

Firth H, Missier P.

[Workload-aware streaming graph partitioning.](#)

In: GraphQ Workshop, co-located with EDBT/ICDT 2016 Joint Conference.

6-7 April 2016, Bordeaux, France: CEUR Workshop Proceedings.

Copyright:

© 2016, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2016 Joint Conference (March 15, 2016, Bordeaux, France) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CCby-nc-nd 4.0

Link to paper:

<http://ceur-ws.org/Vol-1558/paper26.pdf>

Date deposited:

22/09/2016



This work is licensed under a

[Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International licence](#)

Workload-Aware Streaming Graph Partitioning

Hugo Firth
School of Computing Science
Newcastle University
h.firth@ncl.ac.uk

Paolo Missier
School of Computing Science
Newcastle University
paolo.missier@ncl.ac.uk

ABSTRACT

Partitioning large graphs, in order to balance storage and processing costs across multiple physical machines, is becoming increasingly necessary as the typical scale of graph data continues to increase. A partitioning, however, may introduce query processing latency due to inter-partition communication overhead, especially if the query workload exhibits skew, frequently traversing a limited subset of graph edges. Existing partitioners are typically workload agnostic and susceptible to such skew; they minimise the likelihood of any edge crossing partition boundaries.

We present our progress on **LOOM**: a streaming graph partitioner based upon efficient existing heuristics, which reduces inter-partition traversals when executing a stream of sub-graph pattern matching queries \mathcal{Q} . We are able to continuously summarise the traversal patterns caused by queries within a window over \mathcal{Q} . We do this using a generalisation over a trie data structure, which we call **TPSTry++**, to compactly encode frequent sub-graphs, or motifs, common to many query graphs in \mathcal{Q} . When the graph-stream being partitioned contains a match for a motif, LOOM uses graph-stream pattern matching to capture it, and place it wholly within partition boundaries. This increases the likelihood that a random query $q \in \mathcal{Q}$ may be answered within a single partition, with no inter-partition communication to introduce additional latency.

Finally, we discuss the potential pitfalls and drawbacks which exist with our approach, and detail the work yet to be completed.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems

1. INTRODUCTION

Recently there has been a proliferation of web hyperlinks, social network users, protein interaction networks, and other content readily modelled as large graphs. Sub-graph pattern

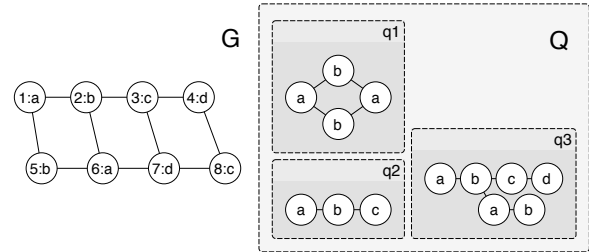


Figure 1: An example graph G with query workload \mathcal{Q}

matching over these graphs is common to many modern applications, including fraud detection [18], recommender systems [7] and genome analysis [4]. Pattern matching can be simply discussed in terms of sub-graph isomorphism, where, given a labelled query graph G_{q_i} and a labelled parent graph G , the answer to the query is all sub-graphs in G which are isomorphic to G_{q_i} ; i.e. all sub-graphs which have the same structure (vertices, edges, and labels) as G_{q_i} . For instance, given the graph and query workload in figure 1, the answer to q_1 would be the sub-graph of G containing the vertices 1, 2, 5, 6 and their interconnecting edges. Figure 1 also demonstrates query graphs which share common sub-structure. In this work, we exploit the existence of such frequently reoccurring sub-graphs within a query workload \mathcal{Q} to improve \mathcal{Q} 's performance over large, distributed graphs. We refer to frequent sub-graphs of query graphs as motifs.

Pattern matching is computationally complex and, over “big-graph data”, would prove prohibitively expensive to a single commodity machine. Distributed graph partitioning has long been seen as a viable approach to address such scalability issues in graph processing frameworks [11, 12], and graph database management systems (GDBMS) [1]. These systems distribute vertices and computation across multiple machines, using a simple hash function to determine vertex placement by default. Although a *hash-partitioning* is efficient to compute and creates partitions with even numbers of vertices, it ignores vertex locality, and is therefore to create a large number of inter-partition edges. This is undesirable, incurring a high communication overhead between partitions for many types of graph workload, including pattern matching queries.

