Towards Incremental Graph Mining on Streaming Graphs

by

Rakesh Mahadasa

B.Tech., National Institute of Technology, Calicut, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Rakesh Mahadasa**
Degree:	**Master of Science**
Thesis title:	**Towards Incremental Graph Mining on Streaming Graphs**
Committee:	**Chair:** Saba Alimadadi Assistant Professor, School of Computing Science
	Keval Vora Supervisor Assistant Professor, School of Computing Science
	William (Nick) Sumner Committee Member Associate Professor, School of Computing Science
	Tianzheng Wang Examiner Assistant Professor, School of Computing Science

Abstract

Analyzing the structural properties of graphs is important in various domains including bioinformatics, malware detection, and social network analysis. The fast-changing nature of real-world graphs demands efficient solutions to analyze them in real-time. While general-purpose graph mining systems have been developed to analyze static graphs, there is no comparable solution for mining insights from dynamic graphs. Existing application-specific streaming solutions store intermediate results and explore extraneous matches that are not relevant for the final results, degrading their overall performance.

In this thesis, we present Protean, the first general-purpose graph mining system for streaming graphs. Protean incorporates two key components. First, a novel differential pattern matching engine that directly finds useful subgraphs resulting from graph updates and operates independently from the snapshot matching engine. And second, a dynamic processing model that captures cross-pattern dependencies for dynamic subgraph exploration, and utilizes an efficient multiversioning strategy to avoid exploration of automorphic matches.

For applications where the patterns of interest are known apriori, the differential pattern matching engine constructs an efficient pattern exploration plan to find matches affected by the graph updates. On the other hand, applications that dynamically determine the patterns of interest often demand exploring the data graph from scratch in order to generate results for new patterns of interest. For such applications, Protean tracks cross-pattern dependencies using a Pattern-Dependency DAG (P-DAG for short) and dynamically invokes the right matching engine (differential or snapshot) based on the impact of graph update. As the matching tasks arising from the two matching engines demand different computing power, Protean parallelizes the matching tasks from the two engines in different ways in order to maximize performance.

Our evaluation shows that Protean achieves low latency response to updates, often computing fresh results in less than a millisecond, which is crucial for continuous graph mining. As the existing purpose-built streaming graph mining solutions take minutes or even hours to process updates, Protean's update-driven parallel processing model enables orders of magnitude better performance, and scales to graphs with billions of edges.

Keywords: Streaming Graphs; Graph Mining; Pattern Matching; Pattern-Awareness

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Keval Vora for his continuous support and motivation that helped me during my research. Without his invaluable guidance, this thesis would not be possible.

I would also like to thank my friend and labmate Kasra Jamshidi, for his immense help and encouragement while working on this thesis. I am grateful to my fellow labmates at Parallel and Distributed Computing Lab for their kindness and support during the course of my time at SFU: Mugilan Mariappan, Lynus Vaz, Joanna Che, Miao Liu, Pourya Vaziri, Matthew Morrison and Nachiketa Ramesh.

Finally, I would like to thank my family and friends for their help and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As graph data becomes increasingly available in practice, graph mining applications are commonplace in many domains including bioinformatics, computer vision, malware detection, and social network analysis [35, 9, 12, 28]. Graph mining algorithms glean insights about the structure of the input graph through exploration and aggregation of interesting subgraphs. As real-world graphs are constantly changing, efficiently processing dynamic graphs to generate continuous results as the graph structure changes becomes important in order to obtain timely results for the latest graph version and to analyze how results change with graph structure updates.

Incremental processing is a useful technique that reduces the amount of recomputation by directly operating on changes (or differences) resulting from graph structure updates. While incremental processing has been shown to deliver high performance for dynamic graph processing systems [23, 32], it requires maintaining intermediate state to be able to directly process the difference in values resulting from graph updates. However, since the exploration space in graph mining workloads is often exponential, they generate a large amount of intermediate subgraphs that cannot be easily held in memory. For instance, a recent work [18] shows how systems that maintain intermediate subgraphs (either in memory or on disk) end up crashing due to lack of memory/storage capacity, or do not finish executing in a reasonable amount of time due to large number of disk read-write operations. Furthermore, the size of result sets generated by graph mining workloads itself is massive. For instance, it is common for the final number of matching subgraphs (i.e., subgraphs of interest, also called *matches*) to be in the range of trillions even for graphs comprising just a few million edges. Such high memory requirements make traditional incremental processing techniques infeasible for graph mining workloads.

Thankfully, the subtasks involved in graph mining workloads are 'local', i.e., matches originate from their corresponding subset of vertices and edges, and hence, matches from a given region of the graph remain unaffected from updates to other regions of the graph. This is fundamentally different from graph processing workloads (like PageRank) where the subtasks are 'global' and require frequent synchronization of intermediate results throughout execution (e.g., Bulk Synchronous Parallel [30] processing). Hence, graph mining can be performed on dynamic graphs without saving the intermediate subgraphs, such that the results are incrementally adjusted by exploring only the affected region in the graph (i.e., where updates take place) instead of restarting the entire mining process from scratch.

1.1 Challenges

Developing a general-purpose dynamic graph mining system is not straightforward due to the complexities involved in incremental exploration within the affected graph region, as well as the changing run-time requirements of graph mining applications which guide subgraph exploration based on previous results.

1.1.1 Incremental Subgraph Exploration

Naïvely invoking graph mining tasks on the affected graph region causes redundant computation which may not contribute to the final result. We illustrate this using a strawman approach, described next.

When updates are applied to the graph, an abstract subgraph can be constructed around the updates to bound the results that must be recomputed [13]. The abstract subgraph is based on the specific graph mining application; for instance, if we are counting 4-sized motifs [1], then the abstract subgraph should contain all vertices within a 3-hop distance of the endpoints of the updated edges, along with the edges between those vertices. Then matches can be explored in the abstract subgraph using any static graph mining exploration strategy. The abstract subgraph is searched for matches which contain the new graph updates as well as those which ignore them; the latter is required to identify matches that were once interesting, but no longer due to new updates. The matches are then checked to decide whether they impact the results based on the graph update. This involves checking whether the explored match contains the newly added edge, whether it contains the endpoints of the edge that was removed, etc. Afterwards, the verified matches can be used to adjust the final aggregation result. E.g. in motif counting, new matches increment the corresponding motif counts whereas old matches that are no longer valid decrement their motif counts.

This approach provides accurate results without restarting the mining process from scratch, but it performs significantly more computation than required to adjust results based on graph updates. While the abstract subgraph limits the amount of recomputation, it contains many subgraphs that are not useful and need not be explored because they do not contain the updated edges or their endpoints. Furthermore, the amount of unnecessary recomputation depends on where the update is applied (e.g., in dense vs. sparse regions of the graph), and the size of the graph mining query. For instance, to count 3-motifs in the

[1] A motif or pattern is an abstract description of the structure of a subgraph to be explored.

Patents graph [17], this strategy explores 1.8 million matches when 10 edge updates are applied, out of which only 540 matches (0.03%) are useful; whereas during 4-motif counting, it explores 220 million matches of which only 4,488 (0.002%) are useful. Hence, it becomes crucial to develop an incremental subgraph exploration strategy that aggressively eliminates unnecessary explorations and does not rely on maintaining intermediate subgraphs.

1.1.2 Dynamic Subgraph Exploration

The set of subgraphs to be explored in applications like motif counting and pattern matching does not change during execution, and is known apriori. For such cases, incremental subgraph exploration can be used directly to compute final results.

However, applications like Frequent Subgraph Mining (FSM) dynamically determine the set of subgraphs to explore as execution progresses. Specifically, the exploration process in FSM progresses iteratively such that only patterns that are frequent are extended into larger patterns, which are then explored in the next iteration. In this case, graph updates can affect previous decisions made regarding which subgraphs should be explored in each iteration. For instance, newly added edges can cause a pattern that was initially infrequent to become frequent: in this case, extensions of this new frequent pattern that were never explored before must now be explored. Directly exploring the subgraphs in an incremental fashion (for instance using the abstract subgraph method described above) would lead to incorrect results mainly because those patterns were never explored before, and the results of an incremental exploration will have no basis to merge with in order to deliver the final results.

A simple approach is to compute results for all possible patterns up front regardless of whether they contribute to the final results for frequent patterns, thus enabling incremental subgraph exploration whenever the graph structure updates. However, this requires exploring all possible subgraphs of the data graph (hence why FSM prunes explorations based on frequencies). Such high computation costs are only justified if the results for every pattern will be useful at some point upon graph update.

Thus a smarter strategy is required that tracks the patterns that have been explored so far, and explores new patterns in the entire graph instead of only the updated region.

1.2 Overview of Protean

We propose Protean, a general-purpose graph mining system for streaming graphs that generates timely results with continuous graph updates. Protean maintains only the aggregation values for graph mining applications (i.e., does not track the intermediate matches), and incrementally adjusts the graph mining results without exploring unnecessary subgraphs. It does so using: (a) a novel *differential pattern matching engine* that directly explores useful matches resulting from graph updates; (b) a snapshot pattern matching engine that explores

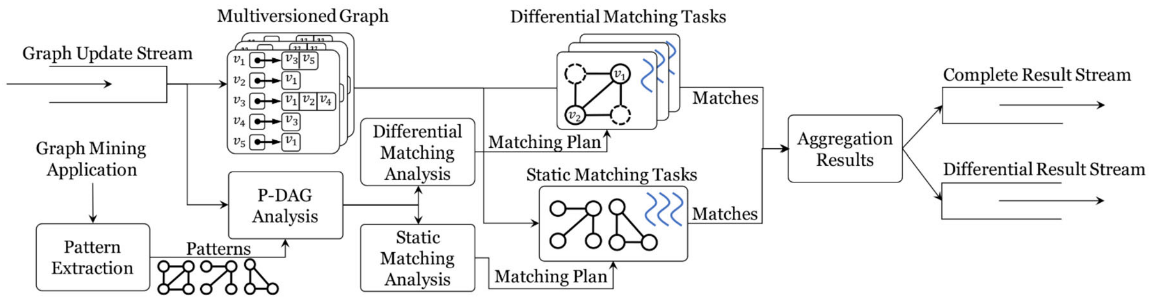


Figure 1.1: Protean system overview.

matches in the entire graph; and, (c) a dynamic processing model that guides the exploration of subgraphs using the two engines depending on the graph update. Figure 1.1 shows an overview of Protean's design.

