Dryadic: Flexible and Fast Graph Pattern Matching at Scale

Abstract—**Graph pattern matching searches a data graph for all instances of one or more query patterns. Since it is one of the most fundamental problems in graph analytics, many graph pattern matching systems have been proposed with distinct features to provide a mix of flexibility and performance. It is generally accepted that distinct use cases may necessitate the use of different systems. In this paper, we propose Dryadic, a system which integrates comprehensive flexibility features, yet can still outperform five state-of-the-art graph pattern matching systems on the use cases they optimize for. Unlike existing systems that employ a case-by-case design strategy, all functionalities of Dryadic are centered around a powerful intermediate representation, the computation tree structure, which encodes the matching algorithms for arbitrary patterns. Dryadic implements novel techniques to optimize the computation tree and maps it to different backends to perform compiled, interpreted, or distributed graph pattern matching. Extensive experiments on nine real-world graphs of different scales show that Dryadic, despite its all-in-one nature, is often one to three orders of magnitude faster than other systems in three common usage scenarios.**

I. Introduction

The amount of graph data has recently surged in numerous domains, including bioinformatics [33], social networks [30], and cybersecurity [40]. Each domain has its own demands for graph data processing, but each also shares a significant interest in *graph pattern matching*. This family of problems stems from the well-known Subgraph Isomorphism problem, and requires finding all subgraphs in a dataset that are isomorphic to a given query pattern. As pointed out by Sahu et al. [43], many problems, such as motif enumeration, clique finding, and subgraph matching, are variants of the *graph pattern matching* problem. Many applications, including fraud detection [36] and graph mining [17], use subgraph pattern matching as the primitive within their core functionality, and consequently demand a graph matching system that can provide both sufficient *flexibility* and high *performance*.

A *flexible* subgraph pattern matching system should support the following key features. First, the system should enable both edge-induced and vertex-induced pattern matching. Edge-induced matching focuses on identifying the connectivity between vertices, which is widely used in graph database systems [37], [50]. Vertex-induced matching additionally considers the absence of connectivity and is useful for graph characterization [2], [49]. Second, the system should support labeled pattern matching to allow applications to leverage the rich non-topological data that labels can offer. Third, the system should handle arbitrary pattern queries which may or may not be available for offline optimization. Finally, the system should be able to process large-scale graphs with billions of edges, which is increasingly important due to the tremendous growth rate of real-world graphs [43].

The *performance* requirement faces even more challenges. Subgraph Isomorphism is a known NP-Complete problem [14], so the inherent workload for graph pattern matching problems is far heavier than that of traditional graph analytics (e.g. graph traversal). High performance systems should optimize not only for the irregularity inherent in the graph data structure, but also the complex control flow and redundant computation induced by the matching algorithm. Moreover, pattern queries may only be available online, inhibiting exhaustive search-based optimization used in most compilation systems.

There exist many *performance*-oriented systems for graph pattern matching, but each specializes in only a subset of the *flexibility* features. For example, DAF [18] can efficiently process small labeled graphs, but its performance drops quickly when processing reasonably large graphs with millions of edges. AutoMine [32] employs compilation techniques to generate query code tailored to a given set of patterns but must re-compile for every new query. Pangolin [8] leverages the high-performance parallel operation and scheduling of the Galois engine [38] and avoids re-compilation for new queries. However, Pangolin can be more than ten times slower than AutoMine for heavy workloads. While it is always possible to “tune” the experiments to favor one system over another, as shown in Section II, these systems form a superiority cycle for common workloads. Therefore, no system can consistently outperform the others. A user would have to use multiple different systems to handle distinct usage scenarios.

This paper proposes Dryadic, a flexible and efficient graph pattern matching system which 1) supports the above-mentioned important *flexibility* features to enable easy adoption into existing real-world applications and 2) offers substantially better *performance* than all existing systems in the usage scenario they are specialized for. Dryadic is motivated by an observation that existing systems specialize the execution of similar pattern matching algorithms in specific settings. The observation demonstrates an opportunity to employ a key methodology leveraged in compiler research – intermediate representation-centered optimization and backend support.

However, building the intermediate representation and efficient backends for graph pattern matching presents multiple unique challenges. First, the intermediate representation must be flexible enough to meet the needs of different pattern matching algorithms (e.g., simultaneous multi-pattern matching). Second, the intermediate representation should be easy to manipulate to apply both static and dynamic optimizations. Third, different usage scenarios (e.g., patterns available offline vs. online) demand dramatically different backends, each of which requires significant effort to develop.

To address these challenges, Dryadic builds a flexible intermediate representation, called the computation tree, which encodes a compact representation of matching algorithms for arbitrary labeled or unlabeled patterns. The tree structure facilitates important optimizations to eliminate computational redundancy both within a single pattern's algorithm, and between the algorithms for multiple patterns. It also supports a work-stealing runtime, which enables parallel workers to steal fine-grained tasks from other workers to dramatically improve load balance. In comparison, existing work only supports coarse-grained stealing. Moreover, the tree structure is simple enough to interpret, compile, and parallelize.

Empowered by its intermediate representation, Dryadic's backends are much easier to build compared to specialized systems. Each backend only needs to determine the order to execute the computation tree on the data graph and implement the rudimentary operations. We implemented three backends with about 3,000 lines of C++ code. Dryadic can compile the computation tree directly to efficient, reusable C++ code for parallel and distributed execution when the target pattern is fixed and used across multiple input graphs. If the pattern is only known at runtime, Dryadic quickly builds a computation tree and uses the Galois parallel engine [38] as an interpreter to run it on the input graph.

To evaluate Dryadic's performance in different usage scenarios, we always compare it with the state-of-the-art graph pattern matching system specialized for that use case. The highlights of the results are summarized as follows: 1) For labeled pattern matching, Dryadic is up to 56X and on average 11.4X faster than DAF on 64 workloads using 10 patterns and 7 graphs. 2) We study two use cases in unlabeled multi-pattern matching. For single-machine parallel motif enumeration, Dryadic outperforms Pangolin and AutoMine by up to 25.4X and 6X, respectively. For single-machine parallel motif-counting, Dryadic is on average 5X faster than PGD [2], which is the fasted manual implementation of size-4 motif counting (to our best knowledge). 4) For distributed graph pattern matching on 16 machines, Dryadic is up to 20X faster than CECI.

Overall, the paper makes the following contributions. 1) We propose the Dryadic system to harmonize *flexibility* and *performance*, which supports the most comprehensive features and still outperforms specialized state-of-the-art subgraph pattern matching systems. 2) We propose the computation tree representation to encode matching algorithms for arbitrary labeled and unlabeled patterns, as well as multiple optimizations to eliminate computation redundancy and improve load balance. 3) We develop a set of backends to efficiently map these optimized trees to unique execution environments.

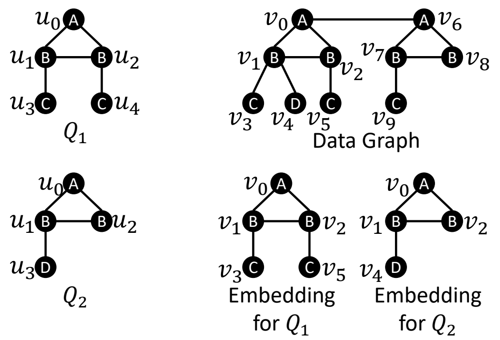


Fig. 1: A graph pattern matching example with two labeled patterns.

