Approximate Pattern Matching in Massive Graphs with Precision and Recall Guarantees

Tahsin Reza
Matei Ripeanu
Electrical and Computer Engineering
University of British Columbia
{treza,matei}@ece.ubc.ca

Geoffrey Sanders
Roger Pearce
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
{sanders29,rpearce}@llnl.gov

ABSTRACT

There are multiple situations where supporting approximation in graph pattern matching tasks is highly desirable: *(i)* the data acquisition process can be noisy; *(ii)* a user may only have an imprecise idea of the search query; and *(iii)* approximation can be used for high volume vertex labeling when extracting machine learning features from graph data. We present a new algorithmic pipeline for approximate matching that combines edit-distance based matching with systematic graph pruning. We formalize the problem as identifying all exact matches for up to k edit-distance subgraphs of a user-supplied template. We design a solution which exploits unique optimization opportunities within the design space, not explored previously. Our solution is *(i)* highly scalable, *(ii)* supports arbitrary patterns and edit-distance, *(iii)* offers 100% precision and 100% recall guarantees, and *(vi)* supports a set of popular data analysis scenarios. We demonstrate its advantages through an implementation that offers good strong and weak scaling on massive real-world (257 billion edges) and synthetic (1.1 trillion edges) labeled graphs, respectively, and when operating on a massive cluster (256 nodes/9,216 cores), orders of magnitude larger than previously used for similar problems. Empirical comparison with the state-of-the-art highlights the advantages of our solution when handling massive graphs and complex patterns.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Information systems → Data mining**; • **Mathematics of computing → Graph algorithms**.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380566

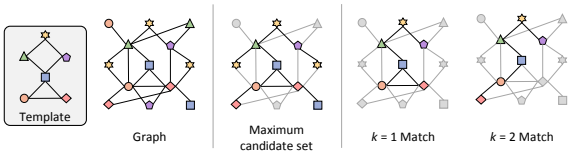


Figure 1: Edit-distance based approximate matching: (left) a search template $\mathcal{H}_0$ and background graph $\mathcal{G}$, and (right) example matches at $k = 1$ and $k = 2$ edit-distance. (Center) the *maximum candidate set* for the search template - the (approximate) match superset.

KEYWORDS

graph processing; pattern matching; distributed computing

ACM Reference Format:
Tahsin Reza, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. 2020. Approximate Pattern Matching in Massive Graphs with Precision and Recall Guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3380566

1 INTRODUCTION

Pattern matching in graphs, that is, finding subgraphs that match a small *template graph* within a large *background graph* has applications in areas as diverse as information mining [17, 72], anomaly and fraud detection [33], bioinformatics [2] and program analysis [36]. A *match* can be broadly categorized as either *exact* - there is a bijective mapping between the vertices/edges in the template and those in the matching subgraph [66], or *approximate* - the template and the match are just similar by some defined similarity metric [9, 73]. Multiple real-world usage scenarios justify the need for approximate matching. These include:

(S1) Dealing with the computational intractability of exact matching - to reduce the asymptotic complexity of exact matching [2, 16, 33] (as, in the general case, this problem is not known to have a polynomial time solution) by relaxing the quality of the solution over at least one of multiple axes. The diversity of these axes highlights the multiple and overlapping meanings with which the term *approximate* has been used. On the one side, most solutions do not provide recall guarantees, i.e., not all

possible matches are returned. On the other side, for many solutions, the user is not offered strong guarantees on the the similarity level between the search template and the matches offered, thus they do not offer precision guarantees [12].

(S2) *Uncertainty regarding acquired data* - the acquired data can be noisy, leading to a background graph that is different from the ground truth [12, 73]. In these cases, approximate matching is used to highlight subgraphs that may be of interest and have to be further inspected; for example, in genomics pipelines [43].

(S3) *Exploratory search* - a user may not be able to come up with a search template a priori [13]. In such scenarios, the user starts with an approximate idea of what (s)he may be looking for, and relies on the system's ability to identify 'close' or approximate matches [4, 33, 73]. Multiple application areas (e.g., financial fraud detection and organized crime group identification) have usage scenarios that fall in this category [31].

(S4) *Extracting features for machine learning from networked data* - as most machine learning solutions use data in tabular form, they are unable to directly incorporate topological information from networked data. A number of recent efforts address this problem [25, 34, 47] by using sampling techniques to collect neighborhood information that is then used as a vertex feature. A complementary feature engineering strategy is marking each vertex with the patterns in which it participates. Approximate matching can be used to define groups of patterns of interest: a high-throughput matching pipeline can be used to bulk-label the background graph; it produces a per-vertex vector that indicates whether the vertex participates in an approximate match (bounded by, for example, a user-provided edit-distance), and indicates the specific patterns(s) it participates in matches for.

(S5) *Other use cases* - Aggarwal et al. [1, 33] present additional use cases that can be mapped to approximate matching. These include, frequent subgraph mining [2, 5, 58], motif counting [2, 58], or graph alignment [43, 70].

The Target Problem Scenarios. We target problem scenarios where the user: *(i)* has a strict definition of match similarity: i.e., a user-specified bound on the similarity between the search template and the approximate matches returned by the system; and *(ii)* needs *full precision* (i.e., there are no false positive in the solution set) and *full recall* (i.e., all matching vertices and edges are identified). Multiple use cases in the categories *(S2) – (S5)* above, benefit from these properties.

Among these scenarios, we primarily target a use case where, rather than enumerating individual matches, the user is interested in efficiently labeling each vertex in the background graph with the version(s)[1] of the search template it matches. On the one side, this is useful for generating labels for a machine learning pipeline, e.g., *(S4)*; and it implies that we need the ability to handle situations where matches are frequent. On the other side, this supports obtaining various derived output, for example: *(i)* the union of all the matches; *(ii)* the union of matches for each template version separately; or *(iii)* the full match enumeration for each template version; all with guarantees of 100% precision and 100% recall. This way, scenarios *(S2) – (S5)* above can be supported. The experimental section (§5) focuses on demonstrating viability of our solution for scenarios *(S3)* and *(S4)*, and to compare with related work, we target motif counting (part of *(S5)*).

The Similarity Metric: Edit-Distance. We quantitatively estimate similarity between the user-provided search template and a candidate match using *edit-distance* [9]. We restrict the possible edits to *edge deletion*, while allowing the user to indicate *mandatory edges*. Additionally, we restrict edits so that the search template remains connected. We discuss the reason to focus on edge deletions and possible extensions in §3.

One high-level intuition for the semantic of the search is the following: the user is searching for a set of interrelated entities each belonging to some category (corresponding to labeled vertices in the search template). The user specifies a superset of the relationships between these items (corresponding to edges in the search template), as well as the maximum number of relationships that may be removed by the search engine (corresponding to the edit-distance), and may indicate mandatory relationships (i.e., edges in the search template).

Overview of the Approximate Matching Solution. Given a search template and an edit-distance k, we aim to identify all vertices and edges participating in each k edit-distance version of the search template in the background graph. Since we seek full precision and recall, this class of approximate matching queries can be equivalently stated as the problem of *finding exact matches for all* $0, \ldots, k$ *edit-distance prototypes*[1] *of the given search template* [73].

Our solution harnesses two high-level design principles: *(i)* search space reduction, and *(ii)* redundant work elimination, which, we show, enable significant performance gains compared to a naïve approach (that would use exact matching to independently search each prototype). The implementation of these principles is made feasible by three observations: First, the *containment rule* (§2, Obs. 1) observes that prototypes p at distance $k = \delta$ from the original search template can be searched within the reduced subgraph generated while identifying matches at distance $k = \delta + 1$ without degrading recall. Second, we observe that one can efficiently

[1] A *prototype* is a version of the original search template within the edit-distance bound specified by the user.

build the *maximum candidate set* that eliminates all the vertices and edges that do not have any chance to participate in a match regardless of the distance to the search template, thus reducing the initial search space. Finally, we observe that prototypes have common substructures; thus viewing prototypes as a *set of constraints*, each vertex and edge participating in a match must meet [51], enables finer work granularity, and makes it possible to identify and eliminate redundant work when searching for exact matches of similar prototypes. We present the full intuition for our approach, and the algorithmic and distributed implementation details in §3 and §4, respectively.

Contributions. We make the following contributions:

(i) Solution Design. We design a solution for a class of approximate matching problems that can be stated as identifying all vertices and edges that participate in exact matches for up to k edit-distance prototypes of a given search template. Our design views the search template as specifying a set of constraints and exploits key relationships between prototypes (§2) to reduce the search space and eliminate redundant work (§3).

(ii) Optimized Distributed Implementation. We offer a proof-of-concept implementation (§4) on top of HavoqGT [46], an open-source asynchronous graph processing framework. Our implementation provides infrastructure support for iterative search space reduction, transferring the results of constraint verification between prototypes at edit-distance one (which enables eliminating redundant work), effective match enumeration, load balancing (with the ability to relaunch computation on a pruned graph using a smaller deployment and search prototypes in parallel), and producing key types of output (e.g., labeling vertices by prototype membership(s)).

(iii) Proof of Feasibility at Scale. We demonstrate the performance of our solution by experimenting on datasets orders of magnitude larger than those used by the prior work (§5). We show a *strong scaling* experiment using the largest openly available real-world dataset whose undirected version has over 257 billion edges; and a *weak scaling* experiment using synthetic, R-MAT [11] generated graphs of up to 1.1 trillion edges, on up to 256 compute nodes (9,216 cores). We show support for patterns with arbitrary label distribution and topology, and with an edit-distance that leads to generating thousands of prototypes. To stress our system, we consider patterns containing high frequency vertex labels (up to 9.5 billion instances). Finally, we empirically compare our work with a state-of-the-art system, Arabesque [63], and demonstrate the significant advantages our system offers for handling large graphs and complex patterns.

(iv) Demonstrate Support for Multiple Usage Scenarios, including Bulk-labeling and Exploratory Search. In addition to demonstrating support for match enumeration and counting, we show how the pipeline can be used to efficiently bulk-label the vertices of a graph with their membership to various prototypes within edit-distance from the user-provided search template: over 150 million labels are generated in a massive webgraph with 3.5 billion vertices by discovering membership in the 150 prototypes within edit-distance $k = 4$ from the search template (Fig. 8). On the other side, we show a use case of exploratory search: the system begins with a 6-Clique and extends the search by incrementally increasing the edit-distance until the first match(es) are discovered; searching over 1,900 prototypes in the process (§5.5).

(v) Application Demonstration on Real-World Datasets. We demonstrate that our solution lends itself to efficient computation and pattern discovery in real-world approximate pattern matching scenarios: we use two real-world metadata graphs we have curated from publicly available datasets Reddit (14 billion edges) and the smaller International Movie Database (IMDb), and show practical use cases to support rich pattern mining (§5.5).

(vi) Impact of Optimizations and Evaluating Overheads. We study the impact and trade-offs of a number of optimizations used (§5.4). We demonstrate the cumulative impact of these optimizations significantly improves the runtime over the naïve approach (§5.3). Also, we analyze network communication and memory usage of our system, and compare with the naïve approach (§5.7).

Table 1: Symbolic notation used.

