An Efficient System for Subgraph Discovery

Aparna Joshi, Yu Zhang, Petko Bogdanov, and Jeong-Hyon Hwang
Department of Computer Science, University at Albany—SUNY
{ajoshi, yzhang20, pbogdanov, jhwang}@albany.edu

***Abstract*—Subgraph discovery in a data graph (finding subsets of vertices and edges satisfying a user-specified criteria) is an essential and general graph analytics operation with a wide spectrum of applications. We present Nuri, a general subgraph discovery system that allows users to succinctly specify subgraphs of interest and criteria for ranking them. Given such specifications, Nuri efficiently finds the k most relevant subgraphs. It prioritizes (i.e., expands earlier than others) subgraphs that are more likely to expand into the desired subgraphs (*prioritized subgraph expansion*) and proactively discards irrelevant subgraphs from which the desired subgraphs cannot be constructed (*pruning*). Nuri can also efficiently store and retrieve a large number of subgraphs on disk without being limited by the size of main memory. We demonstrate using both real and synthetic datasets that Nuri only on a single core outperforms the closest alternative distributed system running on 40 cores by more than 2 orders of magnitude for clique discovery and 1 order of magnitude for subgraph isomorphism and pattern mining.**

***Keywords*-subgraph discovery, prioritization, pruning**

I. Introduction

Social networks, the World Wide Web, transportation networks and protein-protein interaction (PPI) networks are commonly modeled as *graphs* in which *vertices* represent entities, *edges* represent relationships between entities, and vertex/edge labels represent certain properties of the entities/relationships. Given such a graph, the problem of finding *subgraphs* (subsets of vertices and edges) that meet specific user-defined criteria arises in a variety of applications. For example, prominent star and clique structures of high homophily (i.e., sharing attributes) in social networks have been shown instrumental to understanding the nature of society [1]. In the biological domain, subgraphs in a PPI network of highest gene expression disagreement across phenotypes (e.g., healthy/sick) are essential for identifying target pathways and complexes to manipulate a condition [2]. Other applications include computer network security [3], financial fraud detection [4], and community discovery in social and collaboration networks [5].

In response to the aforementioned demand, researchers have developed custom solutions for specific types of subgraph discovery problems. Examples include subgraph isomorphism search algorithms [6], [7], [8] and techniques for discovering frequent subgraphs [9], [10], cliques [11], [12], [13], [14], quasi-cliques and dense subgraphs [15], as well as communities [5]. The above techniques, however, are one-off solutions for specific problems and are usually difficult to use/extend for different subgraph discovery computations.

Systems specifically targeted to subgraph discovery have recently been proposed [16], [17], [18]. These systems initially construct one-vertex (or one-edge) subgraphs and then repeatedly expand subgraphs into larger ones by adding a vertex or an edge at a time. As discussed later in this paper, however, these systems often exhibit limited performance (even when they employ a large number of servers) mainly due to the sheer number of subgraphs that they have to examine. They may also produce an extremely large result set (e.g., millions of subgraphs) which a human analyst cannot easily deal with.

We propose a new subgraph discovery system, called Nuri, that overcomes the above limitations. Nuri supports various computations (i) *conveniently* (as opposed to the complexity of developing custom solutions) via an API that enables succinct implementation of these computations and (ii) more *efficiently* when compared to the closest existing alternative systems [16], [17], [18] by quickly finding the k most relevant subgraphs according to user-provided specifications.

The key advantageous features of Nuri are as follows:

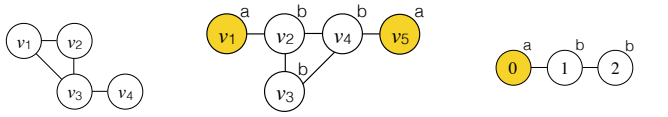
API. Our API allows users to enable the desired subgraph discovery computation as soon as they implement *only one* function which determines whether or not a given subgraph matches their interest. They do not need to write code for creating and expanding subgraphs as well as dealing with situations where there are too many subgraphs to fit into the memory. To speed up the computation, users may implement additional functions for the optimizations explained below.

Prioritization. Users can implement a function in our API to assign a higher priority to subgraphs that are more likely to expand into subgraphs of interest than others (e.g., cliques with the potential to expand into larger cliques). Given this function, Nuri expands subgraphs with a higher priority before other subgraphs, thereby quickly finding the desired subgraphs. The previous subgraph discovery systems [16], [17], [18] lack this feature and thus have inherent performance limitation.

Pruning. As soon as the result set contains k entries, it becomes unnecessary to expand subgraphs whose expansions cannot lead to subgraphs that are more relevant than these k entries (e.g., in the case of finding the largest cliques, when the result set already contains a clique of size 4, no need to consider cliques that can only expand into cliques of up to size 3). Nuri can detect and safely discard such subgraphs according to a user-specified function.