The problem of minimising the number of inter-partition, or cut, edges in a distributed graph, whilst maintaining an even distribution of vertices, is known as *k-balanced graph*

partitioning, which is NP-Hard [3]. Despite this, there exist several practical solutions [8, 17, 19, 20] to the problem. Some, such as the state-of-the-art “offline” partitioner METIS [8], are memory intensive, their performance suffering over graphs with billions of vertices [19]. They may also have to perform expensive repartitioning in the presence of graph changes. Others [17, 19, 20] adopt a simpler streaming graph partitioning model. A graph-stream is an ordering over the elements of a dynamic, growing graph, often by creation time. Social networks are often viewed as graph-streams [16]. Procedures on graph-streams will usually consider each graph element, in order, just once and therefore likely have very good complexity, regardless of graph size. Streaming graph partitioners typically produce more inter-partition edges than METIS, but are much faster.

Although these streaming graph partitioners do reduce the number of cut edges and seamlessly handle graph updates, they are agnostic to the specific workload being executed over the graph partitioning: edges to be cut are computed purely based upon graph structure. For some workloads, which may traverse a limited subset of edges in a graph, such partitionings may incur unnecessary communication overhead [15, 20]. An example of such a workload is a one of pattern matching queries, where the topologies which are likely to be traversed are those which correspond to motifs defined in query graphs. In this work we focus on a different measure of partitioning quality: *the probability of inter-partition traversals* which is different from *the number of inter-partition edges*, given a workload \mathcal{Q} .

Whilst systems such as LogGP [20] do attempt to collect runtime statistics in order to improve graph partitioning for a given workload, they are focused on the Bulk-Synchronous-Parallel (BSP) model of computation used by Pregel-like systems for “offline” analytical workloads. Our goal, however, is to improve graph partitionings for a given “online” workload of pattern matching queries over a dynamic labelled graph, as would be common to GDBMS.

1.1 Contributions

We describe a novel extension to existing streaming graph partitioning methods [17], aimed at avoiding introducing unnecessary inter-partition communication overhead for a given query workload. More precisely, let \mathcal{Q} be a workload of queries over G , along with the relative frequency of each query in \mathcal{Q} . We are able to efficiently derive the most common motifs from the query graphs in \mathcal{Q} . Using an approach to graph-stream pattern matching [16], we identify those sub-graphs in G which match these motifs, and are therefore likely to be traversed during the execution of a random $q \in \mathcal{Q}$. Having grouped a graph-stream into frequently traversed sub-graphs, we are then able to use the successful *Linear Deterministic Greedy* heuristic [17] (LDG) to assign **these** to a partition, excepting some balance constraints. This increases the likelihood that a random $q \in \mathcal{Q}$ can be processed without causing inter-partition traversals and communication overhead.

Concretely, this work makes the following contributions:

- We extend an existing streaming graph partitioning approach [17], to account for the probabilities of crossing partition boundaries during execution of a query from a given workload \mathcal{Q} .
- We present an efficient intensional representation of

the probable edge traversals caused by a given workload of sub-graph pattern matching queries \mathcal{Q} .

- We propose a graph-stream pattern matching approach to transform a stream of vertices and edges G into a sequence of motifs, where each motif represents a sub-graph in G likely to be traversed during execution of a random $q \in \mathcal{Q}$.

The rest of this paper is organised as follows. In the subsequent section we discuss background material and related work. In Section 4.2 we present the *TPSTry++* datastructure for capturing an intensional representation of graph traversals from a query workload. In Section 4.3 we present a detailed overview of the streaming graph partitioner which accounts for the workloads captured in Section 4.2. In the Conclusion we discuss our progress with this work, highlight its limitations, and present some potential avenues for future study.

2. DEFINITIONS

A **labelled graph** $G = (V, E, L_V, f_l)$ is of the form: a set of vertices $V = \{v_1, v_2, \dots, v_n\}$, a set of pairwise relationships called edges $e = (v_i, v_j) \in E$ and a set of vertex labels L_V . The function $f_l : V \rightarrow L_V$ is a surjective mapping of vertices to labels.

A **graph motif** is simply a sub-graph structure which occurs repeatedly within a graph or graph-stream G .