Object(s)	Notation
background graph, vertices, edges	$\mathcal{G}(\mathcal{V}, \mathcal{E})$
background graph sizes	$n := \lvert\mathcal{V}\rvert$, $m := \lvert\mathcal{E}\rvert$
background vertices	$\mathcal{V} := \{v_0, v_1, \ldots, v_{n-1}\}$
background edges	$(v_i, v_j) \in \mathcal{E}$
max./avg./stdev. of vertex degree	$d_{max}, d_{avg}, d_{stdev}$
label set	$\mathcal{L} = \{0, 1, \ldots, \lvert\mathcal{L}\rvert - 1\}$
vertex label of v_i	$\ell(v_i) \in \mathcal{L}$,
search template, vertices, edges	$\mathcal{H}_0(\mathcal{W}_0, \mathcal{F}_0)$
distance k prototype p	$\mathcal{H}_{k,p}(\mathcal{W}_{k,p}, \mathcal{F}_{k,p})$
set of all dist.-k prototypes	$\mathcal{P}_k$
set of non-local constraints[2] for $\mathcal{H}_0$	$\mathcal{K}_0$
maximum candidate set	$\mathcal{M}^\star$
solution subgraph w.r.t. $\mathcal{H}_{k,p}$	$\mathcal{G}^\star_{k,p}(\mathcal{V}^\star_{k,p}, \mathcal{E}^\star_{k,p})$

2 PRELIMINARIES

Our aim is to find structures similar to a small labeled *template graph*, $\mathcal{H}_0$, within a very large labeled *background graph*, $\mathcal{G}$. We describe the graph properties of $\mathcal{G}$, $\mathcal{H}_0$, and the other graph objects we employ. Table 1 summarizes our notations.

A *vertex-labeled graph* $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{L})$, is a collection of n vertices $\mathcal{V} = \{0, \ldots, n-1\}$ and m edges $(i, j) \in \mathcal{E}$, where

$i, j \in \mathcal{V}$, and each vertex has a discrete label $\ell(i) \in \mathcal{L}$. We often omit $\mathcal{L}$ in writing $\mathcal{G}(\mathcal{V}, \mathcal{E})$, as the label set is shared by all graph objects in a given calculation. Here, we assume $\mathcal{G}$ is *simple* (i.e., no self edges or multiple edges), *undirected* ($(i, j) \in \mathcal{E}$ implies $(j, i) \in \mathcal{E}$), and *vertex-labeled*, although the techniques we develop can be easily generalized, including to *edge-labeled graphs*.

We discuss several graph objects simultaneously and use sub- and super-scripts to denote the associations of graph constituents: the background graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the search template $\mathcal{H}_0(\mathcal{W}_0, \mathcal{F}_0)$ (vertices, $\mathcal{W}_0$, and edges, $\mathcal{F}_0$), as well as several low-edit-distance approximations to $\mathcal{H}_0$, which we call *template prototypes* or, in short, *prototypes*.

Definition 1. **(Prototypes within Edit-Distance k.)** *The* edit-distance *between two graphs $\mathcal{G}_1, \mathcal{G}_2$ with $|\mathcal{V}_1| = |\mathcal{V}_2|$ (and all vertex label counts equivalent) is the minimum number of edits (i.e., edge removal in our context) one needs to perform to make $\mathcal{G}_1$ isomorphic to $\mathcal{G}_2$. For template $\mathcal{H}_0(\mathcal{W}_0, \mathcal{F}_0)$, we define* template prototype *$\mathcal{H}_{\delta,p}(\mathcal{W}_{\delta,p}, \mathcal{F}_{\delta,p})$ as a connected subgraph of $\mathcal{H}_0$ such that $\mathcal{W}_{\delta,p} = \mathcal{W}_0$ and $\mathcal{F}_{\delta,p} \subset \mathcal{F}_0$ that is edit-distance $\delta \leq k$ from $\mathcal{H}_0$. Index p is merely to distinguish multiple edit-distance δ prototypes. Note $\mathcal{H}_{0,0}$ is the template $\mathcal{H}_0$ itself. Let $\mathcal{P}_k$ be the set of all connected prototypes within edit-distance k.*

Definition 2. **(Solution Subgraph $\mathcal{G}^\star_{\delta,p}$ for Prototype $\mathcal{H}_{\delta,p}$)** *The subgraph $\mathcal{G}^\star_{\delta,p} \subset \mathcal{G}$ containing all vertices and edges participating in one or more exact matches to $\mathcal{H}_{\delta,p}$.*

Definition 3. **(Approximate Match Vectors)** *The membership of a vertex (or edge) to each $\mathcal{G}^\star_{\delta,p}$ for all $p \in \mathcal{P}_k$ is stored in a length $|\mathcal{P}_k|$ binary vector that represents if, and how, the vertex (or edge) approximately matches $\mathcal{H}_0$ within distance k.*

The approximate match vectors represent a rich set of discrete features usable in a machine learning context; our techniques could also populate the vector with prototype participation rates, should a richer set of features be desired.

We list below the two key relationships between templates that are edit-distance one from each other that we leverage to form more efficient approximate matching algorithms. Essentially, we can more efficiently compute the solution subgraph for a given prototype $\mathcal{G}^\star_{\delta,p}$, from information gained while previously computing solution subgraphs of prototypes with higher- (or lower-) edit-distance $\mathcal{G}^\star_{\delta+1,p'}$ (§3).

Observation 1. **Containment Rule.** *Consider two prototypes $\mathcal{H}_{\delta,p}, \mathcal{H}_{\delta+1,p'}$ in $\mathcal{P}_k$ that are within edit-distance one with respect to each other, i.e., $\mathcal{F}_{\delta,p} = \mathcal{F}_{\delta+1,p'} \cup \{(q_{i_{p'}}, q_{j_{p'}})\}$. Let $\mathcal{E}(\ell(q_{i_{p'}}), \ell(q_{j_{p'}}))$ be the set of all edges in $\mathcal{E}$ that are incident to labels $\ell(q_{i_{p'}})$ and $\ell(q_{j_{p'}})$. We have $\mathcal{V}^\star_{\delta,p} \subset \mathcal{V}^\star_{\delta+1,p'}$ and $\mathcal{E}^\star_{\delta,p} \subset \mathcal{E}(\ell(q_{i_{p'}} i), \ell(q_{j_{p'}})) \cup \mathcal{E}^\star_{\delta+1,p'}$. This implies*

$$\mathcal{V}^\star_{\delta,p} \subset \bigcap_{p' : \mathcal{F}_{\delta,p} = \mathcal{F}_{\delta+1,p'} \cup \{(q_{i_{p'}}, q_{j_{p'}})\}} \mathcal{V}^\star_{\delta+1,p'} \quad \text{and}$$

$$\mathcal{E}^\star_{\delta,p} \subset \bigcap_{p' : \mathcal{F}_{\delta,p} = \mathcal{F}_{\delta+1,p'} \cup \{(q_{i_{p'}}, q_{j_{p'}})\}} \left(\mathcal{E}^\star_{\delta+1,p'} \cup \mathcal{E}(\ell(q_{i_{p'}}), \ell(q_{j_{p'}})) \right).$$

Observation 2. **Work Recycling.** *We recycle information gained during non-local constraint checking (defined in §3) in a 'top-down' manner: If a vertex/edge passes a non-local constraint check for $\mathcal{H}_{\delta,p_1}$, then it will pass the check for $\mathcal{H}_{\delta+1,p_2}$. Additionally, we recycle information in a 'bottom-up' manner: if a vertex/edge passes a non-local constraint check for $\mathcal{H}_{\delta+1,p_2}$, then it will likely pass the check for $\mathcal{H}_{\delta,p_1}$ (or $\mathcal{H}_{\delta+1,p_3}$) and we postpone on checking that constraint until the last verification phase $\mathcal{G}^\star_{\delta,p_1}$ (or $\mathcal{G}^\star_{\delta,p_3}$). ('Lateral' recycling between $k = \delta$ prototypes is also possible but not explored in this paper.)*

3 SOLUTION OVERVIEW

Given a search template $\mathcal{H}_0$ and an edit-distance k, our primary goal is the following: for each vertex v in the background graph $\mathcal{G}$, populate a $|\mathcal{P}_k|$ length binary vector indicating, for each prototype p in $\mathcal{P}_k$, whether v participates in at least one match with p.

Two high-level design principles are the cornerstones for our approach and enable its effectiveness and scalability: *(i)* search space reduction, and *(ii)* redundant work elimination. To implement these principles, our approach leverages the key observations on relationships between prototypes at edit-distance one presented in §2 (Obs. 1 and 2), and takes advantage of the observation that the search template can be seen as specifying a set or constraints each vertex and edge participating in a match must meet.

This section is structured as follows: first, it presents our reason to base our approach on constraint checking (and not on one of the many other existing approaches for exact pattern matching); then it summarizes how constraint checking has been used in the past for exact pattern matching [51]; and finally, it presents the contributions of this paper: how we implement the aforementioned design principles by extending the constraint checking primitives, and an overview of the approximate matching pipeline we propose.

Why Build on Constraint Checking? The key reason that makes us believe that starting from the constraint checking approach offers a major advantage compared to starting from other exact matching approaches is the following: Support for the containment rule (Obs. 1, key to our search space reduction technique) requires efficiently producing the solution subgraphs (Def. 2). A solution based on constraint checking offers directly the solution subgraph, at a much lower cost than any other exact matching framework we are aware of (as these focus on directly producing the match enumeration, and the solution subgraph would need to be inferred from matches). A second advantage is that thinking in terms of constraints can be employed uniformly across our system (as presented in this paper).

Constraint Checking for Exact Pattern Matching. A search template $\mathcal{H}_0$ can be interpreted as a set of constraints the vertices and edges that participate in an exact match must meet [51]. To participate in an exact match, a vertex must satisfy two types of constraints: *local* and *non-local*, (Fig. 2 (top) presents an example). *Local constraints* involve only the vertex and its neighborhood. Specifically, a vertex participating in an exact match needs to have non-eliminated edges to non-eliminated vertices labeled as prescribed by the adjacency structure of its corresponding vertex in the search template. *Non-local constraints* are topological requirements beyond the immediate neighborhood of a vertex. These are required by some classes of templates (with cycles or repeated vertex labels) and require additional checks to guarantee that all non-matching vertices are eliminated. (Fig. 2 (bottom) illustrates the need for these additional checks with examples). Essentially, non-local constraints are directed walks in the search template. A vertex in the background graph that is part of a match, verifies all the non-local constraints for which it is the source in the search template; that is, a walk starting from the vertex and similar to that specified in the constraint, must exist in the background graph. Past work defines the necessary set of constraints needed to guarantee full precision and recall for exact matching [51], and proposes heuristics to select and order additional constraints to enhance performance [65].

The exact matching pipeline iterates over these constraints to eliminate all the vertices and edges that *do not* meet *all* constraints, and thus reduces the background graph to the solution subgraph for the search template. The *approximate* search pipeline builds on the same constraint checking primitives yet introduces innovations at multiple levels: at the algorithmic level it uses new algorithms and heuristics that make use of these primitives; while at the infrastructure level it maintains additional state to identify: *(i)* edit-distance level solution subgraphs, and *(ii)* constraint execution results that may be reused across prototypes. Additionally, the infrastructure manages the intermediate search state differently, and uses different optimizations in terms of constraint ordering and load balancing. We explain these below.