II. Background and Motivation

Many graph pattern matching systems have slightly different definitions of the problem and claim a variety of distinct features. In this section, we define the graph pattern matching problem in a flexible manner. Next, we describe five distinct features which distinguish multiple state-of-the-art pattern matching systems from others. Finally, we present the demand and opportunity to implement all these features in one single system while providing the best performance over all existing systems.

A. Graph Pattern Matching Basics

Given a query graph pattern $Q = (V_Q, E_Q)$ where V_Q is a set of vertices and E_Q is a set edges whose end vertices are in V_Q. The graph pattern matching problem accepts an input graph G and identifies all subgraphs of G that are isomorphic to Q. Each such subgraph $S = (V_S, E_S)$ is called an induced subgraph and the mapping from Q to S is called an embedding. S is a vertex-induced subgraph if S includes all edges in G whose endpoints are in V_S. Otherwise, S is edge-induced. Like many prior studies [32], [3], [26], we focus on undirected graphs, while the techniques can be readily applied to directed patterns and graphs.

Figure 1 shows an example of matching two query patterns in a data graph. A graph pattern matching system should identify two sets of subgraphs. The subgraphs in the first set should be isomorphic to $Q1$ and those in the second set isomorphic to $Q2$. To compute a set, the system can generate a matching order. For instance, the matching order could be (A, B, B, C) to match $Q1$. The system then follows this order to identify embeddings of $Q1$ in the data graph. In this paper, we use the matching order generation algorithm proposed in AutoMine [32]. We leave dynamic matching order generation and selection to future work.

B. Distinct Features in Existing Systems

State-of-the-art systems implement some distinct *performance* features, which empower them to substantially exceed the capabilities of prior work. We illustrate five critical features and briefly describe how they are implemented in those systems.

Feature 1: Symmetry breaking. The inherent symmetry in the pattern may cause redundant computation by identifying the same embedding more than once. For instance, without symmetry breaking, each embedding of $Q1$ in Figure 1 is identified twice because based on the pattern topology and label information, it is impossible to distinguish the two B vertices connected to A. A popular method [16] is to use the vertex IDs to break symmetry. For $Q1$, we can enforce that in the same embedding the first vertex matching B should have a larger ID than the second vertex matching B. As in multiple prior systems [3], [31], [41], we also follow this idea to break symmetry.

Feature 2: Multi-pattern redundancy elimination. When a user queries multiple patterns, matching the patterns sequentially may cause significant computation and data access redundancy. Consider the two patterns in Figure 1. Since they share the same triangle-shaped sub-pattern, the two queries should be executed at the same time to share the sub-embeddings corresponding to the shared sub-pattern. Arabesque [49], RStream [53], and Pangolin [8] naturally exploit the shared sub-patterns through their iterative exploration-reduction execution model. Each iteration consists of an exploration and a reduction phase. The exploration phase extends a set of initial embeddings by appending one more connected vertex or edge to each. The reduction phase runs an isomorphism check on each extended embedding to determine whether it matches a subpattern of the query. The system keeps the matched ones as initial embeddings for the next iteration. When embeddings of different patterns are formed from the same sub-embedding, these systems successfully eliminate redundancy.

Feature 3: Leveraging efficient graph processing runtime. Several existing systems are built upon highly optimized parallel runtime to improve performance [49], [8]. For example, Pangolin is built upon Galois, a parallel processing engine particularly good at irregular applications, to accelerate operation scheduling and synchronization. Pangolin thus evolves together with Galois and enjoys its additional graph processing features. In comparison, standalone graph pattern matching systems may not get adopted due to the limited functionality.

Feature 4: Code specialization. Compilation-based graph pattern matching systems, represented by EmptyHeaded [1] and AutoMine, generate a specialized program for the given pattern. The program has a nested loop at each layer trying to extend the embedding by including one more vertex. This approach avoids substantial runtime overhead incurred by non-compilation based systems, but it generates a new program whenever a new pattern is given, which requires another round of compilation, but may be amortized over many uses.

Feature 5: Parallel and Distributed matching. Pattern matching on large-scale graphs is compute-intensive and has a huge amount of inherent parallelism, which motivates many systems to leverage parallel and distributed execution to improve performance. AutoMine and Pangolin respectively use OpenMP and Galois for parallel processing on a single machine. Distributed pattern matching systems, represented by CECI [3], focus their optimizations on load balance and minimizing the data exchange between machines.

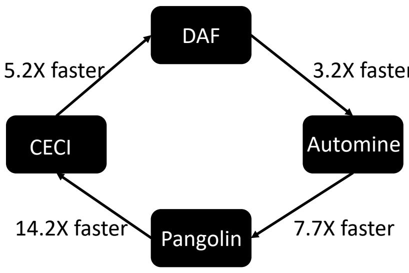


Fig. 2: The circular "superior-to" relationship between five graph pattern matching systems.

C. Demand and Opportunity for One Single Flexible and Efficient Graph Matching Systems

Every existing system is short in one or more of the flexibility and performance features in order to specialize for a particular use case. They have a good reason to make such a choice because different features often indicate conflicting optimization goals. However, modern applications are complex and may encounter different usage scenarios, which makes it difficult to determine the best system to satisfy different requirements.

To demonstrate this challenge, we conduct four experiments, each to compare two systems, with several popular graph datasets used in many prior studies [32], [3], [49], [8]. Figure 2 shows that the five considered systems form a circular "superior-to" relationship. DAF is faster than AutoMine when running five different queries on the Mico graph (details in Section IV) because the pattern queries are given online, so AutoMine's code generation and compilation costs are on the critical path. When we match a fixed pattern (a size-4 clique) on the LiveJournal graph, AutoMine produces better performance than Pangolin, which in turn outperforms CECI. CECI could be 5.2X faster than DAF on the Patents graph thanks to CECI's distributed computing capability.

We observe that these graph pattern matching systems run similar algorithms, which identify and prune certain sets of candidate vertices in the data graph to match the pattern vertices. Each system employs manual optimizations for the targeted setting (e.g., large graph size or availability of query patterns). However, in many other domains, systems are not built in such an ad hoc manner. For example, database systems build one or more intermediate representations for arbitrary queries, and leverage a set of query optimizations and backends to map the representations to specialized executions. The Java virtual machine uses a similar methodology to build even more general intermediate representations, optimizations, and backends for arbitrary Java programs. This paper is the first to apply the intermediate representation-centered design philosophy, which has been tremendously successful in other domains, to graph pattern matching.

III. The Dryadic System

The Dryadic system integrates all the flexibility features identified in the previous sections, and still outperforms each state-of-the-art graph pattern matching system in its area of specialization. The power of Dryadic roots in a key methodology leveraged by compiler research – intermediate representation-centered optimization and backend support. The intermediate representation encodes the matching algorithms for arbitrary patterns and is amenable to both static and dynamic optimizations, as well as compilation and interpretation. Each backend is tailored for a particular user scenario or hardware setting, and maps the intermediate representation to efficient execution.

Dryadic implements a tree-structured intermediate representation, referred to as the computation tree, to encode the graph pattern matching algorithms. Dryadic has three major components centered around the computation tree. The tree construction component takes the input patterns, and generates the matching order and symmetry-breaking restrictions for each pattern. It then builds the computation tree by merging the matching orders and associates each node in the tree with a compound set operation. The tree optimization component applies several optimizations to the computation tree to eliminate redundant computation and improve load balance. The execution component can be configured to meet specific needs, which supports three execution modes, including interpreted execution by Galois and C++ code generation for parallel and distributed execution.