A **pattern matching** query is defined in terms of sub-graph isomorphism. Given a pattern graph $Q = (V_Q, E_Q)$, a query should return G' : a set of sub-graphs of G . For each returned sub-graph $G'_i = (V'_i, E'_i)$ there should exist a bijective function f such that: (a) for every vertex $v \in V'_i$, there exists a corresponding vertex $f(v) \in V_Q$; (b) for every edge $(v_1, v_2) \in E'_i$, there exists a corresponding edge $(f(v_1), f(v_2)) \in E_Q$; and (c) for every vertex $v \in G'_i$, the labels match those of the corresponding vertices in Q , $l(v) = l(f(v))$.

A **graph partitioning** is defined as an disjoint family of sets of vertices $P_k(V) = \{V_1, V_2, \dots, V_k\}$. Each set V_i , together with its edges E_i (where $e_i \in E_i$, $e_i = (v_i, v_j)$, and $\{v_i, v_j\} \subseteq V_i$), is referred to as a *partition* S_i . A partition forms a proper sub-graph of G such that $S_i = (V_i, E_i)$, $V_i \subseteq V$ and $E_i \subseteq E$.

3. BACKGROUND & RELATED WORK

The three main areas of work which relate to our own are: 1) graph partitioning, particularly when workload-aware; 2) frequent sub-graph mining; & 3) graph-stream pattern matching. We provide an overview of the first below, but defer discussion of the latter two to sections 4.2 and 4.3 respectively, for context.

3.1 Graph partitioning

balanced graph partitioning is an NP-Hard problem with application to many areas across distributed systems and scientific computing; it has been exhaustively studied in literature since the 1970s [6, 8, 9], and several practical solutions exist [8, 19]. One such solution is METIS [8], a reliable standard for offline, fast partitioning. METIS is a *multilevel* technique: it computes a succession of recursively compressed graphs, partitions the smallest then “projects” that partitioning onto previous graphs in the sequence, applying local refinement techniques [9] to the partitioning at

each step. This produces a balanced k -way partitioning on the original graph, optimised for minimal edge cut.

Despite its prevalence, there are some issues with METIS which makes it unsuitable for our goal of workload-aware partitioning of a large, dynamic graph. Firstly, the performance of METIS suffers in the presence of graphs with more than a few hundred million elements [19]. Secondly, if a graph partitioning produced with METIS, or other offline techniques, grows over time then expensive full repartitioning operations will be required to maintain partition quality. Finally, METIS may account for a static query workload known *a priori*, using individual edge-weights to represent traversal frequency, however tracking this information is memory intensive, and otherwise non-trivial.

The streaming graph partitioning model [17] addresses the first two of these shortcomings. By assigning vertices and edges to a partition as soon as they arrive and not storing them to perform introspection of graph structure, such partitioners are able to maintain a small memory footprint. Thus, streaming partitioners such as Fennel [19], created by Tsourakakis *et al*, are able to scale to large graphs unbounded by the main memory of a host machine. Also, because element placement is computed “on the fly”, streaming partitioners adapt seamlessly to graph growth, applying the same placement operation for each new vertex and edge that arrives over time.

It is worth noting, however, that the heuristics used by streaming graph partitioners are sensitive to the order of graph elements in a stream [17]. There are three categories of graph ordering commonly considered when evaluating streaming graph partitioners: *random*, *adversarial* and *stochastic*. Consider an 2-way partitioning for the graph in figure 1, with a vertex ordering of $V = (1, 3, 6, 8, 2, 4, 5, 7)$. Given no neighbours for the first half of vertices received, a naive partitioner might greedily place them in a single partition which, intuitively, causes a final balanced partitioning with the maximum edge cut: $|E|$. This is an adversarial ordering. For the streaming graph partitioning approach described in this work, we will consider stochastic ordering; that is, a graph-stream continuously generated by some stochastic process, such as user input.

3.2 Workload aware partitioning

To the best of our knowledge, existing streaming graph partitioning solutions do not satisfy our goal for this work: producing *workload-aware* graph partitionings, which account for the edge traversals patterns of a given “online” workload of sub-graph pattern matching queries.