Search Space Reduction. A first design principle we embrace is search space reduction: we aim to aggressively prune away the non-matching part of the background graph (both vertices and edges). Search space pruning has multiple benefits: first, it leads to improved node-level locality by reducing the problem size; second, it eliminates unprofitable parts of the search space early, leading to lower compute and network overheads; and finally, it enables, reloading the problem on a smaller set of nodes, thus reducing network traffic and improving overall solution efficiency. Search space reduction is made feasible by two observations: *(i)* the maximum candidate set and *(ii)* the containment rule (Obs. 1).

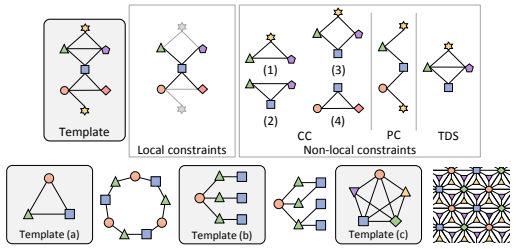


Figure 2: (Top) Local and non-local constraints of a template: Local constraints involve a vertex and its immediate neighborhood; here, the blue (square) vertex has four neighbors with distinct labels. On the right, the various non-local constraints that can be generated: CC - cycle constraints, PC - path constraints and TDS - template-driven search constraints. Since the template is non-edge-monocyclic, the TDS constraint is required - it is created by combining two cycles constraints, (1) and (2), that share an edge. (Bottom) Three examples that illustrate the need for non-local constraints: invalid structures (at the right of each template) that would survive in a graph if only local constraints are used (reproduced from [51]).

The Maximum Candidate Set. A first observation is that some of the vertices and edges of the background graph that can not possibly participate in any match, can be identified only based on their local properties. For example, these are the vertices that do not have a label that exists in the search template; or that do not have at least one neighbor potentially participating in any approximate match with the search template. We call this subgraph the *maximum candidate set*, $\mathcal{M}^\star \subseteq \mathcal{G}$, that is the union of all possible approximate matches of the template $\mathcal{H}_0$, irrespective of the edit-distance k. Fig. 1 presents an example, while §3.1 presents the iterative algorithm to identify $\mathcal{M}^\star$. Importantly, the maximum candidate set can be identified only based on information local to each vertex, thus at a low cost.

The Containment Rule - search space reduction for edit-distance k. Obs. 1 highlights that there exist natural dependencies among the matches for the prototypes that are at edit-distance one from each other (i.e., one edge difference). Since the prototypes at $k = \delta + 1$ distance are generated by removing one edge from one of the $k = \delta$ prototypes (as long as the prototype remains connected), the solution subgraph $\mathcal{G}^\star_{\delta,p}$ for a prototype p at distance $k = \delta$, is a subset of the solution subgraph $\mathcal{G}^\star_{\delta+1,q}$ for any prototype q derived by removing one edge from p (Fig. 3(a)). For approximate matching, this relationship introduces the opportunity to infer matches at distance $k = \delta$ from the union of all matches at distance $k = \delta + 1$ with guarantees of recall (no false negatives). This has two advantages: *(i)* the search for matches at level $k = \delta$ is performed on the reduced graph at level $k = \delta + 1$, and *(ii)* the result of some constraints already verified at distance $k = \delta + 1$ can be reused (without rechecking) at level $k = \delta$, thus leading to work reuse, as we discuss next.

Redundant Work Elimination. Viewing the search template as a set of constraints enables identifying substructures

that are common between prototypes and eliminating work that are duplicated when searching each prototype in isolation. For example, in Fig. 3(b), all the $k = 1$ and $k = 2$ prototypes share the 4-Cycle constraint. According to the containment rule (Obs. 1), in this example, if a vertex participates in a match for any $k = 1$ prototype with the 4-Cycle constraint, the same vertex must also participate in at least one match for at least one of the $k = 2$ prototypes that also has the 4-Cycle constraint. Therefore, for the same vertex in $\mathcal{G}$, such non-local constraints can be verified only once (for either $\mathcal{H}_0$ or a prototype) and this information can be reused in later searches - eliminating a large amount of potentially redundant work (this design artifact is evaluated in §5.4).

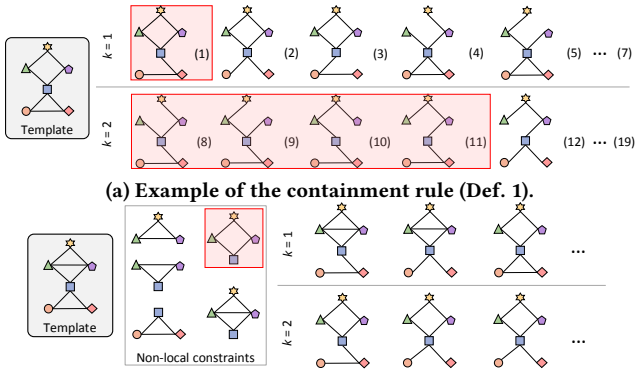
3.1 The Approximate Matching Pipeline

Alg. 1 is the high-level pseudocode of the approximate matching pipeline - the top-level procedure to search matches within k edit-distance of a template $\mathcal{H}_0$. The system iterates *bottom-up* (i.e., starting from prototypes at edit-distance k) over the set of prototypes $\mathcal{P}_k$ (we discuss optimization opportunities presented by a *top-down* approach in the next section). For each vertex $v \in \mathcal{G}$, the algorithm identifies the set of prototypes ($\rho(v_j)$ in Alg. 3) v participates in at least one match. Below we describe the key steps of the pipeline presented in Alg. 1. §4 presents in detail each functional component of the distributed infrastructure we have built.

***Prototype Generation* (Alg. 1, line #4).** From the supplied template $\mathcal{H}_0$, prototypes in $\mathcal{P}_k$ are generated through recursive edge removal: $k = \delta + 1$ distance prototypes are constructed from $k = \delta$ distance prototypes by removing one edge (while respecting the restriction that prototypes are connected graphs). If the template has mandatory edge requirements then, only the optional edges are are subjected to removal/substitution. We also perform isomorphism checks to eliminate duplicates. For prototypes with non-local constraints, each constraint is assigned an unique identifier. If two prototypes have the same non-local constraint, they also inherit its unique identifier (used by non-local constraint checking to track and ignore redundant checks for the same vertex in the background graph).

We note that as long as $\mathcal{H}_0$ includes all the edges the user may be interested in, the approximate search platform only needs to identify variations of $\mathcal{H}_0$ that can be obtained through edge deletion (i.e., edge addition is not required). Supporting other interesting search scenarios such as *wildcard* labels on vertices or edges [14], or edge 'flip' (i.e., swapping edges while keeping the number of edges constant) fits our pipeline's design and requires small updates.

***Maximum Candidate Set Generation* (Alg. 1, line #7).** This procedure first excludes the vertices that do not have a corresponding label in the template, then, iteratively, excludes the vertices that have all required neighbors excluded,



(a) Example of the containment rule (Def. 1).

(b) Work reuse, eliminating redundant constraints checks.

Figure 3: (a) A total of 19 prototypes at distance $k \leq 2$; 7 at distance $k = 1$ and 12 more at $k = 2$. The containment rule: prototypes (8) – (11) are generated from (1) through edge removal, so (1) can be searched within the union of the solution subgraphs of (8) – (11). (b) Non-local constraint reuse: for the same vertex, the highlighted 4-Cycle constraint can be checked for the $k = 2$ prototypes and the information can be directly used by the $k = 1$ prototypes.

and stops if no vertex is excluded at the end of an iteration. If the template has mandatory edge requirements then the vertices without all the mandatory neighbors specified in the search template are also excluded. As a key optimization to limit generated network traffic in later steps, the procedure also excludes edges to eliminated neighbors. Prototype search begins on the maximum candidate set.

Algorithm 1 Identify up to k Edit-Distance Matches

1: **Input:** background graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, template $\mathcal{H}_0(\mathcal{W}_0, \mathcal{F}_0)$, edit-distance k
2: **Output:** (i) per-vertex vector (of length $|\mathcal{P}_k|$) indicating v's prototype match (ρ); (ii) for each prototype $p \in \mathcal{P}_k$, the solution subgraph $\mathcal{G}^\star_{\delta,p}$
3: **Algorithm:**
4: generate prototype set $\mathcal{P}_k$ from $\mathcal{H}_0(\mathcal{W}_0, \mathcal{F}_0)$
5: for each prototype $p \in \mathcal{P}_k$, identify the non-local constraint set $\mathcal{K}_0$
6: $\delta = k$
7: $\mathcal{G}^\star_{\delta+1} \leftarrow$ MAX_CANDIDATE_SET($\mathcal{G}$, $\mathcal{H}_0$)
8: **do**
9: $\quad \mathcal{G}^\star_t \leftarrow \emptyset$
10: $\quad$ **for all** $p \in \mathcal{P}_\delta$ **do** ▷ alternatively, prototypes can be searched in parallel
11: $\quad\quad \mathcal{G}^\star_{\delta,p} \leftarrow$ SEARCH_PROTOTYPE($\mathcal{G}^\star_{\delta+1}$, p)
12: $\quad\quad \mathcal{G}^\star_t \leftarrow \mathcal{G}^\star_t \cup \mathcal{G}^\star_{\delta,p}$; output $\mathcal{G}^\star_{\delta,p}$ ▷ matches can be listed by enumerating in $\mathcal{G}^\star_{\delta,p}$
13: $\quad \mathcal{G}^\star_{\delta+1} \leftarrow \mathcal{G}^\star_t$; $\delta = \delta - 1$ ▷ distributed $\mathcal{G}^\star_{\delta+1}$ can be load rebalanced
14: **while** $\delta \geq 0$

Match Identification within Edit-Distance* k *through Iterative Search Space Reduction. We reduce the problem to finding matches at distance $k = \delta$ in the union of solution subgraphs of prototypes at distance $k = \delta + 1$, and iteratively decrease k until $k = 0$. This is reflected in lines #8 – #14 in Alg. 1. This iterative process performs two key operations:

(i) Prototype Match Identification. Each $k = \delta$ prototype p is searched by invoking the *SEARCH_PROTOTYPE* routine (Alg. 1, line #11): it identifies the vertices and edges in $\mathcal{G}$ that match p, updates the per-vertex match vector ρ accordingly

(Alg. 2, line#11), and generates the solution subgraph $\mathcal{G}^{\star}_{\delta,p}$, which can be used, for example, for match enumeration. *SEARCH_PROTOTYPE* performs exact matching by using local and non-local constraint checking routines, (Alg. 2, lines #6 and #9). (§4 provides details of how these primitives support the approximate matching pipeline.)

(ii) Search Space Reduction for $k = \delta$ Prototypes. Alg. 1, line #12 and #13, implement the containment rule. Logically, they compute the union of solution subgraphs of prototypes at distance $k = \delta + 1$. In terms of actual operations, this means just calculating the new bitmask over the match vector ρ that can query whether a vertex is a member of any distance $k = \delta + 1$ prototype. (The actual implementation does not make a new copy of the graph, it uses bit vector based per vertex/edge data structures to maintain various states and delete vertices/edges when they become obsolete.)