The Dryadic system is straightforward to use. A user only needs to provide a file to describe patterns of interest in a simple format and specify one of the supported execution modes (i.e., compiled or interpreted). Dryadic also implements helper functions to generate patterns. For instance, it can generate all connected patterns of a certain size to support applications such as motif counting.

A. Computation Tree-Centered Representation and Optimization

Dryadic builds a computation tree to naturally support simultaneous multi-pattern matching as described in Section III-A1. The static optimizations move partial set operations to upper levels of the tree to avoid redundant computation, and are described in Section III-A2. The runtime optimization achieves load balance through fine-grained work stealing as described in Section III-A3.

1) Computation Tree Construction: Given a pattern, Dryadic uses prior techniques to compute a matching order and the restrictions to break symmetry as described in Section II. The matching order of a pattern specifies the dependencies of computations to identify the vertices in an embedding, while the restrictions enforce "id-is-larger" relations between some of the vertices to avoid identifying the same embedding multiple times. The matching order, the restrictions, and the topology of the pattern together determine the operations to compute the matched vertices for each pattern vertex.

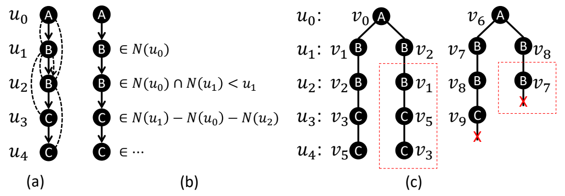


Fig. 3: A computation path for $Q1$ in Figure 1 and its corresponding embedding trees for a data graph. In (a), the solid lines with arrows indicate the matching order, the dotted line with an arrow means the source pattern vertex restricts the destination pattern vertex, and the dotted lines without arrows show the topological relations between the vertices in the query pattern. In (c), the dotted box means the included subtree is pruned by the restriction. A cross shows that no matched vertices can be found to extend the tree along the corresponding path.

Figure 3 (a) shows a graph structure to demonstrate all these three kinds of information. However, it does not directly show what computations should be performed on the data graph. We hence transform it to the representation in Figure 3 (b) by assuming that a user is interested in vertex-induced graph pattern matching. In this new representation, except the first pattern vertex (i.e., A), every pattern vertex is associated with a compound set operation to encode its topological relations with other pattern vertices before it in the matching order. For example, the fourth pattern vertex's compound set operation is $N(u_1) - N(u_0) - N(u_2)$, where N is an operation to return the neighbor list of a vertex and the minus sign represents a set difference operation. The intuitive understanding is that u_3 is in u_1's neighbor list but not in u_0 or u_2's neighbor list. The compound set operations specify ways to extend partial embeddings. Consider an embedding consisting of v_i, v_j, and v_k that matches the sub-pattern formed by u_0, u_1, and u_2 (i.e, a triangle). If v_i, v_j, and v_k match u_0, u_1, and u_2, respectively, each vertex in the set computed by $N(v_j) - N(v_i) - N(v_k)$ forms an embedding with v_i, v_j, and v_k, which matches the sub-pattern consisting of u_0, u_1, u_2, and u_3 (i.e., a tailed triangle). To account for the restrictions, we need bounded set operations like the one associated with the third pattern vertex. In this example, u_1 restricts u_2, so if v_j, which matches u_1, is used to compute a set of vertices to match u_2 (referred to as u_2's vertex set), the ID of each vertex in the set should be smaller than v_j's ID.

We define a matching order and its associated compound set operations as a *computation path*, represented by CP. $CP[i]$ is called a computation node. $CP[i].label$ and $CP[i].SetOp$ respectively represent the label and the associated compound set operation of the ith pattern vertex in the matching order. Note that the computation path only defines how to extend each partial embedding towards the target pattern instead of the order to extend the partial embeddings. Two intuitive orders to extend partial embeddings are breadth-first and depth-first. The breadth-first order requires that smaller partial embeddings should be extended before larger partial embeddings. When

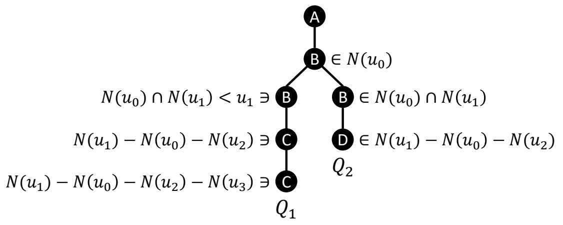

A
$\in N(u_0)$
$N(u_0) \cap N(u_1) < u_1 \ni$
$\in N(u_0) \cap N(u_1)$
$N(u_1) - N(u_0) - N(u_2) \ni$
$\in N(u_1) - N(u_0) - N(u_2)$
Q_2
$N(u_1) - N(u_0) - N(u_2) - N(u_3) \ni$
Q_1

Fig. 4: A computation tree and its corresponding embedding tree for a data graph.

there exist more than one smallest partial embeddings, the embedding generated the earliest should be extended, which can be easily implemented by a queue. The depth-first order, in contrast, extends larger embeddings first, and can be implemented with a stack of increasingly larger embeddings.

Figure 3 (c) shows the matched vertices when running the computation path in Figure 3 (b) on each vertex in the data graph from Figure 1. The execution on each data vertex forms a tree structure, which we call an embedding tree. Note that the figure only demonstrates two embedding trees because only two data vertices successfully match the root pattern vertex. The embedding tree has three important properties. First, its height is at most $|Q_V|$, the number of vertices in the query pattern. Second, each path of length $|Q_V|$ from root to a leaf represents a distinct embedding. Third, the embeddings explored by a pruned sub-tree are present in a non-pruned path, due to symmetry breaking.

Given multiple patterns, Dryadic builds a distinct computation path for each pattern and merges the computation paths to form a computation tree as follows. $CP_i[k]$ and $CP_j[k]$ are merged if and only if the first $k-1$ computation nodes in CP_i and CP_j are merged and $CP_i[k].label = CP_j[k].label$ & $CP_i[k].SetOp = CP_j[k].SetOp$. Figure 4 shows a computation tree by merging the computation paths for $Q1$ and $Q2$ from Figure 1.

As shown in Figure 4, the merged computation tree for $Q1$ and $Q2$ only has seven pattern vertices because the first two vertices in both matching orders have the same associated set operations and their labels are the same. The compound set operations associated with the two vertices and their results are reused by both patterns. However, we cannot merge the third vertices in the two patterns. Although they have the same topological relations to the first two vertices and the labels are the same, their associated set operations are different (i.e., one is bounded and the other is not) due to symmetry breaking. Note the merging does not confuse embeddings for the two patterns. The vertices matched to the last pattern vertex for each pattern uniquely identify the corresponding embeddings.

Given a pattern, Dryadic builds the same computation tree to perform both vertex-induced and edge-induced graph pattern matching. The compound set operations for edge-induced pattern matching replaces set difference with an operation that removes the corresponding vertex from the set, to guarantee no vertex appears more than once in an embedding.