Xu *et al*’s LogGP [20] tackles workload aware partitioning improvement for graphs processed in Pregel-like systems [12], where operations are computed in a vertex-centric fashion across a series of *supersteps*. LogGP collects information about the set of vertices accessed in each superstep, using it to predict the set to be accessed in the next. Subsequently, this meta-data is incorporated with the original graph, transforming it into a hypergraph. With a novel streaming hypergraph partitioning technique, LogGP then repartitions the graph after each superstep, reducing the communication overhead for the next and reducing the overall execution time of an operation. Though LogGP’s approach is workload-aware, its dependence on supersteps and vertex-centric computation renders it unsatisfactory for our goal.

There are a number of works addressing the related problem of workload-aware data partitioning in distributed relational database systems [5,13]. *Schism* [5] and *SWORD* [13] use an *a priori* workload to generate a hypergraph, where each edge represents a set of tuples involved in a single transaction. This hypergraph is then partitioned using a version of METIS to achieve a minimal edge-cut. Mapped back to the original database, the partitioning represents an arrangement of records which causes a minimal number of transactions in the captured workload to be distributed.

Though the goals of these works and our own are similar, they are focused on a relational data model, where typical workloads overwhelmingly consist of short 1-2 “hop” queries. It is unclear how the techniques described would perform given a workload containing many successions of JOIN operations, equivalent to the traversals required for sub-graph pattern matching. Furthermore, these works do not consider dynamic graphs at all.

In [21], Yang *et al* propose algorithms to efficiently analyse online query workloads and to dynamically replicate “hotspots” (clusters of vertices over 2 or more partitions which are being frequently traversed), thereby temporarily dissipating network load. Whilst highly effective at dealing with unbalanced query workloads, Yang *et al* focus solely upon the replication of vertices and edges using temporary *secondary* partitions. They do not consider workload characteristics when producing the initial partitioning, nor do they consider workload characteristics when producing it. This can result in replication mechanisms doing far more work than is necessary over time, adversely affecting the performance of a system. As a result, the partitioning technique we present here could effectively complement many workload aware replication approaches, such as this.

4. LOOM PARTITIONING OVERVIEW

In this section we provide an intuition of how we are going to partition a graph-stream to account for a specific workload. Initially, we describe the efficient streaming-graph partitioning heuristic used as a base for our workload-aware extensions. The heuristic assesses characteristics of each individual vertex before placing them in an appropriate partition. However, by running efficient pattern matching procedures against a buffered window over the graph-stream, we are able to capture motifs; treating these motifs as single vertices, we may then use the same heuristic to place them wholly within beneficial partitions. If the motifs we capture correspond to those likely to be frequently traversed by a known workload Q , then this would increase the likelihood that a random $q \in Q$ is executed without inter-partition traversals. Thus, we subsequently present a method for continuously summarising the motifs most frequently traversed by a given stream of sub-graph pattern matching queries Q . Finally, we present our chosen pattern matching procedure and discuss the issues which currently exist with our partitioning approach.

4.1 Base partitioning heuristic

LOOM’s partitioning is based upon the *Linear Deterministic Greedy* heuristic (LDG) proposed by Stanton and Kliot [17]. LDG is a simple heuristic which seeks to assign a new vertex to the partition where it has the most edges, as this is efficient to compute, and greedily minimises the number of inter-partition edges for each vertex. In order to

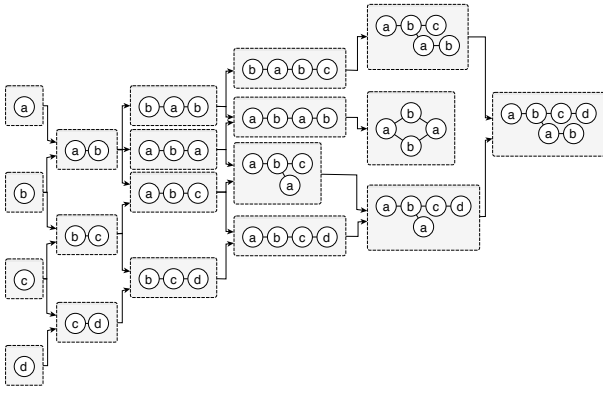


Figure 2: TPSTry++ for \mathcal{Q} in fig.1

create a partitioning which is balanced in the number of vertices, each partition is given a capacity constraint C . For a given vertex v and partition $S_i = (V_i, E_i)$, the number of v 's edges in S_i is weighted by S_i 's free capacity $1 - \frac{|V_i|}{C}$. In this way partitions are progressively more penalised the more vertices they contain. LDG is an effective heuristic [17, 19], reducing the number of edges cut by up to 90%.