Our redundant work elimination technique brings further efficiency: each individual prototype match can take advantage of work reuse - we keep track of the non-local constraints that have been checked at level $k = \delta + 1$ and reuse the information at level $\delta \leq k$ (details in §4).

Algorithm 2 Search Routine for a Single Prototype

```
1: procedure SEARCH_PROTOTYPE (G*_{δ+1}, p)
2:     K_0 ← non-local constraint set of p
3:     G*_δ ← LOCAL_CONSTRAINT_CHECKING (G*_{δ+1}, p)
4:     while K_0 is not empty do
5:         pick and remove next constraint C_0 from K_0
6:         G*_δ ← NON_LOCAL_CONSTRAINT_CHECKING (G*_δ, p, C_0)
7:         if any vertex in G*_δ has been eliminated or
8:         has one of its potential matches removed then
9:             G*_δ ← LOCAL_CONSTRAINT_CHECKING (G*_δ, p)
10:    for all v_j ∈ G*_δ do
11:        update ρ(v_j) to indicate if v_j matches p
12:    return G*_δ
```

4 DISTRIBUTED IMPLEMENTATION

This section presents the system implementation on top of HavoqGT [26], an MPI-based framework with demonstrated scaling properties [45, 46]. Our choice for HavoqGT is driven by multiple considerations: *(i)* Unlike most graph processing frameworks that only support the Bulk Synchronous Parallel (BSP) model, HavoqGT has been designed to support *asynchronous* algorithms, well suited for non-local constraint verification/match enumeration, where high volume communication can be overlapped with computation; *(ii)* HavoqGT also offers efficient support to process scale-free graphs: the *delegate partitioned graph* distributes the edges of high-degree vertices across multiple compute nodes, crucial to scale to large graphs with skewed degree distribution; and *(iii)* An MPI-based implementation likely significantly more efficient than a Hadoop/Spark based solution [3, 21, 39, 57]. We note that our solution can be implemented within any other general purpose graph processing framework that exposes a *vertex-centric* API, e.g., Giraph [20], GraphLab [22] and GraphX [23], independent of the communication strategy used, bulk synchronous or asynchronous.

In HavoqGT, algorithms are implemented as vertex-callbacks: the user-defined *visit()* callback can access and update the state of a vertex. The framework offers the ability to generate events (a.k.a. 'visitors' in HavoqGT's vocabulary) that trigger a *visit()* callback - either on all graph vertices using the $do\_traversal()$ method, or for a neighboring vertex using the $push(visitor)$ method. (This enables asynchronous vertex-to-vertex communication, thus exchanging data between vertices). The asynchronous graph computation completes when all 'visitors' events have been processed, which is determined by distributed quiescence detection [69].

In a distributed setting, each process runs an instance of Alg. 1 and 2 on the distributed graph topology data. Alg. 2 invokes the two primitives that perform local and non-local constraint checking. This section first presents these routines in the vertex-centric abstraction of HavoqGT and the key state maintained by each vertex and its initialization (Alg. 3, also see §5.7) - the constraint checking routines in [51] have been extended to utilize these states, and then highlights the key functionality needed, to support approximate matching (§3.1), and the various optimizations implemented.

Algorithm 3 Vertex State and Initialization

```
1: set of possible matches in a prototype for vertex v_j: ω(v_j)        ▷ prototype state
2: set of matching neighbors in a prototype for vertex v_j: ω'(v_j)     ▷ prototype state
3: map of active edges of vertex v_j: ε(v_j) ← keys are initialized to adj*(v_j) ⊂ E*;
   the value field is a 8-bit long bitset, where individual bits indicate if the edge
   is active in the max-candidate set, and/or edit-distance, and/or in a prototype
   solution subgraph                                                    ▷ prototype state
4: set of non-local constraints vertex v_j satisfies: κ(v_j)            ▷ global state
5: vector of prototype matches for v_j: ρ(v_j)                          ▷ global state
```

Local Constraint Checking (LCC) is implemented as an iterative process. Alg. 4 presents the high-level algorithm and the corresponding $visit()$ callback. Each iteration initiates an asynchronous traversal by invoking the $do\_traversal()$ method, and as a result, each active vertex receives a visitor. In the triggered $visit()$ callback, if the label of an active vertex v_j in the graph is a match for the label of any vertex in the template, it creates visitors for all its active neighbors in $\varepsilon(v_j)$. When a vertex v_j is visited, it verifies whether the sender vertex v_s satisfies one of its own (i.e., v_j's) local constraints. By the end of an iteration, if v_j satisfies all the template constraints, i.e, it has neighbors with the required labels, it stays active for the next iteration. Edge elimination excludes two categories of edges: first, the edges to neighbors, $v_i \in \varepsilon(v_j)$ from which v_j did not receive a message, and second, the edges to neighbors whose labels do not match the labels prescribed in the adjacency structure of the corresponding template vertex (or vertices) in $\omega(v_j)$. Iterations continue until no vertex or edge is marked inactive.

Algorithm 4 Local Constraint Checking

```
1: procedure LOCAL_CONSTRAINT_CHECKING (G*(V*, E*), p)
2:     do
3:         do_traversal(); barrier
4:         for all v_j ∈ V* do
5:             if v_j does not meet local constraints of at least one vertex of p then
6:                 remove v_j from G*                                  ▷ vertex eliminated
7:             else if a neighbor v_i ∈ ε(v_j) does not satisfy requirements of p then
8:                 remove v_i from ε(v_j) ⊂ E*)                        ▷ edge eliminated
9:     while vertices or edges are eliminated
10: procedure VISIT(G*(V*, E*), vq)        ▷ vq - the distributed message queue
11:     for all v_i ∈ ε(v_j) do            ▷ v_j is the vertex that is being visited
12:         vis ← LCC_VISITOR(v_i, v_j, ω(v_j))
13:         vq.push(vis)                                   ▷ triggers visit() for v_i
```

Non-local Constraint Checking (NLCC) iterates over $\mathcal{K}_0$, the set of non-local constraints to be checked, and validates each $C_0 \in \mathcal{K}_0$ one at a time. NLCC leverages a *token passing* approach. Alg. 5 presents the general solution (including the *visit*() callback) to verify a single constraint: *tokens* are initiated through an asynchronous traversal by invoking the *do_traversal*() method. Each active vertex $v_j \in \mathcal{G}^\star$ that is a potential match for the template vertex at the head of a 'path' C_0, broadcasts a token to all its active neighbors in $\varepsilon(v_j)$. When an active vertex v_j receives a token, if all requirements are satisfied, v_j sets itself as the forwarding vertex (v_j is added to t), increments the hop count r, and broadcasts the token to all its active neighbors. If any of the constraints are violated, v_j drops the token. If r is equal to $|C_0|$ and v_j is a match for the template vertex at the tail of C_0, v_j is marked as it meets requirements of C_0 (Alg. 5, lines #14 and #15).

Caching the Result of NLCC – Enabler for Redundant Work Elimination. We reuse the result of constraint checking - a vertex in the background graph that satisfies a non-local constraint for a $k = \delta + 1$ prototype does not need to verify the same constraint in the subsequent $\delta \leq k$ prototypes that share the constraint, hence, avods redundant work (Alg. 5, line #9). This optimization is crucial for cyclic patterns that have dense and highly concentrated matches. We demonstrate the impact of this optimization in §5.4.

Search Space Pruning in the Bottom-Up Mode. In §2 and §3, we established that, when performing a bottom-up search (i.e., starting from the furthest distance prototypes) matches at distance $k = \delta$ can be computed from the union of all matches at distance $k = \delta + 1$. The distributed infrastructure implements this functionality in a simple, yet efficient manner (Alg. 1, lines #12 and #13): we use a hash-based data structure to store the distributed graph to allow fast modifications (i.e., vertex and edge deletion) as well as aggregate the solution subgraphs at distance $k = \delta + 1$ (on which $k = \delta$ distance matches are searched).

Load Balancing. Load imbalance issues are inherent to problems involving irregular data structures, such as graphs. For our pattern matching solution, load imbalance is caused by two artifacts: first, over the course of execution, our solution causes the workload to mutate (as we prune away vertices and edges), and second, nonuniform distribution of matches: the vertices and edges that participate in the matches may reside in a small, potentially concentrated, part of the background graph. To address these issues, we can rebalance/reload a pruned, max-candidate set or intermediate graph ($\mathcal{G}^\star_{\delta+1}$), before searching the $k = \delta$ prototypes (Alg. 1, line #13). We checkpoint the current state of execution, and reload only the set of active vertices and edges that participate in at least one of the $k = \delta + 1$ prototypes. We can reload on the same or fewer processors, and reshuffle vertex-to-processor assignment to evenly distribute vertices and edges across processing cores. $k = \delta$ prototype searches are then resumed on the rebalanced distributed graph. The effectiveness of reshuffling is evaluated in Fig. 9(a), while reloading on a smaller processor set in Fig. 8, and §5.4.

Algorithm 5 Non-local Constraint Checking

```
1: procedure NON_LOCAL_CONSTRAINT_CHECKING(G*(V*, E*), p, C_0)
2:     do_traversal(); barrier
3:     for all v_j ∈ V* that initiated a token do
4:         if v_j violates C_0 then
5:             remove this match from ω(v_j) and if ω(v_j) = ∅, remove v_j from G*
   ▷ vertex eliminated
6: visitor state: token - a tuple (t, r) where t is an ordered list of vertices that have
   forwarded the token and r is the hop-counter; t_0 ∈ t is the token initiator
7: procedure VISIT(G*(V*, E*), vq)
8:     for all v_i ∈ ε(v_j) do                    ▷ v_j is the vertex that is being visited
9:         if token = ∅ and v_j matches the first entry in C_0 and C_0 ∉ κ(v_j) then
10:            t.add(v_j);   r ← 1;   token ← (t, r)
11:        else if token.r < |C_0| and v_j matches the token.r-th entry in C_0
   then
12:            token.t.add(v_j);   token.r ← token.r + 1
13:        else if token.r = |C_0| and v_j matches the token.r-th entry in C_0
   then
14:            κ(v_j).insert(C_0);   return     ▷ globally identify v_j as it meets
   requirments of C_0
15:        else return                                          ▷ drop token
16:        vis ← NLCC_VISITOR(v_i, token);   vq.push(vis)
```

Multi-level Parallelism. The implementation offers multiple levels of parallelism: in addition to vertex-level parallelism (i.e., vertices check constraints in parallel), the infrastructure also enables searching prototypes in parallel (Alg. 1, line #10): prototypes at distance $k = \delta$ can be searched in parallel by replicating the max-candidate set (or the distance $k = \delta + 1$ pruned graph) on multiple (potentially smaller) deployments. Fig. 8 and 9(b), and §5.4 evaluate the impact of this design artifact.

Match Enumeration and Counting Optimization. Given the containment rule, and since a $k = \delta + 1$ prototype is a direct descendent of a $k = \delta$ prototype (see §2), edit-distance based matching presents the opportunity for reusing results of $k = \delta+1$ match enumeration for identifying $k = \delta$ matches: a $k = \delta$ prototype match can be identified from the already computed $k = \delta + 1$ matches by extending the resulting matches by one edge (instead of repeating the search for all edges - evaluated in §5.4).

