using neighborhood groups. As expected, with neighborhood groups enabled PEREGRINE performs 1.5-3× fewer set intersections to match the input patterns. This translates to huge savings: for instance, PEREGRINE performs 152M fewer intersections when matching p_1 on YouTube when using neighborhood groups.

3.3 Match Groups & Fast Paths

With match groups enabled in PEREGRINE, we taxonomize patterns into different classes based on the number of match groups they contain. By doing so, fast paths can be developed for different classes to skip certain depth-first exploration steps. PEREGRINE currently incorporates two fast paths, one for the common case of a single match group, and another for the common case of two match groups. From our example patterns, p_a follows the former fast path, and p_b follows the latter.

3.3.1 Single Match Group

When there is a single match group containing k vertices, there is no need for any further depth-first exploration. It remains only to enumerate all unique k-tuples from a single vector. We can also skip checking whether vertices in m are present in the candidate set for the match group, since all core vertices must be adjacent to the members of the match group, otherwise there would be more than one.

When the graph mining use case only requires the number of pattern instances, the count can be computed in constant time as $\binom{|A|}{k}$ where A is the candidate set for the match group.

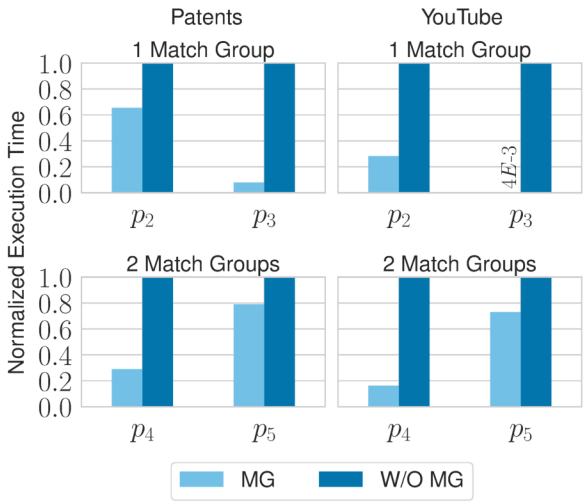

Figure 6: Execution time for pattern matching queries with match group fast paths (MG) and without match group fast paths (W/O MG). All times are normalized w.r.t. W/O MG.

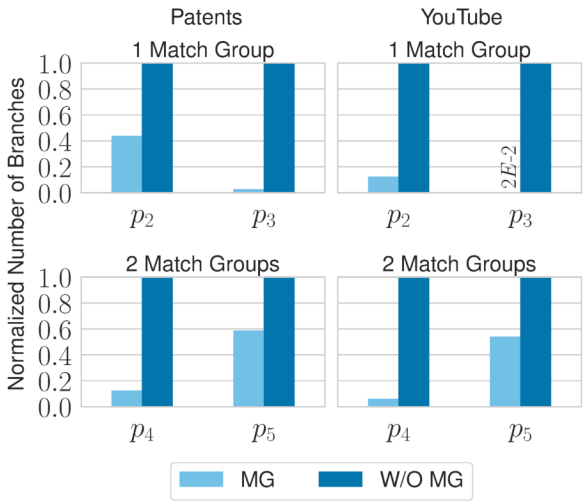

Figure 7: Number of branches for pattern matching queries with match group fast paths (MG) and without match group fast paths (W/O MG). All numbers are normalized w.r.t. W/O MG.

3.3.2 Two Match Groups

When there are two match groups, enumeration requires a Cartesian product of the sets of unique tuples representing matches for vertices in each group, subtracting any overlap. For example, consider p_b. Each of its non-core vertices represents a separate match group. Suppose u_4 and u_5 have candidate sets A and B, respectively. Then the matches for the non-core vertices are precisely the pairs:

$$A \times B \setminus \{(v,v) : v \in A \cap B\}$$

Even though this requires an additional set intersection and an additional set difference to account for overlaps, directly computing this set is much faster than a general depth-first exploration.