2) Operation Motion to Minimize Redundant Computation: Computation tree merging eliminates a certain amount of redundancy, but there still exist two kinds of redundancy. First, set operations may run multiple times with the same inputs. Consider the operation $N(u_1) - N(u_0) - N(u_2) - N(u_3)$ associated with the pattern vertex u_4 in Figure 4. If the vertex matched to u_2 has a large degree, it is possible that the first two set operations (i.e., $N(u_1) - N(u_0) - N(u_2)$) are repetitively run with the three vertices mapped to u_0, u_1, and u_2 as inputs. Because the embedding tree expands to include more vertices at lower levels, this redundancy problem is even more serious than the one addressed by merging the computation trees. Second, the same set operation may be performed multiple times with the same input to match different patterns. As Figure 4 shows, the compound set operations associated with the fourth vertex in $Q1$ and $Q2$ both perform $N(u_1) - N(u_0)$, which may be redundant if the extended embeddings for both patterns have the same vertices mapped to u_0 and u_1.

Algorithm 1: Operation motion algorithm

```
   input : node //A node in the computation tree
1  begin
2      for child in node.children do
3          run Code Motion on child
4      if node.parent ≠ null then
           // node.setOp is the SetOp
              associated with node
           // node.sets is a set of setOps
              moved to node from node's
              sub-tree
5          Insert node.setOp to node.parent.sets
6          for setOp in node.sets do
7              if setOp.parent ∈ node.parent.sets then
8                  continue
9              else
10                 Insert setOp to node.parent.sets
```

We call a set operation a *redundant operation* if it is run multiple times with the same inputs to explore the same embedding tree. Dryadic implements Algorithm 1 to completely eliminate redundant operations by moving set operations to upper levels of the computation tree, which we refer to as *operation motion*. It uses a three-field data structure, $SetOp_i$, to represent a compound set operation associated with a pattern vertex i. The first field, in, includes all pattern vertices connected to vertex i and before vertex i in the matching order. The second field, out, contains all pattern vertices disconnected to vertex i and before i in the matching order. Finally, the res field contains the ID of the pattern vertex restricting vertex i. Recall that within an embedding the ID of the matched vertex for the ith pattern vertex must be larger than the ID of the matched vertex for the pattern vertex indicated by res. If res is -1, the vertex set computed by $SetOp_i$ is not restricted. We define $SetOp.parent$ as the parent of $SetOp$, which is a copy of $SetOp$ except that 1) it excludes the largest ID in $SetOp.in$ and $SetOp.out$ and 2) it has -1 in its res field. Given that $SetOp.parent$ is already computed, the compound set operation corresponding to $SetOp$ only needs to perform one set operation. For example, $N(u_1) - N(u_0) - N(u_2)$'s parent is $N(u_1) - N(u_0)$. Given that u_1 and u_0 are mapped to the same data vertices, the former only needs to perform a set difference operation if the result of the latter is available.

Algorithm 1 recursively moves $SetOp_i.parent$ to associate with the parent vertex of the vertex $SetOp_i$ is associated with. It uses a set container to store all the $SetOp$s associated with a pattern vertex to eliminate duplicates. When the algorithm terminates, every pattern vertex in the computation tree has a set of distinct $SetOps$'. We show in the following lemma (proof omitted for brevity) that this algorithm guarantees that the processed computation tree is free of redundancy.

Lemma 1. *Algorithm 1 does not change the semantics of the computation tree and eliminates all redundant set operations.*

Dryadic implements two additional optimizations to further improve the performance of the computation tree. The first optimization is based on the observation that if $SetOp.ins$ is empty, we have to use expensive set complement operations. For instance, if $SetOp_i.ins = \emptyset$ and $SetOp_i.out = \{j\}$, we need to compute all vertices not in the neighbor list of the vertex matched to pattern vertex j. Hence, Dryadic does not move $SetOp.parent$ to a higher level, if $|SetOp.in| = 1$ and $|SetOp.parent.in| = 0$.

The second optimization is motivated by the observation that because Algorithm 1 always sets the res field to -1 when computing $SetOp.parent$, it may compute an unnecessarily large vertex set. Consider an unlabeled clique pattern. The ith ($i \geq 2$) pattern vertex's parent is at least associated with $SetOp_{i-1}$ and $SetOp_i.parent$. The former is restricted by $i-2$ while the latter is not restricted. However, since $SetOp_i$ is restricted by $i-1$, the matched vertices computed by $SetOp_i.parent$ would be useless if they violate the restriction. Therefore, we should leverage the transitivity of restrictions and also use $i-1$ to restrict $SetOp_i.parent$. We generalize the idea in the following way. Given that $SetOp_j$ is generated by the recursive procedure starting from $SetOp_k$, we assign j to $SetOp_j.res$ is the jth pattern vertex transitively restricts the kth pattern vertex.

3) Load Balance with Minimized Footprint: The computation tree-based representation is amenable to parallelization because a system can use different workers to easily execute it on different vertices in the data graph. However, such a naive parallelization method would lead to a serious load balance problem. As shown in numerous prior studies [25], [15], [7], [42], the degree distribution in real-world graphs is highly skewed, so the embedding tree rooted at one vertex can be several orders of magnitude larger than one rooted on a different vertex. Worse, the more complex the pattern is, the more serious the problem becomes. Assume that we are interested in all the single-edge embeddings in an unlabeled data graph. The gap between the largest embedding tree and the smallest is $O(D_{max}) - O(D_{min})$ where D_{max} and D_{min} respectively represent the largest and the smallest degree in the data graph. If we instead match the triangle pattern, this gap becomes $O(D_{max}^2) - O(D_{min}^2)$.

A popular approach to combating the load imbalance problem is to compute all smaller embeddings before larger embeddings [49], [53], [8]. Explained in the computation tree terms, they perform a breadth-first execution of the computation tree on the data graph. Specifically, they enforce a global synchronization between any two adjacent levels across all embedding trees. Because of the inherent dependencies in the embedding tree structure, they have to maintain at least all embeddings at level i before the computation of vertices at level $i+1$. This approach hence has significant space overhead which cancels or even outweighs its benefit, especially for large graphs.

Although Dryadic can easily support breadth-first execution, its default execution is depth-first to minimize the memory footprint. A straightforward approach to improving load balance is to support coarse-grained work stealing [3]. Every worker is initially assigned a set of data vertices to work on. An idle worker steals from other workers their unprocessed data vertices. However, as mentioned above, this approach does not resolve the load balance issue introduced by the huge disparity between embedding trees rooted at different vertices.

Dryadic improves load balance through fine-grained work stealing based on the computation tree and embedding tree abstraction. While the essence of coarse-grained work stealing is to steal entire embedding trees, Dryadic's fine-grained work stealing supports stealing arbitrary sub-embedding trees. Consider the example in Figure 3. Suppose that worker 2 finishes traversing the embedding tree rooted at data vertex v_6 when worker 1 is still processing the embedding tree rooted at vertex v_0 and is somewhere in the left sub-embedding tree. Worker 2 can in concept safely steal the right sub-embedding tree. However, we should address three problems to properly implement the stealing. First, the runtime should determine which sub-embedding tree, if any left, to steal from the victim worker. Second, we should figure out what data are needed to process the stolen work. Third, we should make sure that processing the stolen work in parallel does not cause data race conditions.