Top-Down Search Mode. Alg. 1 presents the bottom-up approch. An alternative is to perform the search in a top-down manner: the system initially searches for exact matches to the full template and extends the search by increasing the edit-distance by one until a user-defined condition is met. Our implementation also supports this search mode, with small additions to Alg. 1. Due to lack of space we avoid details, yet §5.5 evaluates a use case of this search mode.

5 EVALUATION

We present *strong* (§5.2) and *weak* (§5.1) scaling experiments on massive real-world and synthetic graphs. We demonstrate the ability of our system to support patterns with arbitrary topology and scale to 1,000+ prototypes. We evaluate the effectiveness of our design choices and optimizations (§5.4). We highlight the use of our system in the context of realistic data analytics scenarios (§5.5). Finally, we compare our solution with the sate-of-the-art system, Arabesque [63] (§5.6), as well as the naïve approach (§5.3).

Testbed. The testbed is a 2,634 nodes cluster equipped with Intel Omni-Path interconnect. Each node has two 18-core Intel Xeon E5-2695v4 @2.10GHz processors and 128GB of memory [49]. We run 36 MPI processes per node.

Datasets. The table summarizes the main characteristics of the datasets used for evaluation and shows their storage requirements. For all graphs, we created undirected versions.

	Type	$\|\mathcal{V}\|$	$2\|\mathcal{E}\|$	d_{max}	d_{avg}	d_{stdev}	Size
WDC [54]	Real	3.5B	257B	95M	72.3	3.6K	2.5TB
Reddit [50]	Real	3.9B	14B	19M	3.7	483.3	460GB
IMDb [32]	Real	5M	29M	552K	5.8	342.6	581MB
CiteSeer [63]	Real	3.3K	9.4K	99	3.6	3.4	741KB
Mico [63]	Real	100K	2.2M	1.4K	22	37.1	36MB
Patent [63]	Real	2.7M	28M	789	10.2	10.8	480MB
YouTube [63]	Real	4.6M	88M	2.5K	19.2	21.7	1.4GB
LiveJournal [56]	Real	4.8M	69M	20K	17	36	1.2GB
R-MAT up to Scale 35 [11]	Synth.	34.4B	1.1T	222M	32	3.5K	17TB

The **Web Data Commons (WDC)** graph is a webgraph whose vertices are webpages and edges are hyperlinks. To create vertex labels, we extract the top-level domain names from the webpage URLs, e.g., *.org* or *.edu*. If the URL contains a common second-level domain name, it is chosen over the top-level domain name. For example, from *ox.ac.uk*, we select *.ac* as the vertex label. A total of 2,903 unique labels are distributed among the 3.5B vertices in the background graph. We curated the **Reddit (RDT)** social media graph from an open archive [50] of billions of public posts and comments from Reddit.com. Reddit allows its users to rate (upvote or downvote) others' posts and comments. The graph has four types of vertices: *Author, Post, Comment* and *Subreddit* (a category for posts). For *Post* and *Comment* type vertices there are three possible labels: *Positive, Negative,* and *Neutral* (indicating the overall balance of positive and negative votes) or *No Rating*. An edge is possible between an *Author* and a *Post*, an *Author* and a *Comment*, a *Subreddit* and a *Post*, a *Post* and a *Comment* (to that *Post*), and between two *Comments* that have a parent-child relationship. The **International Movie Database (IMDb)** graph was curated from the publicly available repository [32]. The graph has five types of vertices: *Movie, Genre, Actress, Actor* and *Director*. The graph is bipartite - an edge is only possible between a *Movie* type vertex and a *non-Movie* type vertex. We use the smaller, unlabeled, **CiteSeer**, **Mico**, **Patent**, **YouTube** and **LiveJournal** graphs primarily to compare published results in [63]. The synthetic **R-MAT** graphs exhibit approximate power-law degree distribution [11]. These graphs were created following the Graph 500 [24] standards: 2^{Scale} vertices and a directed edge factor of 16. For example, a Scale 30 graph has $|\mathcal{V}| = 2^{30}$ and $2|\mathcal{E}| \approx 32 \times 2^{30}$ (since we create an undirected version). As we use the R-MAT graphs for weak scaling experiments, we aim to generate labels such that the graph structure changes little as the graph scales. To this end, we leverage vertex degree information and create vertex labels as: $\ell(v_i) = \lceil \log_2(d(v_i) + 1) \rceil$. This, for instance, for the Scale 35 graph results in 30 unique vertex labels.

Search Templates. To stress our system, we: *(i)* use templates based on patterns naturally occurring in the background graphs; *(ii)* experiment with both rare and frequent patterns; *(iii)* explore search scenarios that lead to generating 100+ and 1,000+ prototypes (WDC-3 and WDC-4 in Fig. 5); *(iv)* include in the template vertex labels that are among the most frequent in the respective graphs; and *(v)* similar to [63], we use unlabeled patterns for counting motifs (§5.6).

Experimental Methodology. The performance metric for all experiments is the time-to-solution for searching all the prototypes of a template - for each matching vertex, produce the list of prototypes it participates in. The time spent transitioning and resuming computation on an intermediate pruned graph and load balancing are included in the reported time. All runtime numbers provided are averages over 10 runs. For weak scaling experiments, we do not present scaling numbers for a single node as this does not involve network communication. For strong scaling experiments, the smallest scale uses 64 nodes, as this is the lowest number of nodes that can load the WDC graph and support algorithm state. *We label our technique as* HGT *wherever necessary.*

5.1 Weak Scaling Experiments

To evaluate the ability to process massive graphs, we use weak scaling experiments and the synthetic R-MAT graphs up to Scale 35 (~1.1T edges), and up to 256 nodes (9,216 cores). Fig. 4 shows the search template, RMAT-1, and the runtimes. RMAT-1 has up to $k = 2$ (before getting disconnected), generating a total of 24 prototypes; 16 of which at $k = 2$. On average, ~70% of the time is spent in the actual

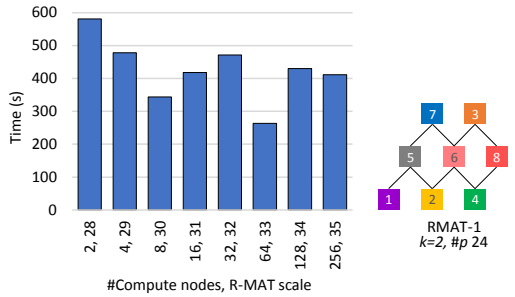


Figure 4: Runtime for weak scaling experiments (left) for the the RMAT-1 pattern (right). The furthest edit-distance searched (k) and prototype count (#p) are also shown. The X-axis labels present the R-MAT scale (top) and the compute node count (bottom). A flat line indicates perfect weak scaling. The template labels used are the most frequent and cover ~45% of the vertices in the backgorund graph. The standard deviation of runtime is less than 5% of the average.

search, while remaining 30% in infrastructure management - switching from $k = \delta + 1$ to $k = \delta$ pruned graph and load balancing. In spite of the random nature of R-MAT generation, we see mostly consistent scaling in runtime except for the Scale 33 graph for which the RMAT-1 pattern happens to be very rare, which explains the faster search time. (Scale 33 has 17.5M matches compared to 64M in Scale 32 and 73M in Scale 34, 3.7× and 4.2× fewer, than in the respective graphs. This is partly because the vertex label '8' is very rare in Scale 33 - less than 1% of the vertices have this label. For Scale 34 and 35, for example, the ratio is 3.2% and 2.3%, respectively.)

5.2 Strong Scaling Experiments

Fig. 6 shows the runtimes for strong scaling experiments when using the real-world WDC graph on up to 256 nodes (9,216 cores). Intuitively, pattern matching on the WDC graph is harder than on the R-MAT graphs as the WDC graph is denser, has a highly skewed degree distribution, and importantly, the high-frequency labels in the search templates also belong to vertices with high neighbor degree. We use the patterns WDC-1, 2 and 3 in Fig. 5. To stress the system we have chosen search templates that generate tens to hundreds of prototypes (WDC-3 has 100+, up to $k = 4$, prototypes). These patterns have complex topology, e.g., multiple cycles sharing edges, and rely on expensive non-local constraint checking to guarantee no false positives. In Fig. 6, time-to-solution is broken down by edit-distance level, and indicates time spent to obtain the max-candidate set, and infrastructure management, separately. We see moderate scaling for both WDC-1 and WDC-2, up to 2.7× and 2×, respectively, with the furthest k prototypes scaling a lot better as they are mostly acyclic. Since WDC-3 has 100+ prototypes, we leverage the opportunity to search multiple prototypes in parallel: given that the candidate set is much smaller than that original background graph (~138M vertices), we replicate it on smaller eight node deployments (this involves

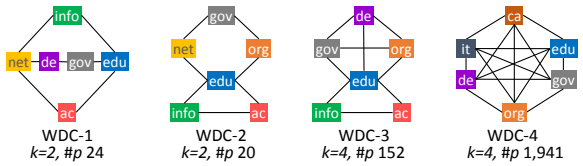


Figure 5: WDC patterns using top/second-level domain names as labels. The labels selected are among the most frequent, covering ~21% of the vertices in the WDC graph: *org* covers ~220M vertices, the 2nd most frequent after *com*; *ac* is the least frequent, still covering ~4.4M vertices. The furthest edit-distance searched (k) and prototype count (#p) are also shown.

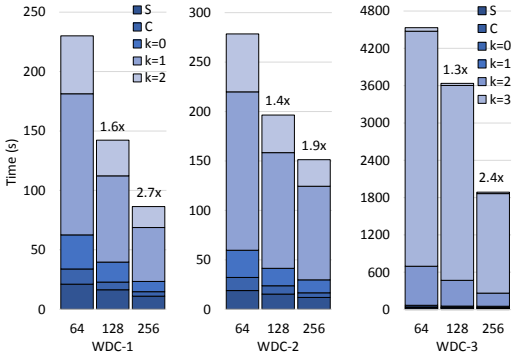


Figure 6: Runtime for strong scaling experiments broken down by edit-distance level, for the WDC-1, 2 and 3 patterns (Fig. 5). Max-candidate set generation time (C) and infrastructure management overhead (S) are also shown. The top row of X-axis labels represent the number of compute nodes. Speedup over the 64 node configuration is shown on top of each stacked bar plot. The standard deviation of runtime is less than 5% of the average.

repartitioning the pruned graph and load balancing) and search multiple prototypes in parallel. For example, WDC-3 has 61, $k = 3$ prototypes; on 64 nodes they can be searched in eight batches (each batch running eight parallel search instances). We observe 2.4× speedup on 256 nodes.

5.3 Comparison with the Naïve Approach

We study the performance advantage of our solution over a naïve approach which generates all prototypes and searches them independently in the background graph. Fig. 7 compares time-to-solution of our technique with the naïve approach for various patterns and graphs: it shows an 3.8× average speedup achieved by our solution over the naïve approach. (The reported time for HGT includes time spent in search and infrastructure management.)