The differential pattern matching engine recognizes the graph structure updates as first class operations over graphs. It constructs efficient pattern exploration plans that directly find matches containing the updated graph elements so that all the matches are directly useful to update the final results. The engine generates results in form of differences; specifically, it explores new matches that contribute to the final result as well as old matches that must be discounted from the previous result. Aggregations that form abelian groups are incrementally adjusted by adding the contributions for new matches and inverting the contributions for old matches that become invalid due to the graph updates.

The snapshot pattern matching engine remains oblivious to the dynamic nature of the graph and explores matches in the entire graph snapshot. While any pattern matching engine can be used for snapshot matching, we develop our snapshot pattern matching engine based on the matching engine developed in Peregrine [18] because it is the state-of-the-art and it constructs efficient pattern exploration plans to find matches throughout the graph.

The two matching engines complement each other in order to maintain up to date results for the graph mining application as graph structure gets updated. Protean's dynamic processing model maintains the information regarding how pattern explorations took place prior to a graph update, along with the aggregation results corresponding to patterns that have been explored in past. It then computes the impact of the graph update on the resulting pattern set, and invokes either differential pattern matching task or snapshot pattern matching task for each pattern to be explored.

The processing model parallelizes the matching tasks from two engines in different ways in order to maximize performance. Since snapshot matching explores the entire graph, it is done in parallel using multiple threads. On the other hand, differential matching for a graph update is quick and hence, differential matching on different updates are parallelized across different threads. Invoking multiple concurrent differential matching tasks on different updates can lead to automorphisms [16] (i.e., different instances of the same match). To avoid this, Protean utilizes a *multiversioning* strategy that orders graph updates so that they are correctly visible to the concurrent matching tasks. By doing so, each differential pattern matching task operates on its own version of the graph and never explores any automorphisms.

Protean exposes an intuitive programming model that allows expressing graph mining applications as static pattern programs, without worrying about the streaming nature of the underlying graph. To maintain flexibility, it outputs streams of differential results as well as complete results, and operates in synchronous mode as well as asynchronous mode.

1.3 Results

We evaluated Protean using different graph mining applications and compared the performance with existing solutions for each of the applications. Our results show that Protean achieves low latency response to updates, where the median latencies are less than a millisecond to process single update, which is crucial for continuous graph mining. Protean's update-driven parallel processing model enables orders of magnitude better performance than existing purpose-built streaming graph mining solutions, and scales to graphs with billions of edges.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 provides the necessary background on graph mining and streaming graph model, and Chapter 3 discusses the related work. Chapter 4 shows how mining tasks can be expressed in Protean as pattern programs. Chapter 5 describes how the differential pattern matching engine and the static matching engine generate exploration plans. Chapter 6 explains Protean's processing model including: (a) how it uses multiversioning to avoid exploration of automorphisms; (b) how it maintains the execution history to invoke differential and static pattern matching tasks; and, (c) how it maintains the aggregation results corresponding to explored patterns. Finally, Chapter 7 presents our experimental results and Chapter 8 concludes the thesis.

Chapter 2

Background

We first define the terminology used in this thesis and review graph mining fundamentals. Then, we introduce the streaming model of graph mining.

2.1 Graph Mining Terminology

We denote the vertex set, edge set, and vertex label set of a graph g by $V(g)$, $E(g)$, and $L(g)$, respectively. If $L(g)$ is empty, we say g is unlabeled. A graph s is a *subgraph* of g if it contains a subset of edges in g and their endpoints, with the same labels as in g. Though the techniques described in this thesis apply to directed and multigraphs as well, for ease of explanation we assume graphs are simple and undirected.

Graph mining involves finding interesting subgraphs of an input *data graph*. These interesting subgraphs are represented by connected *pattern graphs*. A *match* for a pattern p in a data graph G is a subgraph m of G which is *isomorphic* to p. This means there is a bijection between $V(p)$ and $V(m)$ that maps adjacent vertices in p to adjacent vertices in m with the same labels. We say m is *vertex-induced* if all edges in G between the vertices of $V(m)$ are present in $E(m)$. Otherwise, it is *edge-induced*.

Pattern graphs can additionally contain *anti-edges* and *anti-vertices* [18]. These enforce the absence of edges or vertices in matches for a pattern, respectively.

Symmetries in pattern graphs can lead to *automorphic* matches. Two matches M_1 and M_2 of a given pattern are automorphic if $V(M_1) = V(M_2)$. For efficiency and easier program expression, general-purpose graph mining systems only return unique matches, i.e. one representative match from each set of automorphic matches.

2.2 Graph Mining Problems

We describe the common graph mining problems below. These applications are modelled as pattern-matching tasks which generate sets of interesting subgraphs and compute *aggregations* on them. The results of an aggregation form a mapping from an arbitrary *aggregation key*

Motif Size	Number of Patterns
1	1
2	1
3	2
4	6
5	21
6	112
7	853
8	11117
9	261080

Table 2.1: Number of patterns based on their size.

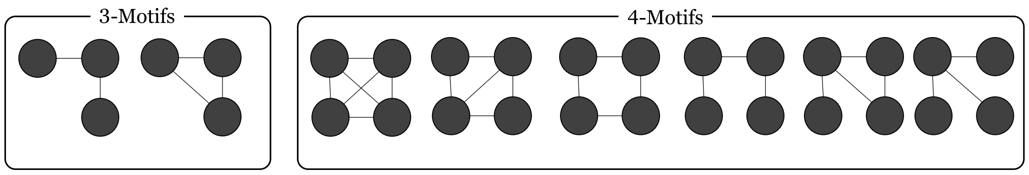


Figure 2.1: Patterns for 3-motifs and 4-motifs.

to an *aggregation value*. The aggregations are typically computed using commutative and associative operators so that aggregation results can be accumulated in parallel.

While the problems listed below focus on *counting* subgraphs of interest, they are often generalized to listing (or enumerating) as well.

2.2.1 Motif Counting

A motif is any connected, unlabeled graph pattern. Given a size k, the problem of counting k-motifs involves counting the occurrences of all motifs having k vertices. This problem typically explores vertex-induced matches; for instance, Figure 2.1 shows all possible patterns in 3-motifs and 4-motifs. As shown in Table 2.1, the number of patterns increases exponentially as k increases, which in turn increases the complexity of the problem. Figure 2.2 shows the example of finding 3-motifs in a data graph. Note that wedges (connected subgraphs that contain 2 edges and 3 vertices) such as $\{u_1,u_3,u_4\}$ are not counted as they induce a triangle.

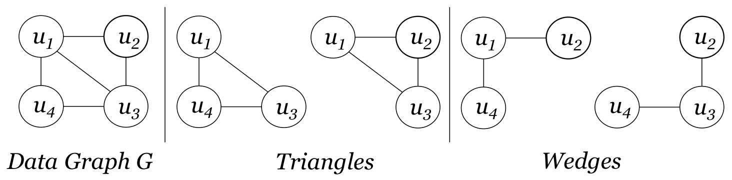


Figure 2.2: Example of finding 3-motifs in a data graph.

2.2.2 Frequent Subgraph Mining (FSM)

This problem involves finding all the labeled patterns in G, whose frequency exceeds a given threshold τ. Given a size k, k-FSM involves finding these frequent labeled patterns with k edges. FSM typically explores edge-induced matches, and measures the frequency of a pattern (also called as support) using multiple methods [20, 26, 37, 25]. Most frequent subgraph mining systems choose the *minimum node image (MNI)* [4] support measure, as this can be computed in polynomial time. On the contrary, some of the other support measures like MIS [20], HO [14] are NP-complete.

MNI support measure also satisfies *anti-monotonic property*, i.e., given two patterns p and p' such that p is a subgraph of p', MNI support of p will be at least as high as that of p'. Anti-monotonicity guarantees that if a pattern p does not meet a given frequency threshold, then all of the subgraphs that contain p are also not frequent. This allows FSM programs to prune infrequent patterns by computing frequencies in a step-by-step fashion.

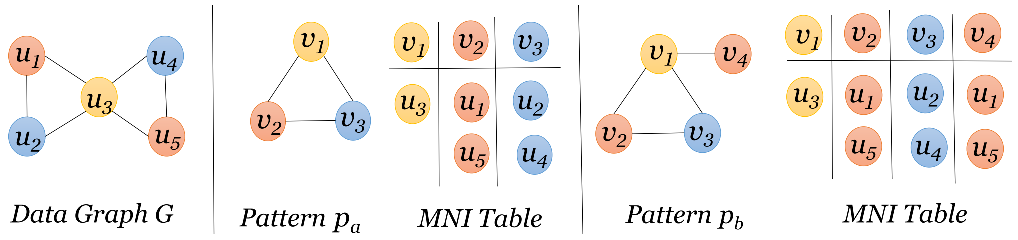


Figure 2.3: FSM example using two patterns p_a, p_b and their MNI Tables.

Using Figure 2.3, we show an example of MNI support calculation. Given a data graph G and pattern p_a, MNI Table of p_a stores all candidates of pattern vertices in columns. The number of columns in MNI table is equal to number of vertices in the pattern p_a, where each column corresponds to a unique pattern vertex. For all possible matches p_a in G (including automorphic matches), we store the set of candidate vertices for a pattern vertex in its respective column. for example there are two matches of p_a in G, the candidates for v_1 is $\{u_3\}$, v_2 is $\{u_1, u_5\}$ and v_3 is $\{u_1, u_5\}$. The support of a MNI table is the size of column that contains minimum number of candidates, which is 1 here. We constructed the MNI table to another pattern p_b in a similar fashion. Since p_a is a subgraph of p_b, the MNI support of p_a will be at least as high as the support of p_b. So, calculating the support of p_a will help in pruning all the extensions of p_a, one of them being p_b, if p_a doesn't satisfy the support threshold constraint.