To solve the first problem, Dryadic gives a total order to all paths of the same length from the root data vertex in an embedding tree. Given two paths, P_i and P_j, in the same embedding tree, $P_i > P_j$ if $P_i[k].ID > P_j[k].ID$ where k is the smallest non-negative integer for $P_i[k].ID$ to be different from $P_j[k].ID$. If such a k does not exist, $P_i = P_j$. Because Dryadic implements the vertex set in a way such that the vertices with smaller IDs are always processed first, if a worker is processing the lth vertex in path P, a path P' has not been processed if $P'[0:l-1] > P[0:l-1]$. Based on the ordering, Dryadic randomly selects an active thread as the victim, and runs Algorithm 2 to locate a sub-embedding tree at the highest possible level for work stealing.

```
Algorithm 2: Locating a sub-embedding tree to steal.
  input : $P$ //The path the victim worker is working on
1 begin
2     for $i \leftarrow 0$ to $P.size - 1$ do
3         $vs \leftarrow$ vertex set containing $P[i]$
4         if $P[i]$ is the last vertex in $vs$ then
5             continue
6         else
7             $v \leftarrow vs.pop\_back()$
              // the stolen sub-embedding
                 tree is at level $i$ rooted
                 at $v$
8             return $v$, $i$
9     return $stealing\_fail$
```

```
Algorithm 3: Depth-first interpretation.
  input : $v$ //A data vertex
  input : $n$ //A node in the computation tree
1 begin
2     if $v$ does not match n.label then
3         return
      // $setMap$ stores computed vertex
         sets
4     for $setOp$ in $n.sets$ do
5         $setMap[setOp] \leftarrow$ compute $setOp$ on $v$ with
           $setMap[setOp.parent]$
6     for $n'$ in $n.children$ do
7         for $v'$ in $setMap[n'.setOp]$ do
8             recurse on $v', n'$
```

To compute the stolen sub-embedding tree, we need the path from the root vertex of the embedding tree to its root vertex, as well as all the computed vertex sets based on only the vertices on that path. We wrap all such data in a data structure called the context of the sub-embedding tree. When a sub-embedding tree is stolen, Dryadic duplicates its context and sends it to the stealer. Dryadic also properly synchronizes the stealer and the victim to avoid data race conditions in the stealing process. Finally, the duplication of the context also guarantees that the processing of the stealer does not conflict in any way with the victim.

B. Different Modes to Execute the Computation Tree

The computation tree representation is flexible and can be executed on data graphs in different modes. In this section, we describe an implementation to use the Galois parallel engine to interpret the computation tree in the depth-first order on data graphs. We also describe the code generation component to directly map the computation tree to nested loops in C++ code for parallel and distributed execution.

1) Galois-Based Interpretation: Dryadic leverages the Galois parallel engine to support flexible pattern matching when the patterns are not known offline. Galois implements a set of features to accelerate irregular applications, which empowers multiple graph processing systems. The most important features leveraged by Dryadic are automatic parallelization and per-thread data allocation. Specifically, Dryadic uses the interface provided by Galois to create a work list and pushes all vertices in the data graph into it. Dryadic then passes a lambda function to Galois's *do_all* operator, which automatically runs the function on each vertex in the work list.

The lambda function executes the computation tree in the depth-first order, which can be implemented in a recursive way as described in Algorithm 3. However, in practice the recursion would incur too much overhead. Thus, Dryadic uses a stack to emulate recursion to improve performance. Dryadic allocates a stack for each thread through Galois's per thread storage allocation API. Each element in the stack is a data structure that contains a vertex on the explored path and pointers to the results of its associated set operations. Dryadic pushes the vertex into the stack and executes a loop. The loop body has two parts. The first part computes the compound $SetOps$ associated with the vertex at the top of the stack. Due to the nature of recursion, $SetOp.parent$ must have been computed and its result stored in the $setMap$ as shown in the algorithm. Dryadic needs to chooses one of two options in the second step. If the vertex at the top of the stack has unprocessed neighbors that match the label of the next pattern vertex, Dryadic pushes one of these neighbor vertices into the stack. Otherwise, it pops an element off the top of the stack. The algorithm terminates if the stack does not have any vertex.

Dryadic supports coarse-grained work stealing because the *do_all* operator ensures that no threads stay idle if the work list still has unprocessed vertices. However, recall that the coarse-grained work stealing is insufficient to address the load imbalance problem. Dryadic allows the Galois runtime to interact with the work stealing runtime, discussed in Section III-A3, by adding a dummy vertex into the work list after all data vertices are pushed. When Galois runs the lambda function on the dummy vertex through the *do_all* operator, the lambda function invokes the stealing runtime to attempt to steal work from other threads. A failed attempt means that no other threads have available sub-embedding tree to steal. Otherwise, the lambda function should process the stolen work and push the dummy vertex back to the work list.

Dryadic implements two optimizations to further improve performance. First, The $setMap$ data structure indexed by $SetOp$ incurs substantial overhead because Dryadic needs to frequently access it for most compound set operations. To address this, Dryadic assigns a unique non-negative integer to each set operation and stores the results in a vector indexed by that integer. Second, each set operation produces a new vertex set which needs to be stored in memory, leading to substantial memory management overhead. Dryadic analyzes the computation tree to figure out the maximum number of set operations associated with the longest path. At the beginning of the execution, it allocates that many memory regions, each having just enough memory to store the max-degree number of vertex IDs. Dryadic reuses these regions to minimize the memory management overhead.

2) Code Generation for Parallel and Distributed Execution:

a) Mapping to nested loops.: As prior systems [32], [1] show, generating and pre-compiling code specialized for specific patterns typically has high performance as it avoids runtime overhead to handle general patterns. For example, one can write a three-level nested for loop to only match the unlabeled triangle pattern, while a general pattern matching system has to at least determine the routine to execute to match the pattern and sometimes even needs to run isomorphism check to determine the pattern of enumerated embeddings [49], [53], [8].

The computation tree naturally maps to a nested loop structure, with each pattern vertex corresponding to a loop. Its children, if any, correspond to consecutive nested loops in its loop body. The outer-most loop traverses all the vertices, while each iteration of the innermost loop computes a set of vertices to match the last pattern vertex. Each of the loops in-between traverses over a vertex set computed by the compound set operation associated with its pattern vertex. We point out three properties of the generated nested loop structure. First, it explores the embedding trees in the depth-first order. Second, an iteration of any inner loop explores a sub-embedding tree. Third, every loop is parallelizable because the iterations of the same loop explore distinct sub-embedding trees, which do not depend on each other.

b) Parallel and Distributed Execution.: Nested loop structures parallelize naturally at the outermost level using standard tools like OpenMP parallel for. In this case, the outer loop corresponds to a single vertex per iteration, and each inner loop adds a vertex to the partial embedding. Because of this property, the execution time of an iteration of the outermost loop is sensitive to the degree of every vertex that appears in a particular embedding. On the LiveJournal [56] graph, the most expensive vertex iteration of motif_4 enumeration takes as long as the cheapest 96.6% of the vertex iterations, accounting for over 7.3% of the total CPU time as a single iteration among 4 million. Obviously, this poses some load balance challenges, which are exacerbated as the pattern size increases.

In a distributed environment, we assign work to machines on the basis of a fair distribution of edges. On an individual machine, the OpenMP dynamic scheduler handles the task of assigning work to threads. Using it with a task granularity of 64 introduces minimal overhead, but helps mitigate the load balance issue by ensuring no one thread has to do excessive work. There is still an upper limit to how well this approach can handle load balance. However, due to the nature of the pre-compiled nested loop structure, it cannot easily interact with the stealing runtime. We leave it to future work.

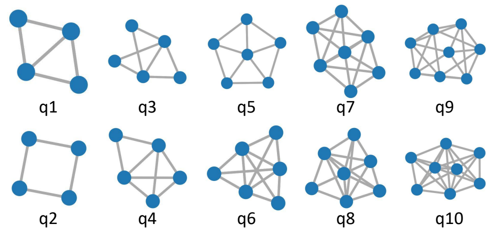


Fig. 5: Pattern queries used in the evaluation.