Targeted Expansion. Users can implement a function in our API to specify whether or not it is adequate to expand a subgraph by adding a vertex or an edge. This feature enables Nuri to create and examine only the necessary subgraphs in contrast to Arabesque [16] which exhaustively creates subgraphs and then filters out irrelevant ones.

Efficient Top-k Aggregate Subgraph Discovery. While some key subgraph discovery computations require grouping of subgraphs (e.g., by their patterns) and aggregation of certain properties (e.g., calculation of pattern frequency) [9], [10], previous subgraph discovery systems such as NScale [17] and G-Miner [18] cannot support such *aggregate* computations. In contrast to Arabesque [16] that must expand all smaller subgraphs before any larger subgraph, Nuri can expand a



(a) Graph with 4 vertices (b) Graph with 5 vertices (a and b are labels) (c) Query/pattern graph

Fig. 1: Sample Graphs

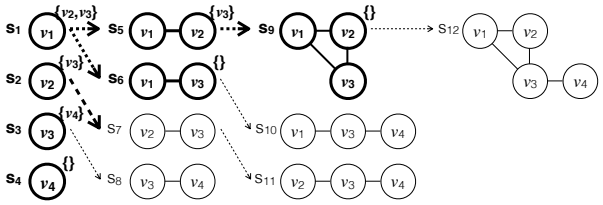


Fig. 2: Clique Discovery (on the graph in Fig. 1a)

group of subgraphs before other smaller subgraphs (*prioritized expansion*) thereby facilitating *early* and *effective pruning*.

On-Disk Subgraph Management. The number of subgraphs that Nuri manages usually grows exponentially with the size and density of the data graph and may even surpass the capacity of the main memory. Nuri has the ability to manage high-priority subgraphs in memory and low-priory subgraphs on disk in a highly efficient manner where the use of disk causes only a slight degradation in performance.

This paper makes the following contributions:

- Two new computational models that efficiently support various top-k subgraph discovery computations through prioritized subgraph expansion and pruning
- An API that allows users to easily implement diverse subgraph discovery computations (and examples demonstrating succinct implementation of representative subgraph discovery algorithms)
- Design and implementation of a system, Nuri, that enables fast top-k subgraph discovery just on a single computer
- In-depth analysis of experimental results which demonstrate between 1 to 2 orders of magnitude reduction of subgraph discovery time for our system running on a single core, compared to the closest alternative distributed system utilizing 40 cores.

II. Background

In this section, we discuss three popular subgraph discovery computations that we adopt to describe and evaluate our general system (Sec. II-A). We also explain previous systems/solutions that are closely related to our work and their limitations (Sec. II-B).

A. Subgraph Discovery Computations

Our goal is to enable efficient discovery of the most relevant subgraphs that meet user-specified criteria within a large data graph. For demonstrative purposes, we adopt the following example discovery computations:

Clique Discovery. A clique is a subgraph in which every pair of vertices are adjacent [11], [12], [13], [14]. The data graph

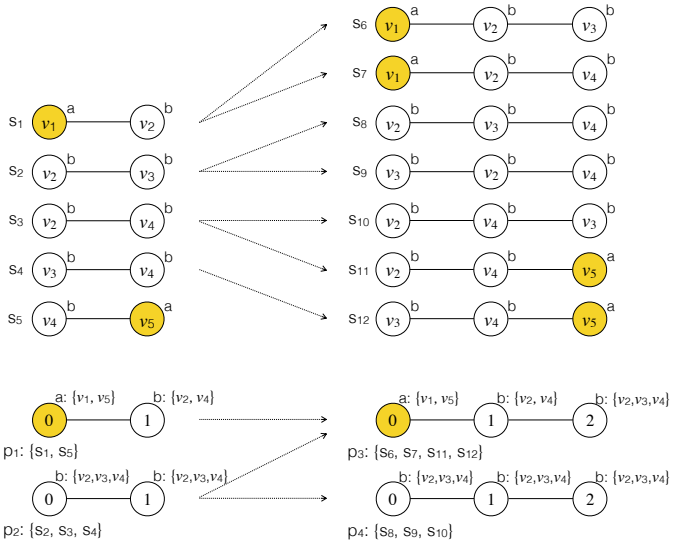


Fig. 3: Pattern Mining (on the graph in Fig. 1b)

in Fig. 1a contains 9 cliques of sizes 1 to 3, depicted as s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, and s_9 in Fig. 2 (details of this figure are explained in Sec. II-B and III-A).