2.2.3 Subgraph Counting

This problem involves counting the number of occurrences of a given pattern in the data graph. The pattern can contain structural constraints in form of missing edges or disconnected vertices. For example, the pattern can specify constraints like certain vertices cannot have any other common neighbor apart from ones specified in pattern. These constraints are specified using *anti-edges* and *anti-vertices* [18] in the pattern.

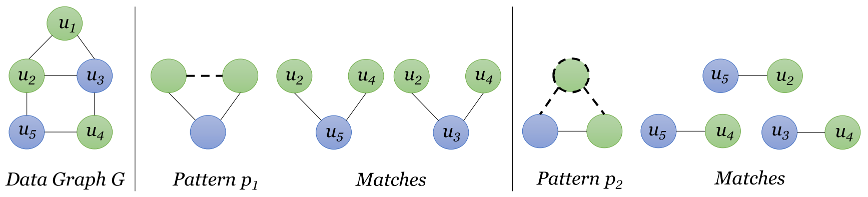


Figure 2.4: Matching patterns with anti-edge and anti-vertex constraints. Match $\{u_1,u_2,u_3\}$ is not valid for p_1 due to edge $(u_1,\ u_2)$, and matches $\{u_1,u_3\}$ and $\{u_2,u_3\}$ are not valid for p_2 due to common neighbors between the vertices.

Figure 2.4 shows how anti-edges and anti-vertices can be used to enforce structural constraints on patterns. p_1 is a wedge, whose end points are connected by an anti-edge. This anti-edge constraint eliminates the match $\{u_1,u_2,u_3\}$ due to the presence of edge $(u_1,\ u_2)$ in G. Similarly, p_2 is an edge with anti-vertex connected to both of its end points. This anti-vertex constraint invalidates matches like $\{u_1,u_3\}$,$\{u_2,u_3\}$.

2.2.4 Local Clustering Coefficient

The local clustering coefficient measures how close the neighbours of each vertex in the graph are to being a clique. This measure is used to determine the degree to which the vertices in a graph tend to form clusters. For instance, in most real-world graphs such as social networks, it is observed that the nodes form tightly knit clusters [27].

Formally, the local clustering coefficient of a vertex v is the ratio of number of triangles that contains v to the number of wedges that contains v in the middle (i.e., v is incident to both edges of the wedge).

2.3 Streaming Graph Model

Protean enables graph mining in the presence of a continuous stream of updates to the data graph's structure. An edge update is a pair $<\pm,(u,v)>$ consisting of an operation, either '+' signifying edge addition or '−' signifying edge deletion, and a target edge (u,v). Applying the edge update to a graph G means modifying the adjacency lists of u and v in G to either add or remove v and u, respectively. As updates to vertices can be easily expressed using an edge update for each neighbor of the target vertex, henceforth we consider all updates to be edge updates.

Streaming graph mining algorithms seek to maintain fresh aggregation results despite continuous updates to the underlying data graph. Updates are streamed from an arbitrary data source, grouped in batches (if necessary) according to the use case, and processed to compute fresh results. Hence, results are computed for every update batch, and depending on the application requirements, the results are output in form of *differences* due to the update batch, or as final results for the graph snapshot including the update batch.

Chapter 3

Related Work

Various works present efficient solutions to specific graph mining applications on streaming graphs. To the best of our knowledge, Protean is the first general-purpose graph mining system for streaming graphs that supports a variety of graph mining applications.

3.1 Pattern Matching

[13, 8, 15, 19, 2] develop pattern matching techniques on dynamic graphs. [2] builds a multi-versioned index of the graph based on the input pattern and adopts a relational view of the input pattern (*i.e.* treats the edges of the input patterns as set of relations). The streaming updates are mapped to each of these relations and updated matches are generated by extending or joining the edge tables. This requires storing the edges that correspond to the relations and performing isomorphism checks to match the input pattern. [13] determines the neighborhood affected by streaming edge updates and explores its subgraphs to find updated matches, hence ending up exploring unnecessary subgraphs that are not directly impacted by the streaming updates. [8] stores matches for subpatterns in an SJ-Tree and joins them based on updates to obtain matches for the input pattern. [15] is a fast pattern detection solution built on top of vertex-centric graph processing frameworks to monitor pattern occurrences in evolving graph snapshots. TurboFlux [19] is a streaming isomorphism solution that builds a data centric graph (DCG) to track candidate data vertices for each pattern vertex. As updates arrive, TurboFlux uses the DCG to determine updated matches. Both TurboFlux and [15] store intermediate results and generate unnecessary matches which must be pruned afterwards. Protean, on the other hand, does not maintain candidate subgraphs and directly explores matches affected by graph updates.

3.2 Motif Counting

Myriad research has been conducted on maintaining approximate motif counts in a streaming graph [5, 21, 36, 33]. These works use different sampling and probabilistic methods to estimate motif statistics, instead of computing exact counts. [5] provides a general unbiased estimator.

[21] and [33] propose fast approximate frequency estimation algorithms for motif counting on temporal graphs. [36] randomly samples streaming edges to estimate motif counts. In contrast, Protean allows efficiently maintaining exact motif counts using its fast differential matching engine.

3.3 Frequent Subgraph Mining

[3] estimates k-size frequent subgraphs in evolving graphs through reservoir sampling. It samples subgraphs in the neighbourhood of an edge update to estimate which patterns are frequent. IncGM+ [1] is an incremental frequent subgraph mining system on streaming graphs. It maintains MNI tables for all encountered patterns, as well as a *fringe set* of matches for patterns which are on the cusp of becoming frequent or infrequent due to an update. For each updated edge, IncGM+ finds matches containing it similarly to [13] and modifies the fringe set and MNI tables as necessary. This involves expensive subgraph isomorphism checks and intermediate results. Protean avoids per-subgraph checks and intermediate results using its pattern-aware matching engines to perform FSM exactly and efficiently.

3.4 Static Graph Mining Systems

Many systems for graph mining on static graphs have been recently proposed [29, 11, 34, 6, 24, 18, 39]. Pattern-based systems like [18] and [24] directly match subgraphs of interest represented by input patterns. Protean applies similar techniques to graph mining in the streaming setting, which carries additional challenges not present in static graph mining. Exploration based graph mining systems [29, 11, 34, 6, 39] instead iteratively extend subgraphs and prune those which are deemed uninteresting. This leads to redundant computation and large memory overheads for intermediate results, which Protean avoids by using pattern-aware matching engines.

3.5 Dynamic Graph Processing Systems

Several works propose efficient graph processing systems for dynamic graphs [23, 32, 10, 22, 31]. These systems focus on graph processing workloads that iteratively propagate values within the graph, which involve fundamentally different kind of computation compared to graph mining workloads.

Chapter 4

Programming Model

Protean enables users to easily express streaming applications as static pattern programs. The user specifies patterns and aggregations declaratively, and implements simple callbacks to control application flow without worrying about the streaming nature of the underlying graph. The system then automatically translates static computations into efficient streaming pattern matching tasks. Protean utilizes aggregations that form abelian groups and incrementally adjusts them based on differences in matches resulting due to graph updates.

Figures 4.1, 4.2, and 4.3 show sample programs for a variety of graph mining use cases. The functions marked in blue are Protean's API, and the user-defined callbacks are marked in orange.

4.1 Pattern Selection

Protean builds over the pattern API developed in [18] to allow dynamic pattern generation during execution. Figures 4.1, 4.2 and 4.3 show examples of pattern mining applications expressed on Protean. The `extendEdge()` and `extendVertex()` methods take a predicate as input and instruct the system to extend a pattern if it satisfies the predicate. This allows easy expression of pattern exploration, such as in the FSM program shown by Figure 4.2. In this example the runtime combines the predicates passed to `filter()` and `extendEdge()` to only extend frequent patterns.

4.2 Aggregation

User-defined computations in Protean begin with matches. Protean invokes a user callback function on every match. Often, this function will transform a match and map the result to an aggregation value. In Figure 4.3, `mapLCC()` maps each vertex in a match to a pair representing the proportion of triangles to triplets.

The aggregation is defined implicitly by the type of the aggregation value. Protean uses the addition operator to combine aggregation values, and applications will fail to compile if

```
DataGraph G = loadDataGraph("input.graph");
UpdateStream U = connectStream("updates.pipe");
Set<Pattern> P = allVertexInducedPatterns(SIZE);

ConcurrentQueue<Map<Pattern, Int>> *result_stream =
  Protean.count(P, G, U).liveResults();

while (Protean.running() || !result_stream->empty()) {
  // write live motif counts to disk
  // without interrupting pattern matching
  Map<Pattern, Int> counts = result_stream->dequeue();
  log(counts);
}
```

Figure 4.1: Non-blocking motif-counting program with handle to live results.

```
Void mapSupport(Match m) {
  map(m.pattern, Domain(m));
}
Bool isFrequent(Pattern p) {
  Domain d = readAggregation(p);
  return d.support() >= THRESHOLD;
}
Bool shouldExtend(Pattern p) {
  return p.numEdges() < SIZE;
}
Void displayResults(Map<Pattern, Domain> result){
  for (auto [pattern, domain] : result)
    cout << pattern << " " << domain.support() << endl;
}

DataGraph G = loadDataGraph("input.graph");
UpdateStream U = connectStream("updates.pipe", t=1s);
Set<Pattern> P = allEdgeInducedPatterns(2);

Map<Pattern, Domain> final_results =
        Protean.match(P, G, U, mapSupport)
            .filter(isFrequent)
            .extendEdge(shouldExtend)
            .forEachUpdateResult(displayResults);
```

Figure 4.2: Blocking Frequent Subgraph Mining program.

the provided aggregation value type does not have such an operator. In a dynamic setting, edge updates can cause previously found matches to disappear. This is handled using an operator which subtracts one aggregation value from another. While simple types like integers are easily added and subtracted, more complicated data types, such as MNI tables [4] used in FSM, can be modified to include a subtraction operator with little user effort. For instance, our MNI table implementation required only 9 lines of code for its subtraction operator.