To count the matches instead of enumerating, PEREGRINE simply computes the cardinality of the set directly. For example, the cardinality of the above set is simply $|A| \cdot |B| - |A \cap B|$.

3.3.3 Three+ Match Groups

The approach for two match groups generalizes to larger numbers of match groups as well. However, the number of additional set operations required to remove overlaps grows combinatorially with the number of match groups. In a pattern with k match groups, it requires $\sum_{i=2}^{k} \binom{k}{i}$ additional set intersections and just as many set differences. This leads to diminishing returns after two match groups, and therefore when there are three or more match groups PEREGRINE simply traverses them in depth-first fashion as described previously.

3.3.4 Performance Results

To measure the impact of the fast paths for the two pattern types, we run pattern matching and 4-motif counting with and without the fast paths enabled. For the pattern matching queries we use two patterns p_2, p_3 which have a single match group, and two patterns p_4, p_5 with two match groups. Figure 6 shows the execution times. For a single match group, the fast path leads to a 1.5-236× speedup for p_2 and p_3, while the fast path for two match groups leads to a $1.25-6\times$ speedup for p_4 and p_5. On YouTube, the two match group fast path improves performance for p_4 by 6× and p_5 by 1.37× despite requiring 50M and 4B more set intersections for p_4 and p_5 respectively.

We also observe performance benefits for 4-motif counting, where 4 out of 6 patterns benefit from fast paths. Overall, 4-motif counting speeds up by $1.6-2.7\times$.

We profile the executions using `perf` to measure the reduction in branches due to the fast paths. Figure 7 shows the results. On the YouTube graph, all of p_2, p_3, p_4, p_5 incur on average 175× fewer branches during matching with the fast paths enabled, culminating in 34× fewer branch misses on average. Even though the 4-motif counting query contains patterns which do not benefit from the optimizations, it still performs $2-4.8\times$ fewer branches and 2.9B fewer mispredictions.

3.4 Scalability

PEREGRINE performs explorations in vertex-parallel fashion. Graph mining use cases are usually embarrassingly parallel, and PEREGRINE utilizes dynamic load balancing techniques

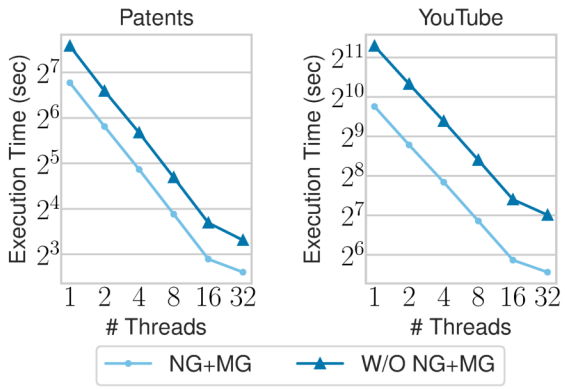

Figure 8: Execution time (in seconds) for 4-motif counting with the presented matching techniques (NG+MG) and without them (W/O NG+MG), across different number of threads.

to reduce workload imbalance (summarized in §4). Hence, PEREGRINE scales linearly with the number of physical cores.

To understand how our advanced matching strategies impact the scalability of PEREGRINE, we study scalability with and without neighborhood groups and match group fast paths. We run 4-motif counting on the Patents and YouTube graphs with varying thread counts between 1 and 32. For each thread count we run PEREGRINE with and without the neighborhood groups and match group techniques, to measure how they impact scalability. Figure 8 shows the results.

PEREGRINE scales well due to its highly parallel processing model and load-balancing strategies. At 16 threads, execution is roughly 15× faster than with 1 thread. We observe a gradual scaling from 16 to 32 threads mainly due to hyperthreading kicking in after the physical core count is exceeded. Graph mining involves irregular memory access patterns that often miss the L3 cache, which is a suboptimal workload for hyperthreading. While one hyperthread waits for main memory access, the processor schedules another hyperthread, which often must also wait for main memory access.