Subgraph Isomorphism. The goal of subgraph isomorphism search is to find, in a labeled data graph, all subgraphs isomorphic to a query graph [6], [7], [8]. A subgraph $G_s(V_s, E_s, L_s)$ in a data graph is isomorphic to a query graph $G_q(V_q, E_q, L_q)$ if there exists a bijection $\mu : V_q \rightarrow V_s$ such that (1) $\forall v \in V_q, L_q(v) = L_s(\mu(v))$ (i.e., vertex v in G_q and the corresponding vertex $\mu(v)$ in G_s have the same label) and (2) $\forall (v, v') \in E_q \Leftrightarrow (\mu(v), \mu(v')) \in E_s$ (i.e., vertices v and v' are adjacent in G_q if and only if the corresponding vertices $\mu(v)$ and $\mu(v')$ are adjacent in G_s). For example, the graph in Fig. 1b contains four subgraphs (depicted as s_6, s_7, s_{11}, and s_{12} in Fig. 3) that are isomorphic to the query graph in Fig. 1c. Subgraph s_6 is isomorphic to the query graph since a bijection such that $\mu(0) = v_1$, $\mu(1) = v_2$, and $\mu(2) = v_3$ satisfies the conditions mentioned above.

Pattern Mining. The goal of pattern mining is to find subgraph patterns that appear at least as frequently as a user-specified threshold in the data graph. For example, the subgraph pattern in Fig. 1c appears in the data graph from Fig. 1b (the subgraphs depicted as s_6, s_7, s_{11}, and s_{12} in Fig. 3 are isomorphic to the pattern in Fig. 1c).

Among several pattern frequency definitions, for the purposes of this example, we consider the *minimum image-based support*, which is defined as the minimum number of mappings for any vertex in the pattern to the corresponding vertices in the data graph [19]. According to this definition, the frequency of pattern p_1 in Fig. 3 is 2 since (i) subgraphs s_1 and s_5 match (i.e., are isomorphic to) p_1 and (ii) the first vertex of p_1 (vertex 0) maps to vertices v_1 and v_5 in Fig. 1b, (iii) the second vertex of p_1 (vertex 1) maps to vertices v_2 and v_4, and thus (iv) the frequency of p_1, denoted $f(p_1)$, is $\min(|\{v_1, v_5\}|, |\{v_2, v_4\}|) = \min(2, 2) = 2$. The other patterns in Fig. 3 have the following frequencies: $f(p_2) = \min(|\{v_2, v_3, v_4\}|, |\{v_2, v_3, v_4\}|) = 3$, $f(p_3) = \min(|\{v_1, v_5\}|, |\{v_2, v_4\}|, |\{v_2, v_3, v_4\}|) = 2$, and $f(p_4) = \min(|\{v_2, v_3, v_4\}|, |\{v_2, v_3, v_4\}|, |\{v_2, v_3, v_4\}|) = 3$.

TABLE I: User Functions

Function	Optional	Description	Default
$expandable(s, \delta)$	yes	returns `true` if it is adequate to expand subgraph s by adding a vertex or an edge δ	returns `true`
$relevant(s)$	no	returns `true` if subgraph s matches the user's interest	N/A
$priority(s)$	yes	returns the application-specific priority of subgraph s	returns `null`
$dominated(s, s')$	yes	returns `true` if all subgraphs into which s can expand are guaranteed to have a lower priority than subgraph s'	returns `false`

Top-k Semantics. The above computations may return an enormous number of subgraphs, overwhelming the user. Our goal is to allow (i) users to specify the desired size k of the result set and a criterion for ranking subgraphs and (ii) the system to obtain such top-k results more efficiently than exhaustively acquiring all results and then ranking them.

B. Limitations of Current Graph Systems

TLV (Think Like a Vertex) graph processing systems such as Pregel [20], GraphLab [21], TurboGraph [22] iteratively update the state (i.e., variables) of each vertex in a manner that eventually computes quantities of interest, such as PageRank [23]. Subgraph discovery computations, however, cannot be succinctly expressed using vertex variables since the number of subgraphs grows exponentially with the size of the graph and there is typically a many-to-many relationship between vertices and subgraphs. For this reason, TLV systems cannot adequately support subgraph discovery computations.

Systems specifically targeted to subgraph discovery have recently been proposed [16], [17], [18]. These *subgraph discovery* systems initially construct one-vertex subgraphs (e.g., s_1, s_2, s_3, and s_4 in Fig. 2) or one-edge subgraphs (e.g., s_1, s_2, s_3, s_4, and s_5 in Fig. 3) and then expand subgraphs into larger ones by adding a vertex or an edge at a time (e.g., in Fig. 3, s_1 into s_6 by adding edge $\{v_2, v_3\}$). The limitations of these systems are as follows: (i) As further explained in Sec. III-A, they cannot perform *prioritized expansion* of subgraphs according to user-specified criteria, inherently losing the opportunity to quickly fill the result set and start pruning out irrelevant subgraphs whose expansions cannot affect the result set. (ii) NScale [17] and G-Miner [18] cannot support aggregate computations (e.g., finding the frequency of a pattern based on the subgraphs having that pattern). (iii) Arabesque [16] adopts *exhaustive expansion* (i.e., constructs all subgraphs that can be obtained by adding a vertex/edge to an existing subgraph) and *post-expansion filtering* (i.e., discards irrelevant subgraphs), and thus may create a large number of unnecessary subgraphs (e.g., non-clique subgraphs such as s_{10}, s_{11}, and s_{12} in Fig. 2).