4.3 Output Streams

Protean outputs streams of differential results as well as complete results. These contain the change in aggregation results due to the latest update batch, and the latest combined aggregation result, respectively.

```
DataGraph G = loadDataGraph("input.graph");
UpdateStream U = connectStream("updates.pipe", n=1000);
Set<Pattern> P = {generateClique(3), generateStar(3)};

Void mapLCC(Match m) {
  Ratio rt = (1,0);
  Ratio rw = (0,1);
  m.pattern == P[0] ?
    (map(m[0], rt), map(m[1], rt), map(m[2], rt)) :
    (map(centerVertex(m), rw));
}

Void displayResults(Map<Vertex, Ratio> result) {
  for (auto [v, ratio] : result) {
    Int triangles = ratio[0];
    Int wedges = ratio[1];
    cout<< v << ": " << triangles / wedges << endl;
  }
}

Map<Vertex, Ratio> final_results = Protean.match(P, G, U, mapLCC)
  .forEachUpdate(displayResults);
```

Figure 4.3: Blocking Local Clustering Coefficient program.

The output streams can be accessed in two ways. Protean can operate in synchronous, blocking mode using the `forEachUpdateResult()` and `forEachUpdateDelta()` methods, which accept a callback function as input. In this mode, the callback is applied to the results after every update batch, and the next update batch is not processed until the callback returns. Alternatively, invoking the `liveResults()` or `liveDeltas()` methods on an application instructs Protean to operate asynchronously in the background, while control returns to the user program. In this mode, a handle to the output stream is returned, which allows users to access the queue of complete or differential aggregation results. Figure 4.1 shows an asynchronous motif-counting program where the user can write data to disk without blocking Protean's pattern matching progress.

4.4 Granularity of Update Batch

Applications like motif counting and subgraph matching are sensitive to graph updates as their results change with every single graph update. Other applications like FSM are less sensitive to single updates as patterns often do not suddenly become frequent or infrequent based on a single graph update. Such applications may require the analysis to be triggered after multiple updates are applied to the graph. Protean can refresh results after every update (for live results) as well as after multiple updates, depending on the application needs. Updates can be grouped into fixed size batches (e.g., batch of 1000 updates shown in Figure 4.3), or based on time intervals (e.g., splitting the update stream into 1 second intervals shown in Figure 4.2).

Chapter 5

Generating Exploration Plans

Protean uses both a static engine operating on graph snapshots and a novel differential matching engine for computing changes in the matching results for an update. In this chapter, we discuss how these engines analyze patterns to generate efficient exploration plans. Section 6 will show how the processing model invokes these engines based on graph updates.

5.1 Matching on Static Graph Snapshot

Protean uses Peregrine's core pattern matching engine [18] as the snapshot matching engine. The input patterns are analyzed to construct matching plans, then the snapshot is traversed according to these plans to generate matches. We briefly explain the relevant analysis and how it enables efficient pattern matching.

First, we apply [16] to obtain a partial ordering on symmetric pattern vertices to prune automorphisms and break symmetries. This is performed as follows. Vertices of the pattern are labeled with unique ids and all automorphisms of the pattern are listed out. For a given pattern p, this involves finding all orientations of p that are isomorphic to p. Then, partial orders are built by iterating over the vertices in the order of their ids. For each pair of ids, they are added to the partial orders set if ordering them eliminates automorphisms. The partial orders are built repetitively until there are no more automorphisms left in the pattern. During the matching phase, these partial orders are enforced on the vertex ids of matches that map to the pattern.

After computing the partial orders, the *pattern core* and *matching orders* are identified according to the ordering. A pattern core is given by a minimum connected vertex cover of the pattern, e.g. the edge (u_2, u_4) in the pattern in Figure 5.1. Matching orders are total orderings of core vertices which satisfy the pattern's partial ordering. The example pattern's partial ordering permits only one matching order: $\{u_2, u_4\}$.

Finally, data vertices are iteratively mapped according to the matching orders. In the example, a data vertex v is mapped to u_2, then each neighbour of v with greater ID is mapped to u_4, yielding matches for the core. Candidates for the remaining pattern vertices

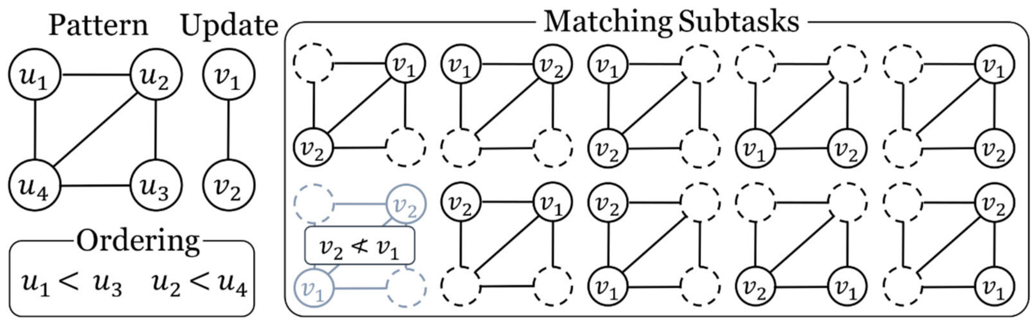


Figure 5.1: Example of matching subtasks generated by differential matching engine based on an edge update.

are obtained through set operations on the adjacency lists of the matched vertices. Hence, the vertices matched to u_2 and u_4 yield the candidates for u_1 and u_3, which produce matches for the entire pattern once mapped.

5.2 Differential Matching upon Graph Update

As the data graph's structure is updated, previously matched subgraphs may disappear as an edge is deleted, whereas new matches can arise when an edge is added. If the input pattern is vertex-induced, both events can occur in a single update. For example, an edge addition can cause a set of vertices to induce a different pattern, invalidating a match, while another set of vertices now induces the input pattern, leading to a new match. We call the new matching subgraphs created by an update *positive matches*, and the previously matching subgraphs which are invalidated are called *negative matches*.

To maintain fresh results after graph updates, our differential pattern matching engine efficiently computes both positive and negative matches. The engine analyzes the impact of graph updates on the input patterns to generate efficient exploration plans that directly match affected subgraphs, without traversing unrelated areas of the data graph.

The key insight in our differential pattern matching engine is to begin matching from the updated data edge, instead of a data vertex. This guarantees that the resulting matches will contain the updated data edge, and hence, no further checks will be required. In order to generate all matches containing the updated edge, we need to map the updated data edge to every compatible pattern edge. This yields matching subtasks which together generate exactly the subgraphs that arise due to the edge update.

5.2.1 Mapping Updated Edge to Pattern Edges

We begin with preliminary pattern analysis as in the previous section, computing the pattern core and matching orders. Each subtask is a partial match mapping the updated edge to a different pattern edge with the same labels. This mapping must also satisfy partial orders. If the endpoints of an edge are not ordered, then the updated edge can be mapped in both directions.

Consider the example pattern in Figure 5.1. Suppose (v_1, v_2) is the updated data edge, with $v_1 < v_2$. Then (v_1, v_2) can be mapped to pattern edge (u_1, u_2) in two different ways since there is no partial ordering between u_1 and u_2: $u_1 \to v_1, u_2 \to v_2$ as well as $u_2 \to v_1, u_1 \to v_2$. On the other hand, (v_1, v_2) can only be mapped to (u_2, u_4) in one way which preserves the partial ordering: $u_2 \to v_1, u_4 \to v_2$. Repeating this for every pattern edge yields 9 subtasks: each of the four unordered edges contribute two subtasks, and the single ordered edge of the pattern core contributes one.

5.2.2 Mapping Updated Edge to Anti-Edges

A recent work [18] introduces anti-edges and anti-vertices as concrete abstractions in patterns to represent absence of edges and vertices. An anti-edge enforces absence of an edge in the data graph between the pairs of vertices, whereas an anti-vertex enforces absence of a common neighbor between vertices. Our differential pattern matching engine natively handles anti-edges and anti-vertices in the pattern.

For patterns containing anti-edges, we map the updated edge to anti-edges in the same way. The matches for these subtasks are negative matches if the update adds an edge, and positive matches if the update removes an edge. This is because when we add an edge to the data graph, all the positive matches that contain this edge, mapped to anti-edge cannot satisfy anti-edge constraint any more. Similarly, all the matches that were not matched due to presence of an anti-edge will become positive matches when the edge mapped to anti-edge is deleted.

Since we do not consider anti-vertices while computing pattern-cores, the pattern-core will not contain anti-vertices. Hence, mapping the updated edge to anti-edge containing anti-vertex can lead to the case where the updated edge is mapped to pattern edge whose end points are not part of the pattern core. Since we start matching from pattern core, this would result in exhaustive matching from scratch as the exploration does not start from updated edge. To avoid this, we extend the pattern core to include the real pattern vertex of the anti-edge whenever the updated edge is mapped to a pattern edge whose end points are not part of pattern core. Note that an anti-edge can have at most one anti-vertex.

5.2.3 Revising Matching Orders for Mappings

To complete the subtasks in Figure 5.1, we must follow the matching orders of the pattern, skipping the vertices that have already been mapped. There are two cases: either the updated edge has been mapped to an edge within the pattern core, or to an edge outside of it.

(A) Updated Edge Mapped to Core Edge. Here, we simply revise the matching orders to begin with the updated edge, possibly by inverting its ordering. After the core is matched there is no change to how the remaining vertices are matched. Note that swapping vertices in a matching order does not affect correctness: in our example pattern matching u_2 then u_4 is equivalent to matching u_4 then u_2.