The neighborhood group and match group techniques predictably improve the performance while retaining high scalability. We observe a 1.6× speedup over the unoptimized execution on Patents and a 2.7× speedup on YouTube at 32 threads. Even though some of the 4-motifs patterns do not benefit significantly from these techniques, the ones that do are relatively expensive to match and hence end up impacting the overall performance. The speedups resulting from our techniques are consistent at all thread counts: on Patents the optimizations lead to $1.6-1.75\times$ speedups while on YouTube they lead to $2.7-2.9\times$ speedups.

4 More Parallelism & Workload Management

There are several other techniques in PEREGRINE that enable it to deliver high performance, especially across different use cases. Below we summarize some of the key techniques.

Vectorized Candidate Set Computation. PEREGRINE computes the candidate sets for pattern vertices using the C++ STL `set_intersection` and `set_difference` functions. When matching labeled patterns, the candidate sets are also pruned of data vertices whose labels do not match. On platforms with strong vectorization support, C++17 execution policies can be used to easily vectorize these operations.

Dynamic Load Balancing. Graph mining algorithms often suffer from workload imbalance due to dense regions of the graph containing many more pattern matches than sparse ones. PEREGRINE assigns workers to explore beginning from each data vertex, and combats workload imbalance by reordering the data vertices based on their degrees and pruning explorations that move from high-degree vertices to low-degree ones. This ensures that workers assigned to high-degree vertices are not overburdened with edges to explore.

Early Termination for Existence Queries. *Existence queries* do not require processing the entire data graph since their results can be evaluated whenever their respective conditions are satisfied. PEREGRINE terminates the exploration process during matching when the user program observes the necessary conditions. This is achieved by notifying the worker threads to stop exploration. 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.

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 Frequent Subgraph Mining where patterns that meet the support threshold can be deemed frequent while matching continues for other patterns. This is implemented using non-blocking calls from matching threads to an asynchronous aggregator thread which combines their local values.

5 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.

Arabesque [48], Fractal [12] and G-Miner [7] are distributed graph mining systems whereas RStream [52] and AutoMine [33] enable graph mining on a single machine.

None of these systems are 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. Lack of pattern-awareness not only makes these systems slower, but also limits their applicability to more complex graph mining use cases.

ASAP [22] and ApproxG [34] enable approximate pattern mining and graphlet counting. Works like GraMi [13], ScaleMine [1], DistTC [20], QFrag [43], TurboISO [19], PruneJuice [39], TurboFlux [26], PGX.D/Async [40] and others [2,4,10,18,29,36,45–47,54] develop purpose-built solutions for specific graph mining problems. OPT [25] is a fast single-machine out-of-core triangle-counting system whose techniques are generalized by DualSim [24] to match arbitrary patterns.

Finally, several works enable processing static and dynamic graphs [11,14,15,21,28,30–32,38,41,42,44,49–51,55]. 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.

6 Conclusion

PEREGRINE is a pattern-aware graph mining system that efficiently explores subgraph structures of interest and scales to complex graph mining tasks on large graphs. We presented an in-depth view of the pattern-analysis techniques powering the matching engine of PEREGRINE which enable its state-of-the-art performance. Our experiments show the importance of pattern-awareness for scalable and performant graph mining. The analysis for our advanced matching strategies takes only a couple of microseconds to compute, yet they improve the overall mining performance by up to two orders of magnitude. Details about other aspects of PEREGRINE including its comprehensive evaluation can be found in [23].

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

[8] Xu Cheng, C. Dale, and Jiangchuan Liu. Statistics and Social Network of YouTube Videos. In Hans van den Berg and Gunnar Karlsson, editors, *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 229–238. IEEE, June 2008.

[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] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[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] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.