III. Computational Model

We present the computational model (Sec. III-A) and its extension for aggregate computations (Sec. III-B).

A. Basic Computational Model

Our computational model aims to quickly find the k-most relevant results according to user-specified functions. Its key principles are as follows:

- *Targeted expansion*: our computational model allows users to specify if it is adequate to expand subgraph s by adding a vertex or an edge δ (see the $expandable(s, \delta)$ function in Table I). For example, users can avoid creation of non-clique subgraphs by making $expandable(s, \delta)$ return `true` only if adding δ to s leads to a clique.
- *Result ranking*: by implementing the $relevant(s)$ function (Table I), users can specify whether or not subgraph s matches their interest (e.g., s is a clique) and thus may be added to the result set. Users can also incorporate their criteria for ranking results into the $priority(s)$ function (e.g., larger cliques assigned a higher priority value).
- *Pruning*: as explained below, it may be possible to calculate an *upper bound* on the priorities of all possible subgraphs into which a subgraph s can expand (e.g., expansions of s would only lead to cliques of size 3 or less). In this case, users can instrument the $dominated(s, s')$ function to return `true` if all supergraphs that s can expand into are guaranteed to have a lower priority than s'. When s' is the k-th entry in the result set and $dominated(s, s')$ returns `true`, it is safe to ignore subgraph s since expansions of s can never affect the result set (i.e., all of the subgraphs obtained through these expansions would have a lower priority than s' and thus never be included in the result set).
- *Prioritized expansion*: our model expands highest priority subgraphs first. This feature allows users to implement the $priority(s)$ function in a manner that quickly fills the result set and facilitates pruning.

Algorithm 1: Basic Computational Model

```
1  for vertex v : V or edge e : E do
2      create a subgraph s containing only vertex v or edge e;
3      insert s into Q;
4  while |Q| > 0 do
5      subgraph s ← remove_max(Q);
6      if relevant(s) and
7          (|R| < k or priority(s) ≥ priority(k-th(R))) then
8          insert s into R;
9          if |R| > k then
10             remove from R each entry ϵ such that
               priority(ϵ) < priority(k-th(R));
11     if |R| < k or !dominated(s, k-th(R)) then
12         for each neighboring vertex (or edge) δ of s do
13             if expandable(s, δ) then
14                 create subgraph s' by adding δ to s;
15                 if |R| < k or !dominated(s', k-th(R)) then
16                     insert s' into Q;
```

Algorithm 1 illustrates how our computational framework carries out subgraph discovery computations. It first creates,

for each vertex (or edge) in the data graph, a subgraph s containing that vertex (or edge) and inserts s into a priority queue Q (lines 1-3). Next, as long as Q contains subgraphs (line 4), it repeatedly dequeues and processes the subgraph s with the highest priority (lines 5-16). If s matches the user's interest (line 6) and if the result set R is not yet full or s has no lower priority than the k-th entry in R (line 7), it adds s to R (line 8) and removes unnecessary entries from R (lines 9 and 10). Furthermore, it examines if subgraph s can be safely ignored (i.e., unnecessary to expand s) (line 11). If not, it considers each neighboring vertex (or edge) δ of s (line 12). If adding δ to s is adequate (line 13; e.g., this expansion will lead to a clique), it expands s into s' by adding δ (line 14). If s' cannot be ignored (line 15), it inserts s' into Q (line 16).

Example: Maximum Clique Discovery. Assume that a user wants to quickly find the largest clique(s) given a data graph in Fig. 1a. For each clique s, the user can consider the set P_s of vertices that can be added to s while forming a larger clique [11] (for details of P_s, refer to Listing 1). For example, in Fig. 2, $P_{s_1} = \{v_2, v_3\}$ since adding v_2 and its edge $\{v_1, v_2\}$ to s_1 leads to clique s_5 and adding v_3 and $\{v_1, v_3\}$ to s_1 leads to s_6. On the other hand, $P_{s_2} = \{v_3\}$ because s_2 has only one such neighboring vertex (v_3) (note that, just like Arabesque [16], our framework does not consider adding vertex v_1 to s_2 since this expansion would result in a duplicate generation of s_5 which is to be obtained by adding vertex v_2 to s_1). The user can enable the desired computation as follows (for the actual implementation of the custom functions, refer to Sec. IV-A):