(B) Updated Edge Mapped to Non-Core Edge. Here, one endpoint of the updated edge must still fall within the pattern core, since it is a vertex cover. Again, we revise the matching orders. However, the endpoint of the updated edge outside of the pattern core may have other edges into the pattern core, whose existence in the matched subgraph must be checked.

For our example pattern when updated edge (v_1, v_2) is mapped to (u_1, u_2), after the pattern core has been matched we can match u_3 as normal, which gives us a mapping for all vertices. But we must ensure the edge (u_1, u_4) is satisfied by checking that v_1 is connected to the vertex matching u_4.

5.2.4 Discussion

The above subtasks in our differential matching engine efficiently compute exactly the subgraphs containing the updated edge, while avoiding automorphisms thanks to pattern-aware techniques. As they are independent of each other, they can be executed in parallel.

By computing the impact of graph structure updates at an edge-level, each differential matching task focuses on matching for a single edge update only. Hence, separate differential pattern matching tasks resulting from different edge updates can potentially impact each other, if those edge updates are within the common neighborhood. In such a case, if the corresponding differential pattern matching tasks are performed concurrently, they can end up generating duplicate matches. For instance, consider a data graph consisting of a single edge (v_1, v_2). If two edges (v_1, v_3) and (v_2, v_3) are simultaneously added to the data graph, and two differential matching tasks t_1, t_2 concurrently try to match triangles, they will both generate the same match $\{v_1, v_2, v_3\}$. t_1 may begin matching from (v_1, v_3), and t_2 from (v_2, v_3), but as both tasks can see both updates, they will both succeed in matching the same triangle. To avoid such redundant matching, Protean's processing model enforces an ordering on updates when invoking the differential matching tasks concurrently, as explained in Chapter 6.

Chapter 6

Processing Model

Protean adopts an update-driven processing model containing two types of pattern matching tasks. A *differential task* serves to compute the difference in the aggregation results caused by an update. A *snapshot task* computes the aggregation on the latest snapshot of the data graph.

6.1 Multiversioning against Automorphisms

Each differential task operates on a single graph update. With multiple updates present in a single update batch, concurrently invoking differential tasks on each update can lead to automorphisms (as discussed in Section 5.2.4). To avoid exploring automorphisms, we enforce a strict ordering on updates within the batch and develop a multiversioned graph so that the differential task for a given update m does not see updates that are ordered after m. By doing so, the differential task finds only matches that are directly impacted by m, including ones that are also affected by other updates that are ordered before m. At the same time, differential tasks corresponding to those other updates never find these same matches since m is not visible to them. In other words, a match is found only by the latest update that directly impacts that match and hence, automorphisms are never explored.

Our multiversioned graph uses the adjacency list data structure where entries within each adjacency list are ordered based on the vertex ids to enable fast list intersection and difference operations. Figure 6.1 shows how the multiversioned graph compactly holds the graph structure while retaining the ordering between individual edge updates within an update batch. Each vertex maintains a version table where the entries point to a 'version' of its adjacency list along with a token that identifies the ordering of that version. Initially, all the version tables contain only a single entry that points to the adjacency lists from the graph snapshot before any update is applied. Then, the updates within the batch are applied in order by adding entries to the respective version table. Specifically for update $<\pm, (u, v)>$, the latest adjacency list versions for vertex u and vertex v are copied to generate their new versions, and these new versions are modified (i.e., adding/removing v and u from u's and

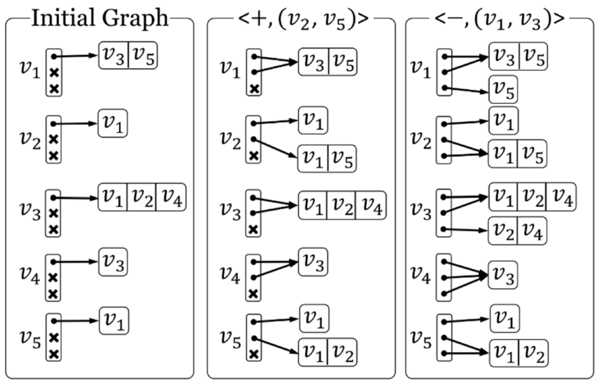


Figure 6.1: Multiversioned graph when processing two simultaneous updates. The $<+, (v_2, v_5)>$ gets ordered before $<-, (v_1, v_3)>$.

v's adjacency list). Finally, the new versions are appended to the respective table along with the order number of the update. As shown in Figure 6.1, v_4 maintains one version of its adjacency list, while the other vertices end up with 2 versions based on the updates.

When finding matches, the matching task for edge update m uses the latest versions of adjacency lists whose token values are at most the order number of m, hence ensuring unique matches. Finally after all the matching tasks for the update batch complete, the latest versions of the adjacency lists are merged back to the base version of the graph and the intermediate versions are freed up. For batches with large number of updates, the updates are micro-batched so that only a constant number of versions are maintained at a given time (to limit the memory consumption) with the latest versions in the micro-batches being merged back to the base snapshot.

6.2 Dynamic Subgraph Exploration

To support dynamic exploration of subgraphs (common across various use cases like FSM), we develop a unified processing model that tracks the dynamic cross-pattern dependencies and the aggregation mappings to compute the necessary differential and snapshot aggregations. In the remainder of this section, we explain how Protean balances computation for differential and snapshot tasks using cross-pattern dependencies.

6.2.1 P-DAG: Pattern-Dependency DAG

Protean tracks cross-pattern dependencies through a directed acyclic graph to correctly invoke differential and snapshot tasks. We call this graph of dependencies the 'Pattern-Dependency DAG' (P-DAG for short). Each node in the P-DAG represents a pattern, and the edges represent the dependencies between different patterns. The P-DAG is acyclic as applications often generate new dependent pattern tasks through extension of patterns, so there is an edge from pattern p_a to p_b in the P-DAG only if p_a is a subpattern of p_b.

```
Void runApplication(Application a, UpdateBatch U) {
  a.pDAG.markActive(a.affectedInputPatterns(U));
  a.dataGraph.applyUpdates(U);

  for (Int level in a.pDAG) {
    Set<Pattern> patterns = a.pDAG.at(level);
    parallel for (Pattern p in patterns) {
      if (a.pDAG.isActive(p) and a.pDAG.isIncrementalTask(p)) {
        for (Update e in U) {
          deltaMatch(a, e, a.dataGraph[e], p, a.matchCallback);
        }
        if (a.extensionCallback(p)) {
          if (a.filterCallback(p)) {
            a.pDAG.addDependencies(p, extend(p));
          }
          a.pDAG.markActive(a.pDAG.dependents(p));
        }
      }
    }

    for (Pattern p in patterns) {
      if (a.pDAG.isActive(p) and a.pDAG.isSnapshotTask(p)) {
        parallelSnapshotMatch(a, a.dataGraph, p, a.matchCallback);
        if (a.extensionCallback(p)) {
          if (a.filterCallback(p)) {
            a.pDAG.addDependencies(p, extend(p));
          }
          a.pDAG.markActive(a.pDAG.dependents(p));
        }
      }
    }
  }

  for (key, val in a.completeResults) {
    if (a.filterCallback(key)) {
      a.resultStream.enqueue(key, val);
    }
  }
}
```

Figure 6.2: Protean's Processing Model.

As shown in Figure 6.2, the P-DAG is traversed level by level starting with the smallest patterns. At each level, the patterns that are affected by a given update are explored using matching tasks. Unlabeled patterns are affected by all updates, but labeled patterns are only affected if they contain an edge with the same labels as the updated edge. After a pattern task is completed, edges to newly generated patterns are added to the P-DAG if the pattern passes the application extension and filter callbacks. These new patterns may have multiple incoming dependencies, as different patterns can extend to the same pattern; nevertheless, the level-by-level traversal of P-DAG ensures that only a single matching task for the affected pattern is generated to avoid redundant tasks due to different dependency edges.

Patterns are explored using either the differential task or the snapshot task, depending on whether their aggregation results are available from previous explorations.

Sample FSM Execution:

In FSM, the initial pattern set consists of edges, and the application contains an edge-extension clause. Hence, the P-DAG will be traversed starting at the pattern with the same labels as the updated edge. If the explored pattern is frequent (identified using the filter callback) and smaller than the maximum size (identified using the extension callback), its dependent pattern tasks are invoked to explore results on the extensions. If the dependent pattern nodes do not exist in the P-DAG, they are first added to the P-DAG to reflect the new cross-pattern dependency, and are then explored in the next level using snapshot tasks. For dependent patterns that are already present in P-DAG, exploration happens using differential tasks. This continues until every level of the P-DAG is traversed.

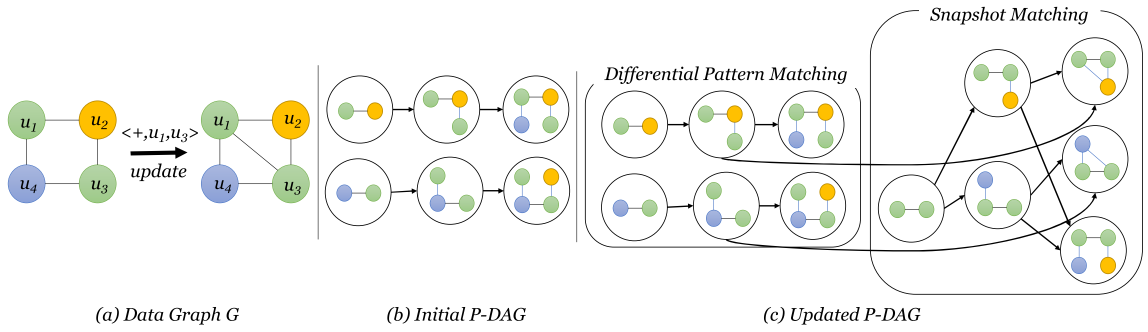


Figure 6.3: P-DAG example.

Figure 6.3 shows a 3-FSM example where P-DAG changes based on the edge update. For the initial data graph, P-DAG only contains six patterns shown in Figure 6.3b. When edge (u_1, u_3) gets added, new patterns are discovered that result into new pattern nodes in P-DAG (shown in Figure 6.3c). These new pattern nodes are matched using snapshot matching while the old ones are matched using differential pattern matching.