IV. Evaluation

In this section, we evaluate the effectiveness of the computation tree and the efficiency of the backends in Dryadic.

A. Methodology

Compared systems. We compare Dryadic with five state-of-the-art systems published in the past two years. **DAF** [18] is the fastest graph pattern matching system for labeled graphs, which avoids exploring failing paths in the embedding tree. **AutoMine** [32] and **Pangolin** [8] are the fastest single-machine graph pattern mining systems and can also be used for pattern matching. They are particularly good at parallel multi-pattern matching. Note that although Peregrine [20] and DwarvesGraph [6] claim to outperform several other systems for multi-pattern matching, they actually perform pattern counting instead of enumeration. We hence exclude them for comparison. **CECI** [3] [1] is the only open-source distributed graph pattern matching system, which claims to outperform multiple single-machine systems, including PsgL [45] and DUALSIM [22]. In addition, we also compare Dryadic with **PGD** [2], a state-of-the-art manual implementation of motif counting. We use PGD to only count connected patterns for a fair comparison.

Datasets and patterns. Table I shows the 9 real-world graphs used in the experiments. Most are from the Stanford SNAP collection of datasets [30], representing a sampling of online social network, interaction, and collaboration graphs. Most of the graphs do not have labels. For consistency, we randomly assign one of 10 distinct labels to each vertex. We use ten non-clique query patterns of five different sizes shown in Figure 5 with randomly generated labels assigned to the pattern vertices. The experiments to compare Dryadic with DAF use these patterns. The other experiments either use clique patterns or all the connected patterns of a certain size.

Machine environment. Our single machine experiments run on a system with two Intel Xeon E7-4830 v3 CPUs (hyperthreading disabled) and 256GB of memory. The system runs Ubuntu 18.04 with Linux kernel 4.15 and compiles with GCC version 7.5 at optimization level -O3. Our distributed experiments run on a cluster of machines each with two Intel Xeon E5-2670 CPUs (hyperthreading disabled) and 64GB of memory, running CentOS version 6.3 with Linux kernel 2.6 and compiling with GCC version 4.4.7 at optimization

[1]The released CECI system (https://github.com/iHeartGraph/ceci-release) cannot produce correct results, as confirmed by the authors on Aug. 9th, 2020. As of April. 19, 2021, they have not fixed the bugs.

Graphs	#Vertices	#Edges	Description
WikiVote [28]	7K	100K	Wiki editor voting
Enron [24]	37K	183K	Email network
Amazon [56]	334K	926K	Product network
DBLP [56]	317K	1M	Collaboration network
MiCo [12]	96K	1.1M	Co-authorship
Patents [29]	3.8M	16.5M	US Patents
LiveJournal [56]	4M	34.7M	Social network
Orkut [56]	3.1M	117.2M	Social network
Friendster [56]	65.6M	1.8B	Social network

TABLE I: Graph Datasets.

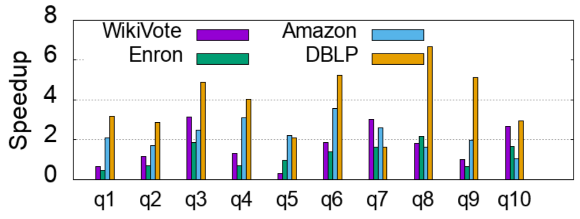


Fig. 6: Results on labeled pattern matching on four graphs.

level -O3. It uses OpenMPI version 3.0.0 to drive the QDR InfiniBand interconnect.

B. Labeled Pattern Matching

We compare Dryadic's generated code with DAF by running the ten patterns in Figure 5 on the first seven graphs in Table I. Edge-induced matching is used because DAF does not support vertex-induced matching. Figure 6 shows the speedups of Dryadic over DAF on the four smaller graphs (i.e., WikiVote, Enron, Amazon, and DBLP). Out of the 40 runs, Dryadic is faster than DAF for 32 runs. Dryadic's average speedups over DAF are 1.34X, 1.47X, and 3.1X for WikiVote, Amazon, and DBLP, respectively. DAF achieves a 1.14X speedup over Dryadic for Enron. However, DAF's performance significantly drops when processing reasonably large graphs. As Figure 7 shows, the performance gap between DAF and Dyradic increases substantially for the three larger graphs. We only show the results for six patterns because DAF times out at 20 minutes for the others. On average, Dryadic outperforms DAF by 8.4X, 37.4X, and 17.7X for Mico, Patents, and LiveJournal, respectively.

By investigating DAF's slowest runs, we identified two problems. First, DAF's auxiliary data structure consistently consumes a large amount of memory. Figure 8 reports the peak memory consumption of DAF and Dryadic for the executions on the three larger graphs. For the largest graph, LiveJournal,

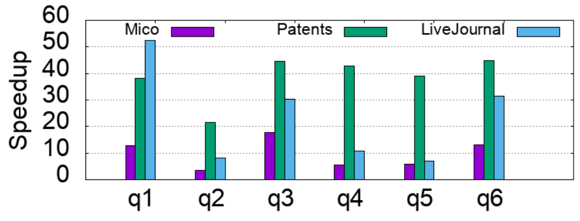


Fig. 7: Results on labeled pattern matching on three graphs.

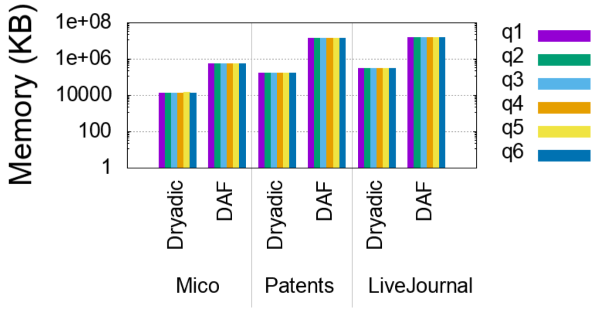


Fig. 8: Memory consumption for labeled pattern matching on three graphs.

Graph	Size	AutoMine	Pangolin	Dryadic-C	Dryadic-G
WikiVote	4	48.2	16.96	1.13	2.19
	5	4182.95	13827.8	454.45	719.47
Enron	4	35.3	29.26	1.24	2.38
	5	10363	34768.5	607.76	1667.18
Amazon	4	0.47	1.14	0.13	0.53
	5	218.53	160.12	7.58	31.47
DBLP	4	0.84	3.87	0.36	0.88
	5	69	734	51.2	41.9
Patents	4	24.4	95	6.18	14.7
	5	2867	24069	1680	748
Mico	4	28.2	111	9.65	16.7
LiveJournal	4	5286	19741	881	1081

TABLE II: Performance comparison between AutoMine, Pangolin, and Dryadic on motif enumeration in seconds.

DAF consumes on average 50X more memory than Dryadic. DAF requires about 16GB memory to process LiveJournal, while the graph data consumes only 296 MB. In comparison, Dryadic only requires 323 MB for the same workload. Second, DAF incurs a large number of recursive calls. For example, DAF reports about 447M recursive calls when running $q5$ on LiveJournal, but Dryadic's nested loop does not invoke any recursive functions.