In LOOM we buffer a sliding window over a graph-stream, and use LDG to assign both connected sub-graphs¹ and single vertices from the buffer to partitions. Stanton and Kliot describe similar extensions in their original work [17]. In particular, the *Greedy EvoCut* partitioning heuristic is closely related to our own. The local partitioning algorithm EvoCut [2] is used to split sub-graphs which occur within the stream buffer into small pseudo-partitions, which are then wholly assigned to parent partitions using LDG. In LOOM however, we attempt to detect sub-graphs within the stream buffer which are likely to be frequently traversed by a workload \mathcal{Q} , and greedily place **those** wholly within a partition.

4.2 Capturing a query workload

In order to detect the sub-graphs from stream G which are likely to be frequently traversed by a workload \mathcal{Q} , we must first discover those motifs which occur frequently within the query graphs defined in \mathcal{Q} . This act of discovery is a form of frequent sub-graph mining.

In a previous work by the authors which is submitted for publication elsewhere, we define the *traversal pattern summary trie* (TPSTry). The TPSTry datastructure, inspired by Li *et al*'s work [10] to find common traversal paths amongst sessions of hyperlink click-streams, is an encoding of the frequent motifs in a workload of path queries. The encoding is intensional, encoding paths of vertex labels, rather than the vertices themselves, in order to save space. Each node n is associated with the set of queries which could cause the path of traversals which n represents. Each node n in the trie is additionally associated with a probability $P(n)$, representing the likelihood of a traversal in graph G along a path whose vertex labels match those of the path $\varepsilon \rightarrow \dots \rightarrow n$ in the TPSTry. Using these probabilities, we are able to estimate the probability of any traversal from a vertex v , given its v label and those of v 's local neighbourhood.

¹When assigning sub-graphs, LDG considers the total edges from all vertices, to each partition.

In this work we extend the TPSTry data structure from a trie to a directed acyclic graph, which we call **TPSTry++**, capable of encoding the features of more complex motifs: branches, cycles etc. Encoding the motifs described by inexact pattern matching queries, such as those including variable length paths, is considered out of scope for this work.

The TPSTry++ is inspired by the work of Ribeiro and Silva, who propose G-Tries [14]. A G-Trie is a trie data structure which stores unlabelled graphs in such a way that a parent node in the trie represents a sub-graph of its children. Each graph in a G-Trie node is represented in its canonical form. A canonical form is guaranteed to be equal for two graphs which are isomorphic to one another, avoiding multiple trie branches per graph. In order to discover frequent sub-graphs (motifs), Ribeiro and Silva traverse the elements of a graph, constructing the branches of a G-Trie as they encounter distinct motifs. A p-value is associated with each node, based upon the number of times a particular motif has been observed.

This process is similar to how we construct a TPSTry++ from a stream of sub-graph pattern matching queries, however there are a number of differences. Firstly, as we capture labelled topologies, the TPSTry++ must be a directed acyclic graph (DAG), rather than a tree, as it may have multiple possible root nodes: one for each vertex with a distinct label. Secondly, we must use a different method for checking isomorphism between two motifs, as the unlabelled canonical form used when constructing G-Tries is no longer sufficient. To match two motifs, we use an efficient algorithm by Song *et al* [16] for computing numerical signatures for graphs. This algorithm is proposed as part of a graph-stream pattern matching approach, which we use to detect matches for \mathcal{Q} 's motifs in a graph-stream G . Both algorithm and pattern matching approach are detailed in the next section 4.3. Note that signature equality constitutes a non-authoritative form of isomorphism checking. However, the probability of signature collisions, and therefore of mistakenly representing distinct motifs with a single TPSTry++ node, is shown to be very low.

Figure 2 shows a representation of a TPSTry++ for the workload \mathcal{Q} in figure 1, without p-values. We capture the motifs common to \mathcal{Q} , along with their frequencies, by executing a simple co-recursive algorithm, presented in algorithm 1, for each query graph G_{q_i} .