6.2.2 Parallelization Strategies

The computational requirements of differential tasks are significantly lower than the snapshot tasks since the former only explores matches resulting from an update. We observe that the differential matching tasks often finish in just 1-10ms whereas the snapshot matching tasks take seconds to generate results for the entire snapshot. To reduce the latencies of matching tasks, Protean utilizes different parallelization strategies for the two types of tasks.

Specifically, Protean invokes multiple differential tasks at the same P-DAG level in parallel. The snapshot tasks, on the other hand, are invoked serially one after the other and each individual snapshot task is executed in vertex-parallel manner, hence dedicating all available cores to a single snapshot task at a time. This difference in parallelization strategy is observed in Figure 6.2: the differential tasks are invoked inside a `parallel for` loop whereas snapshot tasks are invoked in a sequential `for` loop.

6.2.3 Aggregation Maintenance

Protean maintains the set of aggregation results for each pattern task in the P-DAG so that differential tasks incrementally adjust the aggregation results. The commutativity, associativity and invertibility properties of the aggregation are used to incrementally account for positive matches and discount the negative matches. Even though exploration of certain patterns within the P-DAG can become unnecessary over time (e.g. a pattern becoming infrequent deeming its extensions unnecessary), the differential tasks are invoked to keep the aggregation results up to date with the incoming updates in order to avoid expensive snapshot tasks in future.

Garbage Collection:

As new patterns are added to P-DAG over time, the aggregation results corresponding to those new patterns need to be maintained in memory as well. To make room for such new aggregation results, we develop a lightweight garbage collector that carefully frees up aggregation results that are no longer needed by the application based on the current state of the data graph. The candidates for freeing are identified using: (a) aggregation results that fail the application filter; and, (b) patterns in P-DAG that are no longer necessary to be explored. Once the aggregation results are freed up, the corresponding pattern nodes are removed from the P-DAG so that exploration of those patterns for future updates happens using snapshot matching tasks. To maximize the availability of aggregation results for future updates, the garbage collector is invoked only when the available memory capacity becomes insufficient to hold the aggregation results for the new patterns in P-DAG.

Chapter 7

Evaluation

We evaluate the performance of Protean on common graph mining benchmarks, and compare the results with state-of-the-art purpose-built streaming systems: IncGM+ [1] for Frequent Subgraph Mining and TurboFlux [19] for pattern matching.

7.1 Experimental Setup

7.1.1 System

We conducted all experiments on a `c5.9xlarge` Amazon EC2 instance which is equipped with an Intel Xeon Platinum 8124M CPU, comprising 18 physical cores (36 logical cores) and 68GB RAM.

7.1.2 Datasets

Table 7.1 lists the data graphs used in our evaluation. Patents (PA) is a patent citation graph where each patent is labeled with its grant year. Orkut (OK) and Friendster (FR) are unlabeled social network graphs where edges represent friendships between users. Youtube contains videos posted from February 2007 to July 2008 where edges represent relationship between videos and labels are generated by combining the video's rating and length.

The graph updates that are streamed in are an even mix of edge additions and deletions. In each experiment, we loaded 75% of the edges, and the remaining edges were treated as

G	$\|V(G)\|$	$\|E(G)\|$	$\|L(G)\|$	Max. Deg.	Avg. Deg.
(PA) Patents [17]	2.7M	13M	37	789	10
(YT) Youtube [7]	7.8M	44.5M	80	4039	11
(OK) Orkut [38]	3M	117M	—	33133	76
(FR) Friendster [38]	65M	1.8B	—	5214	55

Table 7.1: Real-world graphs used in evaluation.
'—' indicates unlabeled graph.

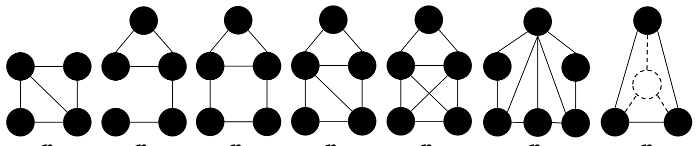


Figure 7.1: Patterns used in evaluation.

edge additions that were streamed in. Edges to be deleted were selected from the loaded graph.

7.1.3 Applications

We evaluated Protean on a wide array of applications: counting motifs with 3, 4, and 5 vertices; Frequent Subgraph Mining (FSM) with patterns of 3 and 4 edges on labeled datasets using various supports; and matching the patterns shown in Figure 7.1.

7.1.4 Baseline Implementation

To evaluate the efficacy of our differential pattern matching engine for different graph mining applications, we implemented a parallel baseline using the straw-man differential matcher described in Section 1. The baseline primarily replaces the differential pattern matching engine in Protean with the bounded region-based subgraph exploration; we call this baseline system **BRBL**.

While matching a pattern with n vertices, BRBL performs DFS to gather the vertices within an $(n-1)$-hop neighborhood of each updated edge (i.e. identifying the abstract subgraph that contains potential matches affected by the edge update). Then, BRBL performs static pattern matching in the abstract subgraph using the static pattern matching engine from Protean. Since BRBL ends up finding matches from the abstract subgraph that do not contain the updated edge as well, it simply discards those matches to maintain correct results corresponding to the updated edge.

7.2 Performance

To study the effectiveness of Protean in generating continuous results from streaming graph, we measure the latency to process a single update, i.e. time taken to generate results for the specific graph mining applications based on the single update. Later in Section 7.3, we will measure the latency when multiple graph updates are applied simultaneously to be processed in batches.

7.2.1 Pattern Matching

Figure 7.2 shows the median and 99th percentile single update latencies for pattern matching executions on Protean. As we can see, Protean is efficient in generating results based on the

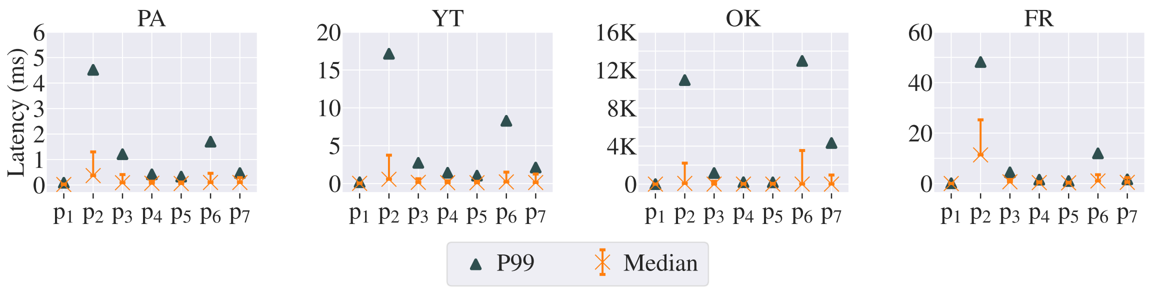


Figure 7.2: Pattern matching latencies for Protean.

App	G	Protean	BRBL	TurboFlux
Match p_1	PA	**0.02**	5.66	0.27
	YT	**0.03**	23.70	0.27
	OK	**0.05**	122126	92.96
	FR	**0.03**	×	×
Match p_2	PA	**0.37**	1522.83	1.91
	YT	**0.57**	51917.70	2.93
	OK	**75.69**	×	×
	FR	**11.36**	×	×
Match p_3	PA	**0.09**	587.60	3.63
	YT	**0.16**	9165.75	3.56
	OK	**2.46**	×	173544.35
	FR	**0.70**	×	×
Match p_4	PA	**0.07**	90.78	0.47
	YT	**0.12**	1960.53	0.70
	OK	**0.88**	×	3359.90
	FR	**0.38**	×	×
Match p_5	PA	**0.06**	76.32	0.49
	YT	**0.11**	1263.99	0.73
	OK	**0.60**	×	2182.98
	FR	**0.30**	×	×
Match p_6	PA	**0.10**	11043.80	1.82
	YT	**0.22**	253855	3.87
	OK	**5.44**	×	×
	FR	**1.04**	×	×

Table 7.2: Pattern matching latencies (in milli-seconds) for Protean, TurboFlux [19] and BRBL.
'×' indicates the execution did not finish within 5 minutes.

update; the median latencies are less than a millisecond for the smaller Patents graph, and they rise to a tens of milliseconds for the Orkut graph which is significantly denser. The 99th percentile latencies are closer to the median for most patterns that are relatively easy to explore. However, Protean does experience visible variation in latencies for expensive patterns such as p_2 and p_6. While p_2 has few edges, its sparseness means there are many matches for it in dense regions of the graph. On the other hand, p_6 is simply a very large pattern with only one partial ordering, which ends up generating 17 relatively difficult subtasks in our differential pattern matching engine. As patterns become expensive, the differences coming from where each update is applied (e.g., dense region or sparse region of the graph) affect the final latencies since the amount of work to be performed is non-trivially impacted by differences from dense regions.

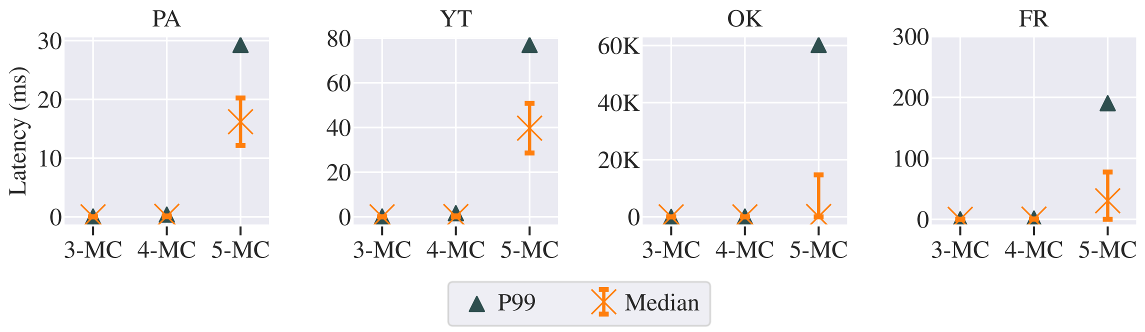


Figure 7.3: motif counting latencies for Protean.