C. Single-Machine Unlabeled Pattern Matching

We compare Dryadic (both its compilation mode, Dryadic-C, and its Galois-based interpretation mode, Dryadic-G) with AutoMine and Pangolin. To evaluate these systems' capability of performing multi-pattern matching, we run size-4 and size-5 motif enumeration (vertex-induced matching), which enumerates all the embeddings for each connected pattern of the given size. We omit the results when Pangolin times out at ten hours. As Table II shows, Pangolin is the slowest system due to its breadth-first execution, which needs to maintain all enumerated sub-embeddings. Although both Dryadic-C and AutoMine's codes are generated, Dryadic-C is faster than AutoMine in all runs with up to 42.8X speedup (Motif-4 on WikiVote). This is mainly because AutoMine suffers from redundant set operations, while Dryadic eliminates such redundancy. Dryadic-G is substantially slower than Dyradic-C in most runs due to its runtime overhead except for the largest graphs where the fine-grained stealing produces the largest benefit.

Figure 9 shows the results of finding cliques of 3 different sizes. Results for some experiments are omitted because

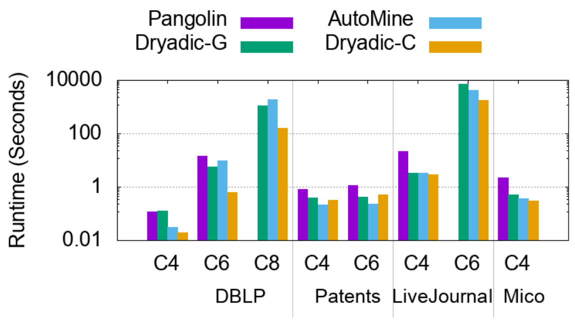


Fig. 9: Performance comparison between AutoMine, Pangolin, and Dryadic on clique finding. The two missing bars indicate that the corresponding experiments for Pangolin run out of memory.

Graph	Dryadic-C	PGD
WikiVote	0.1	0.22
Enron	0.13	0.26
Amazon	0.04	0.21
DBLP	0.08	0.31
Mico	0.4	2
Patents	0.96	14.84
LiveJournal	19.96	260.5

TABLE III: Performance comparison between Dryadic and PGD on size-4 motif counting in seconds.

Pangolin ran out of memory. Pangolin is again the slowest system but the performance gap between it and other systems is smaller. The major reason is that for single-pattern matching Pangolin does not incur as much space overhead as for multi-pattern matching. Since AutoMine applies an aggressive optimization to load only half of the data graph, it is faster than Dryadic-C in 7 out of 8 experiments due to its better cache performance from the smaller working set. Despite running on the entire graph, Dryadic-C outperforms AutoMine for most heavy workloads by up to 2.1X (size-5 clique finding on LiveJournal).

Table III shows the performance comparisons between Dryadic-C with PGD, a state-of-the-art manual implementation of size-4 motif-counting. The results on Orkut and Friendster are omitted because PGD times out at ten hours. Both Dryadic and PGD count the embeddings of rectangle and size-4 clique patterns, and use the same formulas to derive the counts of the other patterns (details of the formulas in [2]). The performance improvement of Dryadic-C over PGD ranges from 1.97X to 15.5X with an average of 5X. Observe the trend that the performance gap tends to widen with larger input graphs. The advantage of Dryadic over PGD comes from two sources. First, Dryadic uses symmetry breaking to avoid over-counting, while PGD may identify the same embedding multiple times. Second, Dryadic's code motion eliminates all redundant set operations, which is not achievable in PGD. Although one can manually implement Dryadic's optimizations in PGD, doing so requires significant effort and is limited to one specific workload (i.e., size-4 motif counting). In comparison, Dryadic's approach is general and applies to arbitrary patterns.

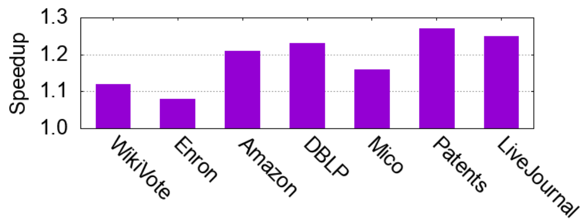
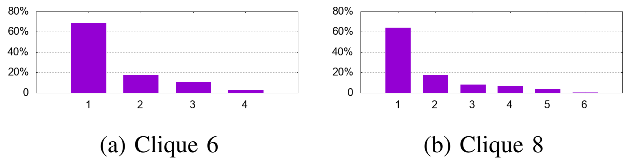


Fig. 10: Performance benefits from fine-grained stealing for motif-4 enumeration.



(a) Clique 6 (b) Clique 8

Fig. 11: Level distribution of stolen sub-embedding trees.

a) Effects of Work Stealing.: Figure 10 shows the performance benefits of Dryadic's fine-grained work stealing on motif-4 enumeration. The speedup due to work stealing is up to 1.27X, demonstrating the serious load imbalance problem across the embedding trees. We also use size-6 and size-8 clique finding on DBLP to investigate the efficiency of the work stealing. Recall that the stealing runtime tries to steal a sub-embedding tree rooted at the highest possible level for each attempt. For each run, we record the root level of each stolen sub-embedding tree. Figure 11 reports the level frequency distribution. For size-6 clique finding, 69% of the sub-embedding trees are rooted at level 1. On average, sub-embeddings rooted at higher levels should be larger, so stealing at higher levels minimizes the relative synchronization overhead of stealing.

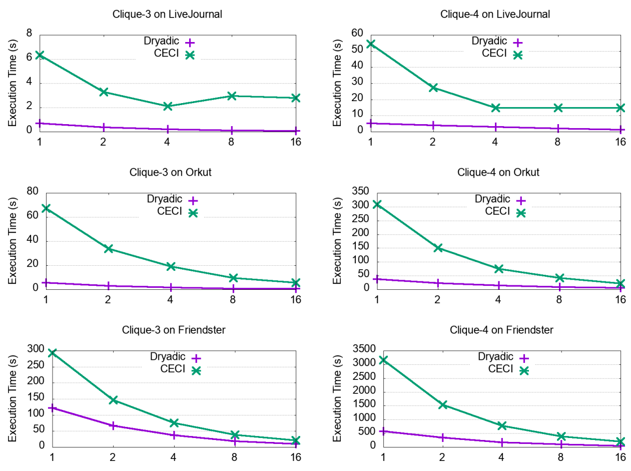


Fig. 12: Performance comparisons between Dryadic and CECI for distributed execution with 1, 2, 4, 8, and 16 machines.

D. Distributed Unlabeled Pattern Matching

In this part, we compare Dryadic with CECI for distributed execution. Since the released CECI system has bugs as mentioned earlier, we use its results collected on a 16-node cluster in the paper [3]. Each node in the cluster has two 8-core Intel Xeon E5-2650 CPUs and 128 GB memory. The CPUs are comparable to the ones in our system. Though our system has less memory, it should not affect the comparisons by much because all the evaluated graphs can already fit in memory. Figure 12 shows the performance benefits of Dryadic over CECI on size-3 and size-4 clique finding using 1, 2, 4, 8, and 16 machines. Due to space limit, we omit the results on other patterns, but they show similar trends. Dryadic substantially outperforms CECI for all the distributed runs. Using 16 machines, Dryadic is on average 20X and 6.7X faster than CECI for size-3 and size-4 clique finding, respectively. Dryadic's single-node performance is much better thanks to the aggressive optimization applied to the computation tree. The sub-figures demonstrate the scalability of the two systems. For size-3 clique finding, Dryadic and CECI have similar scalability for Orkut and Friendster, achieving around 11X-12X speedups by using 16 machines over one machine. But CECI's scalability on LiveJournal drops significantly beyond four machines. A plausible reason is that CECI cannot well address the load balance problem. For size-4 clique finding, CECI has slightly better scalability for Orkut and Friendster, but again fails to scale beyond four machines for LiveJournal.