- *Targeted expansion*: the user can avoid creation of non-clique subgraphs by making $expandable(s, \delta)$ return `true` only when vertex δ is in P_s (i.e., adding δ and its relevant edges to s surely leads to a clique). In this case, every subgraph s obtained through expansion is a clique (i.e., matches the user's interest) and therefore $relevant(s)$ needs to always return `true`.
- *Prioritized expansion*: the user can implement $priority(s)$ so that it returns $(|V_s|, |P_s|)$ where V_s is the set of vertices in s and P_s is the set of vertices that can be added to s while forming a larger clique. When lexicographic ordering is applied to such priority values, our framework expands larger cliques before smaller cliques and, for cliques of the same size, expands a more promising clique (i.e., clique that is likely to expand into larger cliques) before others.
- *Pruning*: the user can enable pruning by instrumenting $dominated(s, s')$ to return `true` if $|V_s| + |P_s|$ (the maximum possible size of the cliques into which s can expand) is smaller than $|V_{s'}|$ (the size of clique s').[1]

Fig. 4 illustrates how our framework can efficiently find the maximum clique (s_9 in Fig. 2) given the above custom functions and the data graph in Fig. 1a. Our framework first creates unit cliques s_1, s_2, s_3, and s_4 while assigning priorities to them (1). It then dequeues s_1 (i.e., the clique with the highest priority), adds s_1 to the result set, expands s_1 into s_5 and s_6, and then enqueues s_5 and s_6 (2). Next, it dequeues s_5, adds s_5 to the result set, removes s_1 from the result set, and expands s_5 into s_9 (3). Then, it dequeues s_9, and replaces s_5 in the result set with s_9 (4). At this point, s_2, s_3, s_4, and s_6 can be pruned out since they cannot expand into cliques as large as s_9.

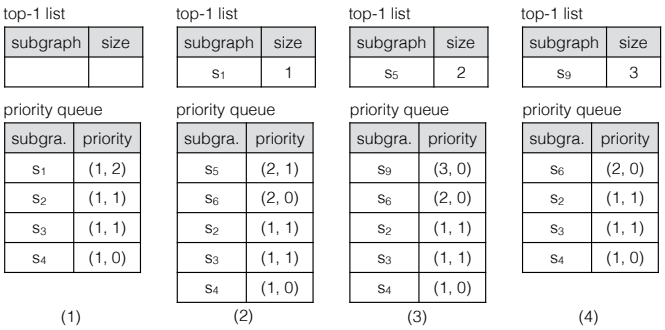


Fig. 4: Maximum Clique Discovery

Discussion. To the best of our knowledge, our subgraph discovery framework is the first one that supports *prioritized subgraph expansion*, an ability to expand more *promising* subgraphs (i.e., subgraphs whose expansions are more likely to quickly fill the result set with high-priority subgraphs) before others, thereby facilitating *early* and *effective* pruning. The benefits of our framework over the prior subgraph discovery systems [16], [17], [18] are evident in Figs. 2 and 4. For example, Arabesque [16] must expand all smaller subgraphs (e.g., all subgraphs of size 1) before any larger one (e.g., s_5), inherently limiting pruning opportunities (as opposed to ours that can expand s_5 before s_2, s_3, s_4, and s_6 and then prune out the latter subgraphs). Also, in contrast to ours that performs *targeted expansion*, Arabesque has to create all subgraphs (*exhaustive expansion*) and then filter out irrelevant ones such as non-clique subgraphs s_{10}, s_{11} and s_{12} (*post-filtering*).

B. Aggregate Computation

Some subgraph discovery computations require *grouping* subgraphs according to a certain feature (e.g., pattern) and then obtaining an *aggregate* value (e.g., frequency) from all of the subgraphs within each group. Our aggregate computational model for these computations have the following differences compared to the basic framework (Algorithm 1):

1) A new user-specified function, $key(s)$, returns the *grouping key* (e.g., pattern) of subgraph s. Our computational model associates each grouping key with the group of subgraphs having that grouping key.
2) Prioritization, insertion into the result set, and pruning of *subgraph groups* are specified by functions $priority(S)$, $relevant(S)$, and $dominated(S, S')$ where each of S and S' is a *subgraph group* (i.e., a group of subgraphs having the same key) as opposed to a *subgraph* in the basic (non-aggregate) framework. As in the basic framework, $expandable(s, \delta)$ is applied to a subgraph s and a vertex (or edge) δ.

Example: Top-*k* Frequent Pattern Mining. Consider the problem of finding the most frequent 2-edge patterns. A user

[1]This pruning condition was first introduced by Carraghan et al. [11]. The original work by Carraghan et al., however, does not specify any criterion for prioritizing cliques.