App	G	Protean	BRBL
3M	PA	**0.02**	1.08
	YT	**0.02**	3.55
	OK	**0.02**	5949.60
	FR	**0.02**	101698.3
4M	PA	**0.10**	95.03
	YT	**0.13**	11440.45
	OK	**0.81**	×
	FR	**0.44**	×
5M	PA	**16.20**	×
	YT	**39.70**	×
	OK	**111.70**	×
	FR	**30.41**	×

Table 7.3: Motif counting latencies (in milli-seconds) for Protean and BRBL. '×' indicates the execution did not finish within 5 minutes.

Table 7.2 compares the median latencies with the BRBL baseline and TurboFlux [19]. Protean outperforms TurboFlux by 5-70146×, and BRBL by up to six orders of magnitude. BRBL suffers severe performance degradation when operating on dense graphs, as it naïvely explores the neighborhood of updates and filters unnecessary matches. Similarly, TurboFlux does not scale well on Orkut graph since its data-centric approach requires maintaining expensive intermediate results. Protean's differential matching engine achieves superior performance as it matches only the subgraphs affected by an update, without unnecessary exploration or intermediate results.

7.2.2 Motif Counting

Figure 7.3 shows the median and 99th percentile single update latencies for 3-, 4-, and 5-motif counting on Protean. The differential tasks handle few small patterns in counting 3-motifs and 4-motifs very easily, often in less than a millisecond. Latencies for counting 5-motifs are relatively higher because there are 21 pattern matching tasks, each of which generates many differential matching subtasks based on the edge mappings. Furthermore, size 5 patterns have more complex topologies and lead to more matches, and hence are more expensive.

Table 7.3 compares median latencies with the BRBL baseline. Protean is at least an order of magnitude faster than BRBL with up to five orders of magnitude faster on cases where BRBL finishes running in a reasonable amount of time. BRBL failed to process updates in reasonable time as the size of patterns increases, since the abstract subgraph to be explored enlarges, especially when the edge update is near high-degree vertices. Protean fares better in this regard because its differential engine performs the same number of subtasks per update per pattern, regardless of the edge update's neighborhood. For example, BRBL suffered a 3222× increase to latency between counting 3-motifs and 4-motifs on the YouTube graph, whereas Protean's performance differed by only 11ms.

7.2.3 Frequent Subgraph Mining

Figure 7.4 shows single update latencies for 3- and 4-FSM by Protean on the Patents and YouTube graphs across different support thresholds. Note that lower support thresholds increase the number of frequent patterns and thus increase the workload. To evaluate the effectiveness of the differential matching engine, the graph updates in this experiment were applied such that they do not lead to any static matching tasks. Later we perform another experiment to show the results for FSM when snapshot tasks are invoked.

On the Patents graph, 3-FSM and 4-FSM have similar latencies because there are few frequent patterns with 4 edges. Furthermore, 4-edge patterns are not much more difficult to match due to the sparseness of the Patents graph and the selective power of its labels. On the larger YouTube graph however, increasing pattern size does lead to significant increases in latency. YouTube also has denser regions, which lead to more variations in performance.

Table 7.4 compares median FSM latencies to BRBL and IncGM+ [1]. Protean is up to 171.6× and 223.8× faster than BRBL and IncGM+, respectively, on workloads they are able to execute. IncGM+ was unable to process any updates to YouTube graph within 3 hours, or updates to PA with support 20K, due to its pattern-unaware exploration approach

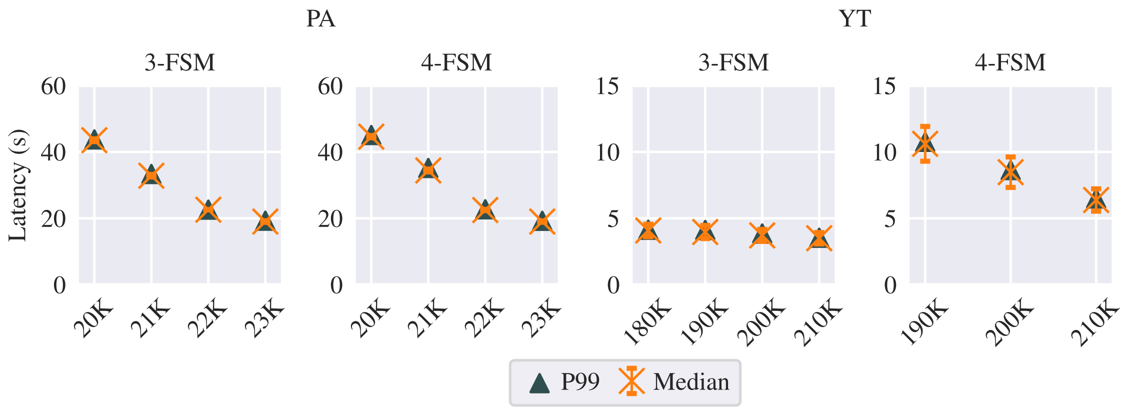


Figure 7.4: FSM latencies for Protean across different support thresholds (on x-axis).

G	Support	Protean	BRBL	IncGM+
3-FSM PA	20K	**43.01**	235.33	×
	21K	**33.46**	195.25	7017
	22K	**23.11**	156.02	4982
	23K	**18.97**	145.63	4246
3-FSM YT	180K	**4.1**	703.55	×
	190K	**3.97**	695.62	×
	200K	**3.71**	619.06	×
	210K	**3.5**	573.54	×
4-FSM PA	20K	**44.6**	242.63	×
	21K	**34.42**	195.46	×
	22K	**22.50**	155.93	×
	23K	**19.03**	143.54	×
4-FSM YT	190K	**10.61**	691.54	×
	200K	**8.47**	613.16	×
	210K	**6.38**	569.60	×

Table 7.4: FSM latencies (in seconds) for Protean, IncGM+ [1] and BRBL. × indicates the execution did not finish within 15 minutes.

which necessitates numerous expensive isomorphism checks as well as large intermediate state. Even BRBL outperforms IncGM+, since it is pattern-based and hence does not require either isomorphism checks nor intermediate state.

Updates Resulting in Snapshot Matching Tasks:

To assess the performance impact of snapshot tasks and P-DAG traversal, we profiled the 3-FSM execution on the YouTube graph with support threshold 210K. Figure 7.5 shows the performance breakdown for a period of execution containing 36 updates. Snapshot tasks are generated whenever the update leads to exploring new patterns (because their subpattern became frequent). We observe that snapshot tasks can be over 2× more expensive than differential tasks, leading to latency spikes in execution. The spikes were associated with newly discovered patterns by that update, numbered above the corresponding bar. As each new pattern generates a snapshot task, execution time increases as a function of the new pattern set.

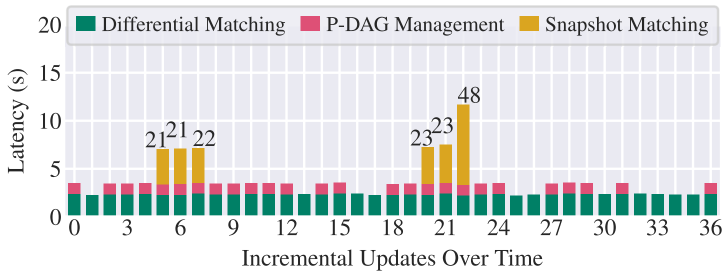


Figure 7.5: 3-FSM latencies (in seconds) for Protean on YouTube graph with 210K support. Numbers near the bars indicate the new patterns explored with snapshot matching.

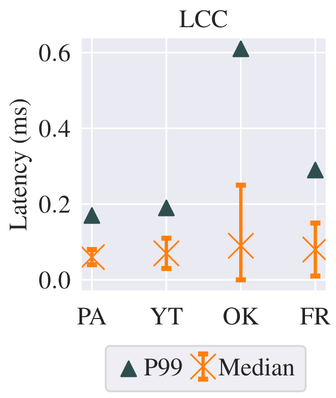


Figure 7.6: Local Clustering Coefficient latencies for Protean.

While P-DAG management is relatively inexpensive, growing and traversing the P-DAG on certain updates consumes about a second. For certain updates where the labels of an edge update affect a pattern hitherto unexplored, Protean generates a large number of candidate patterns (depending on edge mappings) and traverses the P-DAG to determine which to match. Throughout the execution of the 36 updates, the P-DAG contained over 10,000 patterns that were searched for each update in order to identify 158 new patterns.

7.2.4 Local Clustering Coeffiecient

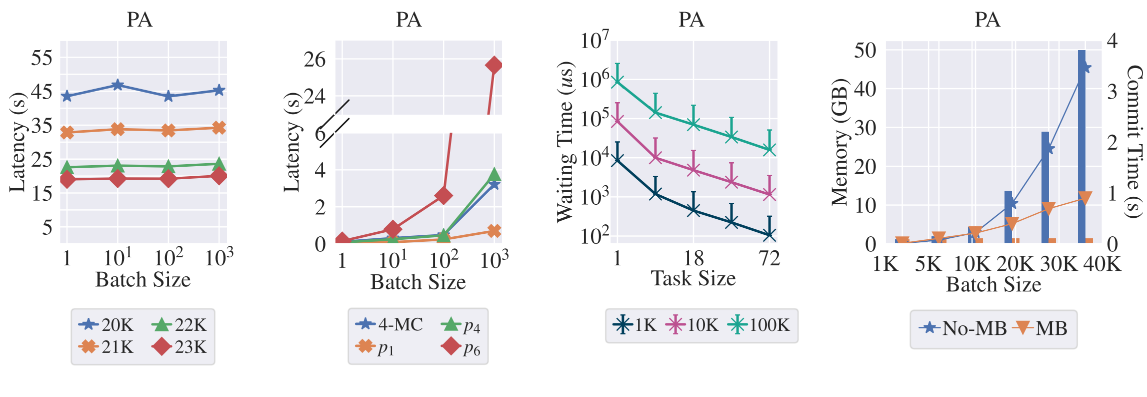
Figure 7.6 shows single update latencies for local clustering coefficient on Protean. Since we compute changes in the frequency of triangles and wedges for the updated vertices only, we were able to process the updates in less than half a millisecond for most of the updates on all the graphs.