Algorithm 1 Recompute TPSTry++ for each query $q \in \mathcal{Q}$

$qG \leftarrow$ the query graph defined by q
 $signature(g) \leftarrow$ the signature of a graph g
 $support(g) \leftarrow$ a map of TPSTry++ nodes to p-values
 $tpstry \leftarrow$ the TPSTry++ for \mathcal{Q}
 $g \leftarrow$ some sub-graph of qG , initially a single vertex

```

weave(qg, tpstry)
  for v in vertices from qG do
    g ← new graph with just {v}
    corecure(g, tpstry)
      sig ← signature(g)
      if sig not in tpstry then
        tpstry ← tpstry + g //Add a g node to TPSTry++
        support(g) ← support(g) + 1
        newEdges ← edges incident to g but not in g
        for e in newEdges
          corecure(g + e, tpstry) //Traverse through qG
  return tpstry

```

Any node in the TPSTry which has a p-value above a user-defined threshold \mathcal{T} is denoted frequent, and the node’s associated sub-graph is considered a **motif** in \mathcal{Q} .

4.3 Detecting motif matches in a graph-stream

The TPSTry++ for \mathcal{Q} provides a set of query motifs which, where they occur in G , are likely to be frequently traversed by a random $q \in \mathcal{Q}$. Given this information, we must attempt to identify sub-graphs in G which match these motifs, in order to make sure they cross partition boundaries as little as possible. As mentioned, global graph introspection or pattern matching operations are expensive and limit the scalability of a partitioner [19]. Instead, we use a graph-stream pattern matching algorithm to capture those sub-graphs in G which match a query motif and occur within a given window² over the graph-stream.

Song *et al* [16] propose a highly efficient algorithm based on *number theoretic signatures*. Offline, they construct a “signature” for each query graph G_{q_i} in a workload. This signature is really a large integer hash, which captures key information about a graph, such as vertices, labels and their degree, as distinct factors. Subsequently, as an edge e arrives online, the signature for a sub-graph S which contains e is calculated by multiplying the previous signature of the sub-graph $S \setminus e$ by the factor for e . If the signature for S is divisible by the signature for G_{q_i} then there is likely to be a match for q_i in S .

Song *et al* demonstrate that if a graph does **not** have a signature equal to that of a given query graph G_{q_i} , then it **cannot** be a match for the query q_i . Note that this is a weaker property than a signature match being equivalent to a graph match, and as such this pattern matching algorithm is non-authoritative; indeed, Song *et al* must use a secondary algorithm to verify matches. However, they also demonstrate that signature collision is highly unlikely, which should be sufficient for our purposes of heuristically improving a partitioning, without further verification.

Recall from algorithm 1 that a signature is computed for each motif represented by a node in the TPSTry++. As each edge arrives in the graph-stream, if it connects two vertices within the stream window to form a sub-graph S , then we compute a signature. If the signature is a match for a node n in the TPSTry++, then S is a match for a motif. For a subsequently added edge, the signature for sub-graph S' must match a signature associated with a child of n , or else S' is not a match for a motif. Note that the sub-graph S' not being a match for a motif does **not** imply that the newly added edge is not part of a sub-graph which is a match.

Figure 3 presents an illustrative example of the above. The subgraph S' is not a match for any node of the TPSTry++ in figure 2, however it contains two distinct instances of the *abc* motif. The pattern matching algorithm does not detect this, because sub-graph signatures are iteratively recomputed with each update, and previous signatures discarded. As a result, we risk assigning the added c labelled vertex to a different partition than the sub-graph S , creating an inter-partition edge which is likely to be traversed. In order to avoid this, we adopt the following simple procedure: if an edge e is added to a sub-graph S , and the new sub-graph S' is not a match for a motif in the TPSTry++, then we incrementally compute a new signature for S' , starting with

²Stream windows may be defined in terms of time, or element count

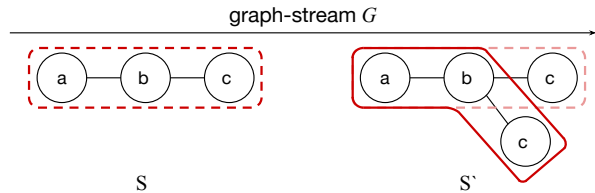


Figure 3: Motif matching over the graph-stream

the new edge e . This computation is similar to algorithm 1, in that we traverse each edge in S' , starting with those incident to vertices in e . After each step we recompute the signature for the sub-graph of S' which we have traversed so far. If this recomputed signature is not in the TPSTry++ then we discard the most recent edge, and do not traverse to its neighbours. We eventually traverse and identify the largest sub-graph of S' which both contains the edge e and is a match for a query motif³.