V. Related Work

General graph analytics frameworks. Numerous graph processing systems have been proposed in recent years to optimize irregular computation [46], [39], minimize communication [15], reduce redundant computation [52], and improve locality [55]. However, such infrastructure-level optimizations cannot take advantage of the unique properties of graph pattern matching workloads. As shown by the Galois backend in Dryadic, a promising direction is to integrate Dryadic with those systems for them to complement each other.

Labeled pattern matching. Practical graph pattern matching can be traced back to Ullmann's backtracking algorithm [51], which proposes the basic approach to iteratively matching pattern vertices based on certain orders. A number of studies follow it to optimize the matching orders to improve performance [10], [44], [58], [34]. As pointed out by Lee [27], the best matching orders depend on the local topology and label distribution within a data graph. Inspired by this finding, TurboIsO [19] and CFL-Match [4] build a query tree using the given pattern and adaptively change the matching order within the same run. However, the query tree introduces numerous false positives and hence redundant computation. DAF [18] addresses this problem by performing matching through a DAG based on the pattern while still supporting adaptive matching orders. Dyradic uses a static matching order and focuses on optimizing the computation tree instead of input adaptation, but it is interesting to combine the two methods in the future.

Single-machine parallel pattern mining. Many studies focus on parallel execution efficiency for graph pattern matching [22], [47], [48]. Chen et al. [8] identifies that a critical reason for existing systems' poor performance is their inefficient implementations of parallel operations and data structures. They closely integrate the Pangolin system with the Galois engine to outperform several systems, including G-Miner [5], Kaleido [57], and Fractal [11]. However, Pangolin's breadth-first execution does not exploit the structure of the given patterns, which is to some degree addressed by AutoMine [32] and Peregrine [20].

Distributed pattern matching. Thanks to the massive parallelism in graph pattern matching, many systems use a distributed system to accelerate the execution. A popular approach is to duplicate the data graph in each node and focus on load balancing for optimization [49], [3]. RADS [41] partitions the graph into multiple machines, which employs a framework of region-grouped multi-round expand verify & filter to reduce communication and minimize the intermediate result storage. BENU [54] implements a global caching technique to exploit data sharing. Lai et al. [26] use the Timely dataflow system [35] to evaluate multiple graph pattern matching algorithms and propose a practical guide.

Continuous pattern matching. Fan et al. [13] propose the first system for continuous graph pattern matching for dynamic graphs. Given an edge change, the system computes the largest subgraph, whose vertices and edges may form embeddings with that change, and runs the pattern matching algorithm on that subgraph with the change. GraphFlow [21] supports continuous pattern matching in a graph database. Different from these two systems, SJ-Tree [9] aggressively stores the intermediate results to improve responsiveness, but is impractical to process large graphs. TurboFlux [23] strikes a better trade-off between space overhead and responsiveness thanks to its data-centric graph structure to more efficiently store and update the intermediate results. Extending Dryadic to handle dynamic graphs is an interesting research direction.

VI. Conclusion

A number of graph pattern matching systems have been proposed in recent years. We showed that they are all specialized for certain settings and no system consistently outperforms the others. We argued that a systematic approach is needed to build a flexible and efficient graph pattern matching system. We presented, Dryadic, a graph pattern matching system, the techniques in which are centered around a powerful intermediate representation. Extensive experiments showed that the backends in Dryadic enable it to substantially outperform five state-of-the-art graph pattern matching systems.

References

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.

[2] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining, ICDM 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 1–10, 2015.

[3] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: compact embedding cluster index for scalable subgraph matching. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1447–1462. ACM, 2019.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1199–1214. ACM, 2016.

[5] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 32:1–32:12, 2018.

[6] Jingji Chen and Xuehai Qian. Dwarvesgraph: A high-performance graph mining system with pattern decomposition, 2020.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 1:1–1:15, 2015.

[8] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on CPU and GPU. *Proc. VLDB Endow.*, 13(8):1190–1205, 2020.

[9] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, pages 157–168, 2015.

[10] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[11] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1357–1374, 2019.

[12] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. GRAMI: frequent subgraph and pattern mining in a single large graph. *PVLDB*, 7(7):517–528, 2014.

[13] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3):18:1–18:47, 2013.

[14] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[16] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In Terence P. Speed and Haiyan Huang, editors, *Research in Computational Molecular Biology, 11th Annual International Conference, RECOMB 2007, Oakland, CA, USA, April 21-25, 2007, Proceedings*, volume 4453 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007.

[17] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, 2007.

[18] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1429–1446. ACM, 2019.

[19] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo$_{\mathrm{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 337–348. ACM, 2013.

[20] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: a pattern-aware graph mining system. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 13:1–13:16. ACM, 2020.

[21] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698. ACM, 2017.

[22] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. DUALSIM: parallel subgraph enumeration in a massive graph on a single machine. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1231–1245. ACM, 2016.

[23] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 411–426. ACM, 2018.

[24] Bryan Klimt and Yiming Yang. Introducing the enron corpus.

[25] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, 2012.

[26] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, 2019.

[27] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.*, 6(2):133–144, 2012.

[28] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.

[29] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 177–187, 2005.

[30] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[31] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: Breaking symmetry for efficient graph mining. *CoRR*, abs/1911.12877, 2019.

[32] Daniel Mawhirter and Bo Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523. ACM, 2019.

[33] Antonio Messina, Antonino Fiannaca, Laura La Paglia, Massimo La Rosa, and Alfonso Urso. Biograph: a web application and a graph database for querying and analyzing bioinformatics resources. *BMC Systems Biology*, 12(5):98, 2018.

[34] Amine Mhedbhi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[35] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455, 2013.

[36] Neo4j. Fraud detection: Discovering connections with graph databases. https://www.whitepapers.online/fraud-detection-discovering-connections-graph-databases/, 2014.

[37] Neo4j. The neo4j graph database system. https://neo4j.com/, 2020.

[38] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013.

[39] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 622–636, New York, NY, USA, 2018. ACM.

[40] Steven Noel. A review of graph approaches to network security analytics. In Pierangela Samarati, Indrajit Ray, and Indrakshi Ray, editors, *From Database to Cyber Security - Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*, volume 11170 of *Lecture Notes in Computer Science*, pages 300–323. Springer, 2018.

[41] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and robust distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, 2019.

[42] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, 2013.

[43] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.*, 29(2):595–618, 2020.

[44] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.

[45] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 625–636. ACM, 2014.

[46] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[47] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. Efficient parallel subgraph enumeration on a single machine. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 232–243. IEEE, 2019.

[48] Shixuan Sun and Qiong Luo. Scaling up subgraph query processing with efficient subgraph matching. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 220–231. IEEE, 2019.

[49] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 425–440, 2015.

[50] TigerGraph. The tigergraph graph database system. https://www.tigergraph.com/, 2020.

[51] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[52] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

[53] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on A single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 763–782, 2018.

[54] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. BENU: distributed subgraph enumeration with backtracking-based framework. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 136–147. IEEE, 2019.

[55] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[56] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[57] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Xueqi Cheng. Kaleido: An efficient out-of-core graph mining system on A single machine. *CoRR*, abs/1905.09572, 2019.

[58] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1):340–351, 2010.