To further explain performance, we study the runs for the RMAT-1 (Scale 34) and WDC-3 patterns at finer detail (both on 128 nodes). For RMAT-1, on average, individual prototype search is 6× faster in HGT. However, the max-candidate set generation and load balancing the pruned graph(s) has additional overhead, which is ~30% of the total time in this case. The max-candidate set has ~2.7B vertices and ~5.9B edges, and ~10% of the total time is spent in load balancing this intermediate graph, hence, the resulting 3.8× speedup

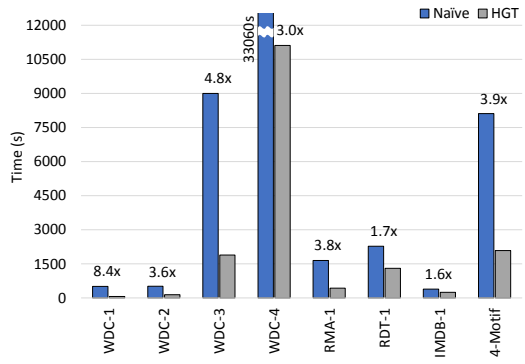


Figure 7: Runtime comparison between the naïve approach and HGT for various patterns and graphs. Speedup over the naïve approch is shown on top of respective bars. For better visibility, we limit the Y-axis and show the Y-axis label (larger than the axis bound) for WDC-4, for the naïve case. RMAT-1, IMDB-1 and 4-Motif (on the Youtube graph) also include time for explicit match counting. For the the rest, we report time to identify union of all matches. The standard deviation of runtime is less than 5% of the average.

over the naïve approach (Fig. 7). There are 73.6M matches at distance $k = 2$ and none at $k < 2$.

In Fig. 8, runtime for WDC-3 (which has 100+ prototypes) is broken down to per edit-distance level. The figure shows how optimizations improve search performance over the naïve approach (when using 128 compute nodes.). The max-candidate set for this pattern is smaller relative to RMAT-1 (Scale 34), yet still large - 138M vertices and 1B edges); therefore, the infrastructure management overhead is lower as well. The furthest edit-distance ($k = 4$) pruned graph also has only 15M vertices. Infrastructure management and load balancing accounts for less than 1% of the total time. For the most optimized case, when prototypes are searched in parallel, each on an eight node deployment, the total gain in runtime is ~3.4× over the naïve solution.

5.4 Impact of Optimizations

Redundant Work Elimination. One key optimization our solution incorporates is work recycling, in particular reuse the result of constraint checking to avoid redundant checks (details in §4). This optimization is crucial for cyclic patterns that have dense and highly concentrated matches in the background graph. This alone offers 2× speedup for some WDC-3 levels (Fig. 8, notice the improvement for $k = 2$ and $k = 1$ in scenario Y) and 1.5× speedup for IMDB-1 (Fig. 7), over the respective naïve runs. The gain is due to reduction in number of messages that are communicated during NLCC; for WDC-3 the reduction is 3.5× and for IMDB-1 it is 6.7×.

Constraint and Prototype Ordering. We use a simple heuristic to improve performance of non-local constraint checking: each walk is orchestrated so that vertices with lower frequency labels are visited early (similar to degree-based ordering used by triangle counting [62]). We explore a

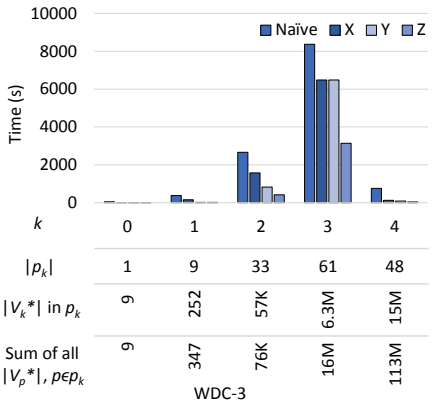
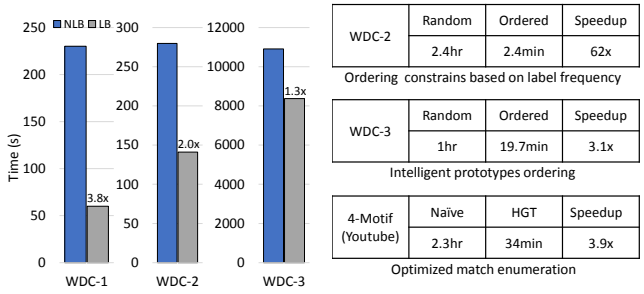


Figure 8: Runtime broken down by edit-distance level for WDC-3. Four scenarios are compared: *(i)* the naïve approach (§5.3); *(ii)* X - the bottom-up technique with search space reduction only; *(iii)* Y - the bottom-up technique including redundant work elimination (§5.4); and *(iv)* Z - the bottom-up technique with load balancing and relaunching processing on a smaller eight node deployment, enabling parallel prototype search (§4). X-axis labels: k is the edit-distance; (center top) p_k is the set of prototypes at distance k; (center bottom) the size of matching vertex set ($V_k^\star$) at distance k (i.e., number of vertices that match at least one prototype p_k); (bottom) total number of vertex/prototype labels generated at distance k.

second optimization opportunity with respect to work ordering - when searching prototypes in parallel, the performance is improved when the runs for the most expensive prototypes are overlapped. Fig. 9(b) summarizes the impact of these two optimization strategies. For prototype ordering, we manually reorder the prototypes (for maximum overlap of expensive searches) based on the knowledge from a previous run, thus this figure offers an upper bound for the performance gain to obtain with heuristics that aim to project prototype cost.

Optimized Match Enumeration. We evaluate the advantage of the match enumeration/counting optimization for edit-distance based matching presented in §4. The 4-Motif pattern (6 prototypes) has 200B+ instances in the unlabeled Youtube graph. The optimized match enumeration technique enables ~3.9× speedup (Fig. 9(b), bottom table).

Load Balancing. We examine the impact of load balancing (presented in §4) by analyzing the runs for WDC-1, 2 and 3 patterns (as real-world workloads are more likely to lead to imbalance than synthetically generated load.) Fig. 9(a) compares the performance of the system with and without load balancing. For these examples, we perform workload rebalancing once, after pruning the background graph to the max-candidate set, which, for the WDC-1, 2 and 3 patterns, have 33M, 22M and 138M vertices, respectively (2–3 orders of magnitude smaller than the original WDC graph). Rebalancing improves time-to-solution by 3.8× for WDC-1, 2× for WDC-2 and 1.3× for WDC-3. Load balancing the pruned intermediate graph takes ~22 seconds for WDC-3 (Fig. 6).



(a)

WDC-2	Random	Ordered	Speedup
	2.4hr	2.4min	62x

Ordering constrains based on label frequency

WDC-3	Random	Ordered	Speedup
	1hr	19.7min	3.1x

Intelligent prototypes ordering

4-Motif (Youtube)	Naïve	HGT	Speedup
	2.3hr	34min	3.9x

Optimized match enumeration

(b)

Figure 9: (a) Impact of load balancing on runtime for the WDC-1, 2 and 3 patterns (Fig. 5). We compare two cases: without load balancing (NLB) and with load balancing through reshuffling on the same number of compute nodes (LB). Speedup achieved by LB over NLB is shown on the top of each bar. (b) Impact of ordering vertex labels in the increasing order of frequency for non-local constraint checking (top), impact of intuitive prototype ordering when searching them in parallel (middle), and impact of our match enumeration optimizations for approximate matching (bottom).

Reloading on a Smaller Deployment and Parallel Prototype Search. Once the max-candidate set has been computed, we can take advantage of this smaller input set: we support reloading the max-candidate set on one (or more) smaller deployment(s), and running multiple searches in parallel. We explore the performance space for two optimization criteria: *(i)* minimizing time-to-solution - all nodes continue to be used but different prototypes may be searched in parallel, each on smaller deployments (top two rows in the table below); *(ii)* minimizing the total CPU Hour [67] - a smaller set of nodes continues to be used, and prototypes are searched one at a time (bottom two rows). The following table presents the results: it compares the overhead of CPU Hour over running on a two node deployment for the WDC-3 pattern (Fig. 5). It also lists the runtime for searching prototypes in parallel given a node budget (128 in this example) - a smaller deployment can be more efficient (due to improved locality). Here, the processing rate on the two node deployment is too slow to yield a notable advantage over the four node deployment.

	#Compute nodes	128	8	4	2
Parallel	Time (min)	124	60	12	15
prototype search	Speedup over 128 nodes	N/A	2.1×	10.3×	8.3×
Sequential	CPU Hour	9,531	588	204	192
prototype search	Overhead w.r.t. 2 nodes	50×	3×	1.1×	N/A

5.5 Example Use Cases

To show how our approach can support complex data analytics scenarios we present three use cases: *(i)* a query that attempts to uncover suspicious activity in the Reddit dataset and uses optional and mandatory edges; *(ii)* a similar search in IMDb; and *(iii)* an example of a top-down exploratory search using the WDC-4 pattern. Fig. 10 presents the search templates and Fig. 7 shows the runtimes.

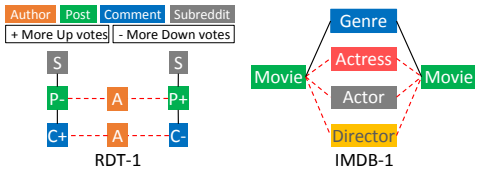


Figure 10: The Reddit (RDT-1) and IMDb (IMDB-1) templates (details in §5.5); optional edges are shown in red, broken lines, while mandatory edges are in solid black.

Social Network Analysis. Today's user experience on social media platforms is tainted by the existence of malicious actors such as bots, trolls, and spammers - highlights the importance of detecting unusual activity patterns that may indicate a potential attack. The RDT-1 query: Identify users with an adversarial poster-commenter relationship. Each author (A) makes at least two posts or two comments, respectively. Comments to posts, with more upvotes (P+), have a balance of negative votes (C-) and comments to posts, with more downvotes (P-), have a positive balance (C+). The posts must be under different subreddits (S). The user is also interested in the scenarios where both the posts and/or the comments were not by the same author, i.e., a valid match can be missing an author-post or an author-comment edge (Fig. 10). The query has a total of five prototypes and over 708K matches (including 24K precise).

Information Mining. IMDB-1 represents the following approximate query: find all the actress, actor, director, 2× movies tuples, where at least one individual has the same role in two different movies between 2012 and 2017, and both movies fall under the genre *Sport*. The query has a total of seven prototypes and 303K matches (including 78K precise).

Exploratory Search. We present an exploratory search scenario: the user starts form an undirected 6-Clique (WDC-4 in Fig. 5) and the search is progressively relaxed until matches are found. The search is configured to return all matches for all prototypes at the distance k for which the first matches are found. No match is found until $k = 4$, where only 144 vertices participate in matches. The search involves sifting through 1,941 prototypes in about three hours; 1,365 prototypes at distance $k = 4$, for which average search time is 5.7 seconds.

5.6 Comparison with Arabesque

Arabesque is a framework offering precision and recall guarantees, implemented on top of Apache Spark [59] and Giraph [20]. Arabesque provides an API based on the Think Like an Embedding (TLE) paradigm, to express graph mining algorithms and a BSP implementation of the embedding search engine. Arabesque replicates the input graph on all worker nodes, hence, the largest graph scale it can support is limited by the size of the memory of a single node. As Teixeira et al. [63] showed Arabesque's superiority over other systems: G-Tries [52] and GRAMI [15] (we indirectly compare with these two systems as well).