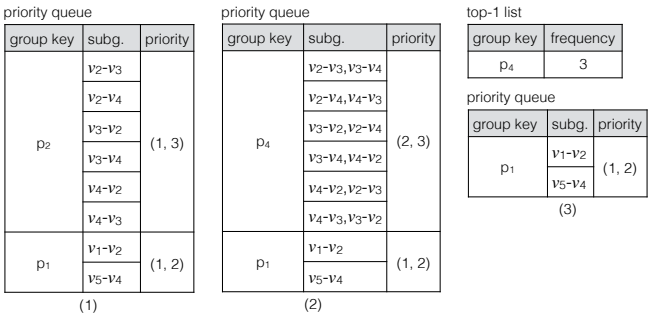


Fig. 5: Most Frequent Pattern Mining

can efficiently solve this problem by implementing custom functions as follows:

- *Subgraph expansion*: to obtain all subgraphs consisting of up to two edges, the $expandable(s, \delta)$ function needs to return `true` if s contains less than two edges. To include 2-edge patterns in the result set, the $relevant(S)$ function needs to return `true` if the grouping key (i.e., the pattern) of S has two edges.
- *Prioritized expansion*: the user can implement $priority(S)$ so that it returns $(m(S), f(S))$ where $m(S)$ denotes the number of edges in the pattern associated with subgraph group S and $f(S)$ denotes the frequency of that pattern. If lexicographic ordering is applied to such priority values, our framework processes larger patterns before smaller patterns, and, for patterns of the same size, processes more promising (i.e., frequent) patterns before others.
- *Pruning*: the user can enable pruning by (i) using a pattern frequency metric with anti-monotonicity (e.g., *minimum image-based support* [19]) which guarantees that any super-pattern p' of p cannot have a higher frequency than p (i.e., $f(p') \leq f(p)$), and (ii) by making $dominated(S, S')$ return `true` if $f(S) < f(S')$. When S' is the k-th entry in the result set and $dominated(S, S')$ returns `true`, all subgraphs in S can be safely ignored since expansions of them cannot affect the result set (due to anti-monotonicity, all subgraph patterns obtained through these expansions would have a lower frequency than the pattern associated with S').

Fig. 5 illustrates how our framework can efficiently find the most frequent pattern (p_4) given the above custom functions and the graph in Fig. 1b. In contrast to the examples shown in Fig. 3 where each subgraph expansion adds an edge to a subgraph (*edge-oriented expansion*) and Fig. 2 where each subgraph expansion adds, to a subgraph, a vertex and its edges connected to a vertex in that subgraph (*vertex-oriented expansion*), this example uses a different subgraph expansion approach which we call *pattern-oriented expansion*. The *pattern-oriented expansion* approach expresses each subgraph as a series of directed edges and expresses each subgraph pattern using the DFS code [24]. It constructs a subgraph s only if the DFS code of its pattern is *minimal* [24], which provides a guarantee that all of the subgraphs matching a pattern (e.g., p_4) can be obtained by choosing *only one sub-pattern* (e.g., p_2) and then expanding the subgraphs matching that sub-pattern. Due to space limitations, we refer the reader to our extended version of this paper [25] for further details of pattern-oriented expansion and our aggregate computational model.

In Fig. 5, our framework first forms two subgraph groups from 8 one-edge subgraphs (1). Next, it selects the group of subgraphs matching pattern p_2 (i.e., the group with the highest priority), expands the subgraphs in that group, and obtains a new group of subgraphs matching pattern p_4 (2). Since p_4 has two edges (i.e., matches the user's interest), it includes the subgraph group for pattern p_4 in the result set (3). It then discards the subgraphs matching pattern p_1 since $f(p_1) < f(p_4)$ (i.e., the patterns that can be obtained by expanding these subgraphs are less frequent than p_4).

Discussion. To the best of our knowledge, our work above is the first top-k aggregate subgraph discovery framework which supports both *prioritization* and *pruning*. Among the existing subgraph discovery systems [16], [17], [18], NScale [17] and G-Miner [18] do not support aggregate computations. Arabesque [16] expands *all smaller* subgraphs before any larger subgraphs (Fig. 3), thereby inevitably limiting prioritization and pruning opportunities. In contrast, due to its use of *pattern-oriented expansion*, our framework can perform *early* and *effective* pruning (e.g., no expansion of subgraphs matching p_1 and no creation of subgraphs matching p_3).

IV. API

We introduce our basic API (Sec. IV-A) and its extension for aggregate computation (Sec. IV-B) and indexing (Sec. IV-C).

A. API for Non-Aggregate Computation

For a non-aggregate subgraph discovery computation (Sec. III-A), users need to choose a subgraph expansion approach between vertex-oriented expansion (Fig. 2) and edge-oriented expansion (Fig. 3). Then, they need to create a new class, in Java, that extends either the `VertexOrientedSubgraph` or `EdgeOrientedSubgraph` type according to the chosen expansion approach. In that class, implementing *only* the `relevant()` method which corresponds to the $relevant(s)$ function in Table I enables the most preliminary form of subgraph discovery (without result ranking, pruning, and prioritization). The user can implement `expandable(Vertex v)` or `expandable(Edge e)` for targeted expansion, `priority()` for result ranking and prioritization, and `dominated(S o)` for pruning, where `S` is a Java generic type referring to the class being created. These methods correspond to the $expandable(s, \delta)$, $priority(s)$, and $dominated(s, s')$ functions in Table I.