4.4 Assigning motif matches to partitions

The pattern matching algorithm described in the previous section will maintain the set of sub-graphs which are currently within the graph-stream window, and which match common motifs from a query workload \mathcal{Q} . Over time, vertices and edges will leave the stream window and be assigned to partitions using the LDG heuristic, as mentioned in section 4.1. When the “oldest” vertex in a motif match is due to be assigned, we assign the whole matching sub-graph at once. Other matching sub-graphs which share common substructure with the sub-graph being assigned, as in figure 3, will also be assigned to the same partition. This greedy approach is naive, as it risks assigning some motif matching sub-graphs to sub-optimal partitions because they share substructure with another sub-graph which was assigned earlier. Furthermore if a set of connected sub-graphs is very large, it is unclear what effect this would have on partition balance, even given LDG’s penalty weighting. Evaluating alternative approaches, including local partitioning of motif matches to separate them across partitions, is a focus of our ongoing work.

As stated previously, isolated vertices, or sub-graphs which do match motifs from \mathcal{Q} , are assigned according to the LDG heuristic.

5. CONCLUSION AND FUTURE WORK

We have presented our ongoing work on LOOM: a workload-aware streaming graph partitioner. Our primary contribution is using a generalised trie data structure to identify query motifs, small sub-graphs common to many of the query graphs defined in a sub-graph pattern matching workload \mathcal{Q} . We have also described how we use an efficient graph-stream pattern matching technique to identify matches for query motifs in a graph-stream G , and greedily assign these matches to partitions to reduce the *probability of inter-partition traversals*.

As future work we will perform extensive evaluation of the prototype LOOM architecture, specifically in the presence of a number of different graph-stream orderings, and different query workloads. Furthermore, our choice of greedy

³This may be none!

assignment semantics, never splitting sub-graphs in G which match query motifs, risks poor performance when large sub-graphs are assigned to sub-optimal partitions in order to maintain partition balance. We must propose a local partitioning procedure for large matched sub-graphs which alleviates this. Finally it would be interesting to extend our base partitioning heuristic (LDG) to incorporate edge traversal probabilities from the TPSTry++ into the process of selecting assignment partitions.

6. REFERENCES

- [1] Titan - Distributed Graph Database. <http://thinkaurelius.github.io/titan/>. Accessed on: 2015-12-01.
- [2] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*, page 235, New York, New York, USA, 2009. ACM Press.
- [3] K. Andreev and H. Racke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, nov 2006.
- [4] G. Brevier, R. Rizzi, and S. Vialette. Pattern Matching in Protein-Protein Interaction Graphs. In *Fundamentals of Computation Theory*, pages 137–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.
- [6] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, mar 1995.
- [7] Z. Huang, W. Chung, T.-H. Ong, and H. Chen. A graph-based recommender system for digital library. In *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 65–73, 2002.
- [8] G. Karypis and V. Kumar. Multilevel k -way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.
- [9] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell systems technical journal*, 49(2):291–307, 1970.
- [10] H. Li and S. Lee. Mining Top-K Path Traversal Patterns over Streaming Web Click-Sequences. *Journal of Information Science and Engineering*, 1133(95):1121–1133, 2009.
- [11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab. *Proceedings of the VLDB Endowment*, 5(8):716–727, apr 2012.
- [12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 135, New York, New York, USA, 2010. ACM Press.
- [13] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD. In *Proceedings of the 16th International Conference on Extending Database Technology*, page 430. ACM Press, 2013.
- [14] P. Ribeiro and F. Silva. G-Tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery*, 28(2):337–377, mar 2014.
- [15] Z. Shang and J. X. Yu. Catch the Wind: Graph workload balancing on cloud. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 553–564, apr 2013.
- [16] C. Song, T. Ge, C. Chen, and J. Wang. Event pattern matching over graph streams. *Proceedings of the VLDB Endowment*, 8(4):413–424, dec 2014.
- [17] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012.
- [18] H. Tong, B. Gallagher, C. Faloutsos, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 737, 2007.
- [19] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342, 2014.
- [20] N. Xu, L. Chen, and B. Cui. LogGP. *Proceedings of the VLDB Endowment*, 7(14):1917–1928, oct 2014.
- [21] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 international conference on Management of Data*, pages 517–528. ACM Press, 2012.