For the comparison, we use the problem of *counting network motifs* in an unlabeled graph (the implementation is available with the Arabesque release). Network motifs are *connected pattern of vertex induced embeddings that are non-isomorphic*. For example, three vertices can form two possible motifs - a chain and a triangle, while up to six motifs are possible for four vertices. Our solution approach lends itself to motif counting: from the maximal-edge motif (e.g., a 4-Clique for four vertices), through recursive edge removal, we generate the remaining motifs (i.e., the prototypes). Then we use our approximate matching system to search and count matches for all the prototypes. The following table compares results of counting three- and four-vertex motifs, using Arabesque and our system (labeled HGT), using the same real-world graphs used for the evaluation of Arabesque in [63]. We deploy both our solution and Arabesque on 20 compute nodes; the same scale as in [63]. Note that Arabesque users have to specify a purpose-built algorithm for counting motifs, whereas ours is a generic solution, not optimized to count motifs only.

	3-Motif		4-Motif	
	Arabesque	HGT	Arabesque	HGT
CiteSeer	9.2s	0.02s	11.8s	0.03s
Mico	34.0s	11.0s	3.4hr	57min
Patent	2.9min	1.6s	3.3hr	2.3min
Youtube	40min	12.7s	7hr+	34min
LiveJournal	11min	10.3s	OOM	1.3hr

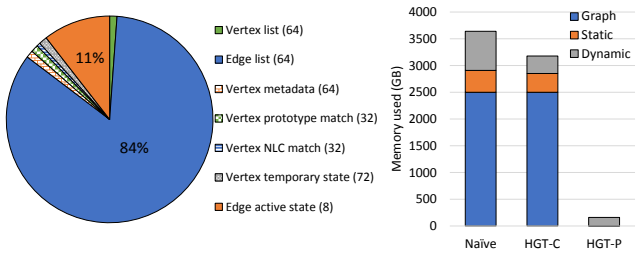
Our system was able to count all the motifs in all graphs; it took a maximum time of 1.3 hours to count four vertex motifs in the LiveJournal graph. Arabesque's performance degrades for larger graphs and search templates: it was only able to count all the motifs in the small CiteSeer (~10K edges) graph in <1 hour. For 4-Motif in LiveJournal, after running for about 60 minutes, Arabesque fails with the out of memory (OOM) error. (We have been in contact with Arabesque authors to make sure we best employ their system. Also, the observed performance is comparable with the Arabesque runtimes recently reported in [33] and [68]).

5.7 Network and Memory Analysis

Message Analysis. The following table shows number of messages for two scenarios, naïve (§5.3) and our optimized technique (labeled HGT), using the WDC-2 pattern (Fig. 5) on 64 compute nodes. The table also lists the fraction of remote messages, and for HGT, the fraction of total messages exchanged during max-candidate set generation. HGT shows 16.6× better message efficiency leading to 3.6× speedup.

	Naïve	HGT	Improvement
Total number of messages	647×10^9	39×10^9	16.6×
% of remote messages	88.5%	89.8%	16.3× (raw)
% due to max-candidate set	N/A	82.5%	N/A
Time	8.6min	2.4min	3.6×

Memory Usage. Fig. 11(a) shows, for the WDC graph, the relative amount of memory used to store the background graph (in CSR format) and different algorithm related state.



(a) (b)

Figure 11: (a) For the WDC graph, relative memory required by the topology, and the various data structures allocated by the algorithm. For each entry, the datatype size (in bits) is shown in the legend. The example assumes there are at most 32 prototypes, 32 vertices per prototype and 32 non-local constraints. (b) Comparison of peak memory usage (in gigabyte) for naïve and HGT, broken down by category: graph topology, statically-, and dynamically-allocated data structures (details in §5.7).

Roughly 86% of the total memory is required to store the graph topology (i.e., edge and vertex lists, and vertex labels), while an additional 14% for maintaining algorithm states (~265GB) which include a statically allocated part needed to support the algorithms we describe (e.g., per-vertex prototype match vectors), message queues, and state maintained by the HavoqGT infrastructure (e.g., for each vertex, the corresponding MPI process identifier).

Fig. 11(b) compares, for WDC-2, using 64 compute nodes, the *total* (i.e., cluster-wide) *peak memory usage* of the naïve approach and HGT, broken down to three categories: the topology information; the 'static' - memory allocated for states (in Fig. 11(a)) before search begins; and the 'dynamic' - states created during the search (primarily message queues). Results for max-candidate set generation (HGT-C), and prototype search (HGT-P), starting from the max-candidate set, are shown separately. Here, the max-candidate set is three orders of magnitude smaller than the original WDC graph; therefore, for HGT-P, the 'dynamic' state dominates the memory usage; the number of messages required for non-local constraint checking is exponential, yet HGT-P improves the 'dynamic' state size by ~4.6× over the naïve approach.

Impact of Locality. Fig. 12 presents results of an experiment that explores the impact of mapping a configuration that keeps the graph partitioning (this is also the number of MPI processes used) constant while varying the number of nodes used. The workload is the WDC-2 template. We run a fixed number of MPI processes (768) but vary the number of nodes: from 16 nodes (48 processes per node) up to 768 nodes (one process per node). This variation has two key effects: On the one side, the ratio of remote communication varies; e.g., with 768 nodes, all inter-process communications are remote. On the other side, the computational load on each node varies; e.g., the configuration with 48 processes per

node, although it has the highest ratio of node-local communication, generates the highest computational load as it oversubscribes the number of cores (there are only 36 cores per node). Fig. 12 shows that, depending on the configuration, the bottleneck can be network communication or compute throughput at the node level. We note that asynchronous processing does a good job of hiding network overhead even when the locality is low (e.g., 6 and 3 processes per node).

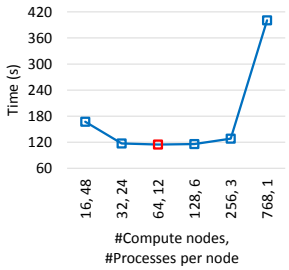


Figure 12: Impact of varying the number of nodes while maintaining constant the number of MPI processes. The best performing setup is shown using a red marker. The same partitioning among processes is used across all experiments.

6 RELATED WORK

The literature of graph processing in general [20, 22, 23, 29, 38, 61], and pattern matching in particular [7, 17, 40, 42, 66, 75], is rich. This section focuses on closely related work.

Graph Similarity Estimators. Besides Edit-distance [9], there exist other methods to estimate *graph similarity*, e.g., Maximum Common Subgraph (MCS) [8], Graph Kernel [30], and solutions based on capturing statistical significance [14]. Our choice for edit-distance as the similarity metric is motivated by three observations: edit-distance *(i)* is a widely adopted similarity metric easy to understand by users; *(ii)* can be adapted to various use cases seen in practice; and *(iii)* can support efficient approximate searches, while providing precision and recall guarantees [9], as shown in this paper.

Approximate Matching. The non-polynomial nature of exact matching has lead to the development of a plethora of *approximate matching* heuristics. Graph edit-distance has been widely used for pattern approximation [18, 41, 53, 73]. SAPPER [73] and the solution in [71], reduce the approximate matching problem to finding exact matches for similar subgraphs, however, evaluated using relatively small synthetic graphs. Random walk has been used for sampling the search space, e.g., G-Ray [64] and ASAP [33], or approximating the candidate matches before enumerating them [6, 7]. Frequent subgraph indexing has been used by some to reduce the number of *join* operations, e.g., C-Tree [27] and SAPPER [73]. Unfortunately, for a billion-edge graph, this approach is infeasible [60]. The Graph Simulation [28] family of algorithms have quadratic/cubic time complexity and have been demonstrated for emerging matching problems when large graphs are involved [17, 19, 35, 37]. The *color-coding* algorithm was originally designed to approximate the number of *treelets* in protein-protein interaction networks [2].

Distributed Solutions. Here, we focus on projects that targets distributed pattern matching and demonstrate it at some scale. The largest scale (107 billion edge graph) is offered by Plantenga's [48] MapReduce implementation of the algorithm originally proposed in [7]. The system can find approximate matches for a restricted class of small patterns. SAHAD [74] and FASCIA [58] are distributed implementations of the color-coding algorithm [2] for approximate treelet counting. Chakaravarthy et al. [10] extended the color-coding algorithm to (a limited set of) patterns with cycles, however, demonstrated performance on much smaller graphs (compared to our work). Although, Arabesque [63] and QFrag [56] (based on Apache Spark [59] and Giraph [23]) outperforms most of their competitors, they replicate the graph in the memory of each node, which limits their applicability to small relatively graphs only. PGX.D/Async [55] offers flow control with a deterministic guarantee of search completion under a finite amount of memory, however, was demonstrated at a much smaller scale than in this paper. ASAP [33] (based on Apache Spark and GraphX [23]) enables trade-off between the result accuracy and time-to-solution; it uses Chernoff bound analysis to control result error [44]. ASAP is only able to approximate match counts and does not offer precision and recall guarantees.

7 CONCLUSION

We have presented an efficient distributed algorithmic pipeline to identify approximate matches in large-scale metadata graphs. Our edit-distance based solution builds on a number of novel observations for identifying all matches within a fixed edit-distance. Our technique exploits natural dependencies among the matches that are at edit-distance one from each other and offers two design mechanisms to achieve scalability and efficiency: *(i)* search space reduction and *(ii)* redundant work elimination. We implement this solution on top of HavoqGT and demonstrate scalability using up to 257 billion edge real-world and 1.1 trillion edge synthetic R-MAT graphs, on up to 256 nodes (9,216 cores). Our solution lends itself to efficient computation and approximate pattern discovery in practical scenarios, and comfortably outperforms the best known work when solving the same search problem.

ACKNOWLEDGEMENT

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-806003). Experiments were performed at the Livermore Computing facility.

REFERENCES

[1] Charu C. Aggarwal and Haixun Wang (Eds.). 2010. *Managing and Mining Graph Data*. Advances in Database Systems, Vol. 40. Springer. https://doi.org/10.1007/978-1-4419-6045-0

[2] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S. Cenk Sahinalp. 2008. Biomolecular Network Motif Counting and Discovery by Color Coding. *Bioinformatics* 24, 13 (July 2008), i241–i249. https://doi.org/10.1093/bioinformatics/btn163

[3] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. 2017. Bridging the Gap Between HPC and Big Data Frameworks. *Proc. VLDB Endow.* 10, 8 (April 2017), 901–912. https://doi.org/10.14778/3090163.3090168

[4] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining (WSDM '19)*. ACM, New York, NY, USA, 384–392. https://doi.org/10.1145/3289600.3290967

[5] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166. https://doi.org/10.1126/science.aad9029

[6] J. W. Berry. 2011. Practical Heuristics for Inexact Subgraph Isomorphism. In *Technical Report SAND2011-6558W*. Sandia National Laboratories, 8.