Example: Maximum Clique Discovery. Listing 1 shows the implementation of a method that returns the set of vertices that can be added to a clique while forming a larger clique [11]. For the first vertex in the current clique (line 7), it adds to a set `p` the neighbors of that vertex (i.e., the vertices that can be added to the clique while forming a 2-vertex clique) (line 8). For every vertex `v` in the current clique except the first vertex, `p` needs to exclude `v` (line 10; since `v` is already in that clique) and retain only the vertices connected to `v` (line 11; i.e., vertices that can belong to a clique containing `v`). As explained in Sec. III-A, the maximum clique discovery computation

Listing 1: Clique Discovery (SubgraphCD)

```
public HashSet<Vertex> p = null;

HashSet<Vertex> p() {
  if (p == null) {
    p = new HashSet<Vertex>();
    for (Vertex v : vertices())
      if (v.equals(seed()))
        p.addAll(neighbors(v));
      else {
        p.remove(v);
        p.retainAll(neighbors(v));
      }
  }
  return p;
}
```

Listing 2: Subgraph Isomorphism Search (VertexIS)

```
public void initialize() {
  for (int h = 1; h <= D; h++)
    for (Vertex n : neighbors(h)) {
      Integer i = (Integer) get(n.label(), h);
      put(n.label(), h, i == null ? n.degree() :
        Math.max(i, n.degree()));
    }
}
```

can be implemented as follows: (i) `expandable(Vertex v)` returns `p().contains(v)`; (ii) `relevant()` returns `true`; (iii) `dominated(S o)` returns `(vertexCount() + p().size() < o.vertexCount())`; (iv) `priority()` returns `new double[] {vertexCount(), p().size()}`.

B. API for Aggregate Computation

An aggregate subgraph discovery computation (Sec. III-B) requires creation of two classes, one extending the `SubgraphGroup` type which contains the `key(Subgraphs)`, `relevant()`, `dominated(S o)`, `priority()` methods (corresponding to the custom functions $key(s)$, $relevant(S)$, $dominated(S, S')$, and $priority(S)$ in Sec. III-B) and another class for representing subgraphs. To enable pattern-oriented expansion (Sec. III-B), the latter class must extend the `PatternOrientedSubgraph` type which includes the `expandable(Edge e)` method. Due to space limitations, we refer the reader to our extended version of this paper [25] for an example implementation of top-k frequent pattern mining.

C. API for Indexing

Indexing techniques for subgraph discovery typically add index entries for each vertex [7], [26]. To support such techniques for Nuri, users need to create a class extending the `Vertex` type and implement the `initialize()` method which is invoked automatically for every vertex when the data graph is loaded into the system. To create and read index entries for each vertex, our API supports the two methods: (i) `put(k1, k2, ···, kn, v)` which associates a key comprising `k1`, `k2`, ···, and `kn` with a value `v` as an attribute of the vertex and (ii) `get(k1, k2, ···, kn)` which returns the value associated with the key comprising `k1`, `k2`, ···, and `kn`.

Example: Top-k Subgraph Isomorphism. Top-k subgraph isomorphism discovery [7] aims to find, in the data graph, the k highest-scored subgraphs which are isomorphic to a query graph. In this example, we define the score of each subgraph as the sum of the degree (e.g., the number of citations of each article) of the vertices in that subgraph. To facilitate pruning, we create an index entry for every vertex in the data graph in a way similar to the work by Gupta et al. [7]. The index stores, for each hop count h such that $h \leq D$ (where D is the maximum diameter of the query graphs to support) and label l, the maximum degree over all vertices that have label l and are h-hop away from the vertex under consideration (Listing 2).

If a subgraph s is obtained by repeatedly expanding a subgraph containing a seed vertex α, it is possible to derive an upper bound on the scores of the subgraphs that subgraph s can expand into. This upper bound is defined as the sum of (1) the current score of subgraph s and (2) an upper bound on the score that can be further added (i.e., $\sum_{v \in \mathcal{M}_s} index(\alpha, label(v), hop(v))$, where $\mathcal{M}_s$ denotes the vertices in the query graph that are not yet matched to any vertex in subgraph s, $hop(v)$ denotes the distance of v in the query graph from the vertex that corresponds to vertex α, $label(v)$ denotes the label of vertex v, and $index(\alpha, l, h)$ denotes the value in the index entry for vertex α, label l, and hop count h. Pruning and prioritization can then be supported by making `dominated(S o)` and `priority()` return `(score() + u() < o.score())` and `new double[] {edgeCount(), score() + u()}`, where `score()` and `u()` return the score and the score upper bound of the current subgraph, respectively. Due to space limitations, we refer the reader to our extended version of this paper [25] for the details of our top-k subgraph isomorphism implementation.