7.3 Sensitivity Analysis

We study the sensitivity of Protean to varying update batch sizes, varying number of concurrent tasks, and impact of micro-batching for multiversioned graph management.

7.3.1 Varying Update Batch Size

Figure 7.7a-b shows the effect of varying update batch sizes on execution times for several applications on the Patents graph. Large batch sizes do not significantly affect latencies of FSM, as Protean traverses the P-DAG to match only necessary patterns and labeled patterns are extremely fast to match. Furthermore, as FSM deals with many different pattern tasks, their subtasks are executed in parallel. For pattern matching, we show the latencies for p_1, p_4, and p_6 to cover a range of difficulty: p_1 is relatively easy and p_6 is relatively hard to match compared to p_4. Here, increasing batch sizes leads to higher latency, especially for



(a) (b) (c) (d)

Figure 7.7: Sensitivity Analysis: (a) shows 3-FSM latencies for varying update batch sizes with support thresholds 20K to 23K; (b) shows motif counting and pattern matching latencies for varying update batch sizes; (c) shows the task waiting times for matching p_1 with 1K, 10K and 100K updates; (d) shows memory consumption and update commit time on the multiversioned graph with micro-batching (MB) and without micro-batching (No-MB).

the more expensive workloads, since the individual updates within the batch are processed as separate tasks, as explained in Section 6.1.

7.3.2 Varying Number of Concurrent Matching Tasks

To study the impact of concurrent execution on task waiting times, we fixed the update stream to contain 1K, 10K, and 100K updates and matched p_1 on the Patents graph with varying number of concurrent matching tasks.

As shown in Figure 7.7c, the task waiting time decreases as the number of concurrent matching tasks increase, since Protean is able to process more updates concurrently. This translates to improved end-to-end performance as individual updates correspond to independent tasks that can be performed in parallel. The dispatch latencies for 100K updates are higher and for 1K updates are lower than 10K updates, mainly because tasks contained in 10K updates have to wait for more number of tasks to finish before they are dispatched.

7.3.3 Impact of Micro-Batching for Multiversioning

We profiled the memory usage and the time taken to apply updates in the multiversioned graph with varying update batch sizes, as well as with the micro-batching optimization toggled with micro-batch size of 1000. Figure 7.7d shows the results. We observe that micro-batching succeeds in keeping memory usage of the multiversioned graph low regardless of batch sizes. Furthermore, micro-batching results in lower update times with large update batch sizes as lesser number of version pointers are copied upon updates.

Chapter 8

Conclusions & Future Directions

This thesis presents Protean, a general-purpose system for low-latency continuous graph mining on streaming graphs. Protean also eliminates unnecessary explorations using a novel differential pattern matching engine that maps the individual updates on to the patterns of interest and constructs an efficient pattern exploration plan to find matches affected by the corresponding updates. Moreover, Protean processes the streaming updates concurrently without matching automorphisms by enforcing an ordering on streaming updates using multiversioning strategy.

In addition to applications like motif counting where patterns of interest are know apriori, Protean also efficiently handles applications like frequent subgraph mining that dynamically determine the patterns to be explored at runtime. Protean tracks the cross-pattern dependencies in form of directed acyclic graph called P-DAG, and upon graph update it uses P-DAG to correctly invoke either differential pattern matching tasks or snapshot matching tasks for each pattern. To maximize performance, Protean uses different parallelization strategies for the two different tasks (differential matching tasks and snapshot matching tasks): the light-weight differential pattern matching tasks are invoked concurrently, whereas snapshot tasks are invoked serially such that each snapshot task is performed in parallel to leverage the available cores. Throughout this process, Protean coordinates incremental and non-incremental (i.e., from-scratch) computations to adjust aggregation results without maintaining intermediate subgraphs or incurring unnecessary explorations.

Protean's programming model enables users to express graph mining applications as static pattern programs and provides users with the option of synchronous (blocking) and asynchronous (non-blocking) output streams. This allows users to apply user-defined callbacks to the streaming results without blocking the Protean's pattern matching progress.

Our evaluation showed that Protean achieves low latency response to updates, often achieving sub-millisecond latencies for various graph mining use cases. Protean's update-driven parallel processing model enables orders of magnitude better performance than existing purpose-built streaming graph mining solutions, and scales to graphs with billions of edges.

8.1 Future Directions

Being the first general-purpose system for incremental graph mining on streaming graphs, Protean can be extended to use approximation techniques while processing graph updates, trading off exact results for better latencies. We can also broaden the scope of Protean to support graph processing and graph mining use cases using a general purpose programming interface.

8.1.1 Approximation Computing

Applications like FSM are often less sensitive to small number of graph updates since the support values increase with matches resulting in new mappings. Hence, maintaining up-to-date aggregation values for all patterns in P-DAG using differential matching and snapshot matching tasks might not result in new frequent patterns. This can be potentially leveraged by maintaining out-of-date or approximate results as updates get applied to reduce the update latencies and make Protean more responsive. The approximate results can be incrementally readjusted after certain updates depending on how stale the results are and where in the graph the updates get applied.

8.1.2 Unifying Graph Processing and Graph Mining

Several streaming graph processing solutions have been developed in literature [23, 32] that focus on iterative graph analytics workloads like pagerank and shortest paths. Protean can be extended using incremental iterative graph processing models in order to express complex graph mining queries that also rely on iterative graph analytics.

Bibliography

[1] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. Incremental Frequent Subgraph Mining on Large Evolving Graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE '17)*, 29(12):2710–2723, 2017.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment (PVLDB '18)*, 11(6):691–704, February 2018.

[3] Cigdem Aslay, Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, and Aristides Gionis. Mining Frequent Patterns in Evolving Graphs. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM '18)*, pages 923–932, 2018.

[4] Bjorn Bringmann and Siegfried Nijssen. What Is Frequent in a Single Graph? In *Advances in Knowledge Discovery and Data Mining (PAKDD '08)*, volume 5012, pages 858–863, 2008.

[5] Xiaowei Chen and John C.S. Lui. A Unified Framework to Estimate Global and Local Graphlet Counts for Streaming Graphs. In *Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM '17)*, page 131–138, 2017.

[6] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proceedings of the VLDB Endowment (PVLDB '20)*, 13(10):1190–1205, April 2020.

[7] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Statistics and Social Network of Youtube Videos. In *Interntional Workshop on Quality of Service (IWQoS '08)*, pages 229–238. IEEE, 2008.

[8] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. A Selectivity Based Approach to Continuous Pattern Detection in Streaming Graphs. *International Conference on Extending Database Technology, (EDBT '15)*, 2015.

[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] 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.

[11] 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.

[12] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, Qinghua Zheng, and Ting Liu. Frequent Subgraph Based Familial Classification of Android Malware. In *International Symposium on Software Reliability Engineering (ISSRE '16)*, pages 24–35, 2016.

[13] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental Graph Pattern Matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, page 925–936, 2011.

[14] Mathias Fiedler and Christian Borgelt. Subgraph Support in a Single Large Graph. In *IEEE International Conference on Data Mining Workshops (ICDMW '07)*, 2007.

[15] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. Toward Continuous Pattern Detection over Evolving Large Graph with Snapshot Isolation. *The VLDB Journal*, 25(2):269–290, April 2016.

[16] Joshua A. Grochow and Manolis Kellis. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Research in Computational Molecular Biology (RECOMB '07)*, 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] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the European Conference on Computer Systems (EuroSys '20)*, pages 1–16, 2020.

[19] 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.

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

[21] Paul Liu, Austin R. Benson, and Moses Charikar. Sampling Methods for Counting Temporal Motifs. In *Proceedings of the ACM International Conference on Web Search and Data Mining (WSDM '19)*, page 294–302, 2019.

[22] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. Llama: Efficient graph analytics using Large Multiversioned Arrays. In *IEEE 31st International Conference on Data Engineering (ICDE '15)*, pages 363–374. IEEE, 2015.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Stanley Milgram. The Small World Problem. *Psychology today*, 2(1):60–67, 1967.

[28] Rahmtin Rotabi, Krishna Kamath, Jon Kleinberg, and Aneesh Sharma. Detecting Strong Ties Using Network Motifs. In *Proceedings of the 26th International Conference on World Wide Web Companion (WWW '17)*, page 983–992, 2017.

[29] 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.

[30] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM (CACM '90)*, 33(8):103–111, 1990.

[31] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization (TACO '16)*, 13(4):32, 2016.

[32] 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.

[33] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. Efficient Sampling Algorithms for Approximate Temporal Motif Counting. In *Proceedings of the ACM International Conference on Information & Knowledge Management (CIKM '20)*, pages 1505–1514, 2020.

[34] 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.

[35] Li Wang, Hongying Zhao, Jing Li, Yingqi Xu, Yujia Lan, Wenkang Yin, Xiaoqin Liu, Lei Yu, Shihua Lin, Michael Yifei Du, Xia Li, Yun Xiao, and Yunpeng Zhang. Identifying Functions and Prognostic Biomarkers of Network Motifs marked by Diverse Chromatin States in Human Cell Lines. *Oncogene*, 39(3):677–689, September 2019.

[36] Pinghui Wang, John C.S. Lui, Don Towsley, and Junzhou Zhao. Minfer: A Method of Inferring Motif Statistics from Sampled Edges. In *IEEE International Conference on Data Engineering (ICDE '16)*, pages 1050–1061, 2016.

[37] 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.

[38] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowledge and Information Systems (KAIS '15)*, 42(1):181–213, 2015.

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