[7] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. 2007. Software and Algorithms for Graph Queries on Multithreaded Architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–14. https://doi.org/10.1109/IPDPS.2007.370685

[8] H. Bunke. 1997. On a Relation Between Graph Edit Distance and Maximum Common Subgraph. *Pattern Recogn. Lett.* 18, 9 (Aug. 1997), 689–694. https://doi.org/10.1016/S0167-8655(97)00060-3

[9] H Bunke and G Allermann. 1983. Inexact Graph Matching for Structural Pattern Recognition. *Pattern Recogn. Lett.* 1, 4 (May 1983), 245–253. https://doi.org/10.1016/0167-8655(83)90033-8

[10] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber. 2016. Subgraph Counting: Color Coding Beyond Trees. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2–11. https://doi.org/10.1109/IPDPS.2016.122

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM Int. Conf. on Data Mining*. Society for Industrial Mathematics, p. 442.

[12] D. Conte, P. Foggia, C. Sansone, and M. Vento. 2004. THIRTY YEARS OF GRAPH MATCHING IN PATTERN RECOGNITION. *International Journal of Pattern Recognition and Artificial Intelligence* 18, 03 (2004), 265–298. https://doi.org/10.1142/S0218001404003228 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0218001404003228

[13] Boxin Du, Si Zhang, Nan Cao, and Hanghang Tong. 2017. FIRST: Fast Interactive Attributed Subgraph Matching. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, USA, 1447–1456. https://doi.org/10.1145/3097983.3098040

[14] Sourav Dutta, Pratik Nayek, and Arnab Bhattacharya. 2017. Neighbor-Aware Search for Approximate Labeled Graph Matching Using the Chi-Square Statistics. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1281–1290. https://doi.org/10.1145/3038912.3052561

[15] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph. *Proc. VLDB Endow.* 7, 7 (March 2014), 517–528. https://doi.org/10.14778/2732286.2732289

[16] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph Pattern Matching: From Intractable to Polynomial Time. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 264–275. https://doi.org/10.14778/1920841.1920878

[17] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified Top-k Graph Pattern Matching. *Proc. VLDB Endow.* 6, 13 (Aug. 2013), 1510–1521. https://doi.org/10.14778/2536258.2536263

[18] Stefan Fankhauser, Kaspar Riesen, and Horst Bunke. 2011. Speeding Up Graph Edit Distance Computation Through Fast Bipartite Matching. In *Proceedings of the 8th International Conference on Graph-based Representations in Pattern Recognition (GbRPR'11)*. Springer-Verlag, Berlin, Heidelberg, 102–111. http://dl.acm.org/citation.cfm?id=2009206.2009219

[19] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. 2013. A distributed vertex-centric approach for pattern matching in massive graphs. In *2013 IEEE International Conference on Big Data*. 403–411. https://doi.org/10.1109/BigData.2013.6691601

[20] Giraph. 2016. Giraph. http://giraph.apache.org

[21] Alex Gittens, Kai Rothauge, Shusen Wang, Michael W. Mahoney, Lisa Gerhardt, Prabhat, Jey Kottalam, Michael Ringenburg, and Kristyn Maschhoff. 2018. Accelerating Large-Scale Data Analysis by Offloading to High-Performance Computing Libraries Using Alchemist. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '18)*. ACM, New York, NY, USA, 293–301. https://doi.org/10.1145/3219819.3219927

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 599–613. http://dl.acm.org/citation.cfm?id=2685048.2685096

[24] Graph 500. 2016. Graph 500 benchmark. http://www.graph500.org/

[25] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[26] HavoqGT. 2016. HavoqGT. http://software.llnl.gov/havoqgt

[27] Huahai He and A. K. Singh. 2006. Closure-Tree: An Index Structure for Graph Queries. In *22nd International Conference on Data Engineering (ICDE'06)*. 38–38. https://doi.org/10.1109/ICDE.2006.37

[28] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS '95)*. IEEE Computer Society, Washington, DC, USA, 453–. http://dl.acm.org/citation.cfm?id=795662.796255

[29] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 58, 12 pages. https://doi.org/10.1145/2807591.2807620

[30] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. 2004. Cyclic Pattern Kernels for Predictive Graph Mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*. ACM, New York, NY, USA, 158–167. https:

//doi.org/10.1145/1014052.1014072

[31] Michael Hunger and William Lyon. 2016. Analyzing the Panama Papers with Neo4j: Data Models, Queries and More. https://neo4j.com/blog/analyzing-panama-papers-neo4j

[32] IMDb. 2016. IMDb Public Data. http://www.imdb.com/interfaces

[33] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 745–761. https://www.usenix.org/conference/osdi18/presentation/iyer

[34] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=SJU4ayYgl

[35] G. Liu, K. Zheng, Y. Wang, M. A. Orgun, A. Liu, L. Zhao, and X. Zhou. 2015. Multi-Constrained Graph Pattern Matching in large-scale contextual social graphs. In *2015 IEEE 31st International Conference on Data Engineering*. 351–362. https://doi.org/10.1109/ICDE.2015.7113297

[36] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09)*. ACM, New York, NY, USA, 557–566. https://doi.org/10.1145/1557019.1557083

[37] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. 2012. Distributed Graph Pattern Matching. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 949–958. https://doi.org/10.1145/2187836.2187963

[38] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[39] N. Malitsky, A. Chaudhary, S. Jourdain, M. Cowan, P. OâĂŹLeary, M. Hanwell, and K. K. Van Dam. 2017. Building near-real-time processing pipelines with the spark-MPI platform. In *2017 New York Scientific Data Summit (NYSDS)*. 1–8. https://doi.org/10.1109/NYSDS.2017.8085039

[40] Brendan D. Mckay and Adolfo Piperno. 2014. Practical Graph Isomorphism, II. *J. Symb. Comput.* 60 (Jan. 2014), 94–112. https://doi.org/10.1016/j.jsc.2013.09.003

[41] Bruno T. Messmer and Horst Bunke. 1998. A New Algorithm for Error-Tolerant Subgraph Isomorphism Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 20, 5 (May 1998), 493–504. https://doi.org/10.1109/34.682179

[42] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (Oct. 2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[43] Benedict Paten, Adam M. Novak, Jordan M. Eizenga, and Garrison Erik. 2017. Genome Graphs and the Evolution of Genome Inference. *bioRxiv* (2017). https://doi.org/10.1101/101816 arXiv:https://www.biorxiv.org/content/early/2017/03/14/101816.full.pdf

[44] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and Sampling Triangles from a Graph Stream. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1870–1881. https://doi.org/10.14778/2556549.2556569

[45] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2013. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 825–836. https://doi.org/10.1109/IPDPS.2013.72

[46] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2014. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 549–559. https://doi.org/10.1109/SC.2014.50

[47] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. ACM, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[48] Todd Plantenga. 2013. Inexact Subgraph Isomorphism in MapReduce. *J. Parallel Distrib. Comput.* 73, 2 (Feb. 2013), 164–175. https://doi.org/10.1016/j.jpdc.2012.10.005

[49] Quartz. 2017. Quartz. https://hpc.llnl.gov/hardware/platforms/Quartz

[50] Reddit. 2016. Reddit Public Data. https://github.com/dewarim/reddit-data-tools

[51] Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. 2018. PruneJuice: Pruning Trillion-edge Graphs to a Precise Pattern-matching Solution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 21, 17 pages. http://dl.acm.org/citation.cfm?id=3291656.3291684

[52] Pedro Ribeiro and Fernando Silva. 2014. G-Tries: A Data Structure for Storing and Finding Subgraphs. *Data Min. Knowl. Discov.* 28, 2 (March 2014), 337–377. https://doi.org/10.1007/s10618-013-0303-4

[53] Kaspar Riesen and Horst Bunke. 2009. Approximate Graph Edit Distance Computation by Means of Bipartite Graph Matching. *Image Vision Comput.* 27, 7 (June 2009), 950–959. https://doi.org/10.1016/j.imavis.2008.04.004

[54] Oliver Lehmberg Robert Meusel, Christian Bizer. 2016. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/index.html

[55] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. 2017. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences and Systems (GRADES'17)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3078447.3078454

[56] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. QFrag: Distributed Graph Search via Subgraph Isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 214–228. https://doi.org/10.1145/3127479.3131625

[57] D. Siegal, J. Guo, and G. Agrawal. 2016. Smart-MLlib: A High-Performance Machine-Learning Library. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 336–345. https://doi.org/10.1109/CLUSTER.2016.49

[58] G. M. Slota and K. Madduri. 2014. Complex network analysis using parallel approximate motif counting. In *Proc. 28th IEEE Int'l. Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 405–414. https://doi.org/10.1109/IPDPS.2014.50

[59] Spark. 2017. Spark. https://spark.apache.org

[60] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (May 2012), 788–799. https://doi.org/10.14778/2311906.2311907

[61] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi,

Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[62] Siddharth Suri and Sergei Vassilvitskii. 2011. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*. ACM, New York, NY, USA, 607–614. https://doi.org/10.1145/1963405.1963491

[63] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 425–440. https://doi.org/10.1145/2815400.2815410

[64] Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. 2007. Fast Best-effort Pattern Matching in Large Attributed Graphs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*. ACM, New York, NY, USA, 737–746. https://doi.org/10.1145/1281192.1281271

[65] N. Tripoul, H. Halawa, T. Reza, G. Sanders, R. Pearce, and M. Ripeanu. 2018. There are Trillions of Little Forks in the Road. Choose Wisely! - Estimating the Cost and Likelihood of Success of Constrained Walks to Optimize a Graph Pruning Pipeline. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 20–27. https://doi.org/10.1109/IA3.2018.00010

[66] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. https://doi.org/10.1145/321921.321925

[67] Edward Walker. 2009. The Real Cost of a CPU Hour. *Computer* 42, 4 (April 2009), 35–41. https://doi.org/10.1109/MC.2009.135

[68] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 763–782. https://www.usenix.org/conference/osdi18/presentation/wang

[69] M. P. Wellman and W. E. Walsh. 2000. Distributed quiescence detection in multiagent negotiation. In *Proceedings Fourth International Conference on MultiAgent Systems*. 317–324. https://doi.org/10.1109/ICMAS.2000.858469

[70] Abdurrahman Yasar and Ümit V. Çatalyürek. 2018. An Iterative Global Structure-Assisted Labeled Network Aligner. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '18)*. ACM, New York, NY, USA, 2614–2623. https://doi.org/10.1145/3219819.3220079

[71] Ye Yuan, Guoren Wang, Jeffery Yu Xu, and Lei Chen. 2015. Efficient Distributed Subgraph Similarity Matching. *The VLDB Journal* 24, 3 (June 2015), 369–394. https://doi.org/10.1007/s00778-015-0381-6

[72] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2007. Out-of-core Coherent Closed Quasi-clique Mining from Large Dense Graph Databases. *ACM Trans. Database Syst.* 32, 2, Article 13 (June 2007). https://doi.org/10.1145/1242524.1242530

[73] Shijie Zhang, Jiong Yang, and Wei Jin. 2010. SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1185–1194. https://doi.org/10.14778/1920841.1920988

[74] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. 2012. SAHAD: Subgraph Analysis in Massive Networks Using Hadoop. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 390–401. https://doi.org/10.1109/IPDPS.2012.44

[75] Feida Zhu, Qiang Qu, David Lo, Xifeng Yan, Jiawei Han, and Philip S. Yu. 2011. Mining top-K large structural patterns in a massive network. *Proceedings of the VLDB Endowment* 4, 11 (8 2011), 807–818.