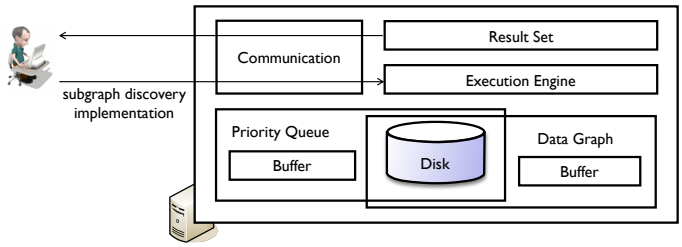
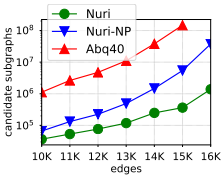


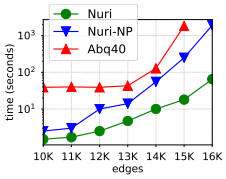
Fig. 6: System Architecture

V. System Architecture

Fig. 6 illustrates the architecture of our system. When a user submits an implementation of subgraph discovery, the *execution engine* carries out the computation according to an appropriate computational model (Sec. III) while inserting the subgraphs (or subgraph groups) matching the user's interest into the *result set*, whose updates are notified to the user. The *communication* component enables such interactions between the user and the system. For each subgraph discovery computation, the execution engine loads the *data graph* from the disk into the main memory. The number of subgraphs kept in the priority queue may increase significantly with the size and density of the data graph. For this reason, we implemented an *external priority queue* that can store a large number of entries on disk without being limited by the size of the main memory [27, Chapter 6]. This implementation manages high-priority subgraphs in memory and low-priory subgraphs on disk in a highly efficient manner where the use of disk causes only a slight degradation in the speed of enqueue/dequeue operations (Sec. VI-F).

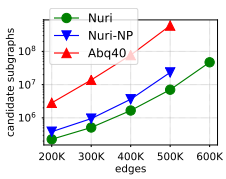


(a) subgraphs

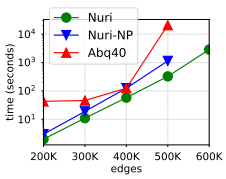


(b) time

Fig. 7: Clique Discovery (`Email`)

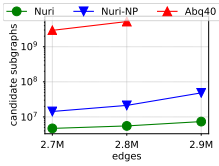


(a) subgraphs

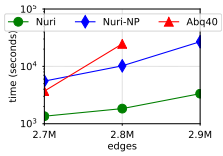


(b) time

Fig. 8: Clique Discovery (`MiCo`)



(a) subgraphs



(b) time

Fig. 9: Clique Discovery (`YouTube`)

TABLE II: Datasets

	$\|V\|$	$\|E\|$	distinct labels
`Email` [28]	986	16K	-
`CiteSeer` [29]	3.3K	4.5K	6
`MiCo` [29]	100K	1.1M	29
`YouTube` [5]	1.1M	2.9M	-
`Patents` [30]	2.7M	14M	37

VI. Evaluation

In this section, we explain our setup (Sec. VI-A) for evaluating the effectiveness of Nuri in comparison to alternative systems for clique discovery (CD), pattern mining (PM), and subgraph isomorphism (SI) computations (Sec. VI-B, VI-C, and VI-D). We also discuss the impact of the result set size (k) on subgraph discovery computations (Sec. VI-E) as well as the space overhead of Nuri and the performance of our external priority queue implementation (Sec. VI-F).

A. Experimental setup

Datasets. We employ five graph datasets from diverse domains and at different scales as shown in Table II. Vertices in the `Email` dataset represent people and each edge between two vertices indicates that at least one email message was sent between the people corresponding to the vertices. The `CiteSeer` dataset represents a citation network in which each publication is labeled by its research area. The `MiCo` dataset expresses a co-authorship network where authors are labeled by their research interests and a pair of authors is connected if they have co-authored at least one paper. `YouTube` represents a social network among users of the service and `Patents` represents a citation network among US patents between 1963 to 1999, where each vertex is labeled by the year the patent was granted.

Systems Compared. We compare two versions of our system, `Nuri` (supporting targeted expansion, pruning, and prioritization) and `Nuri-NP` (supporting only targeted expansion), and Arabesque [16] as a representative of the prior subgraph discovery systems [16], [17], [18]. In contrast to NScale [17] and G-Miner [18], Arabesque supports aggregate subgraph discovery (e.g., pattern mining) and its implementation is openly available. Both `Nuri` and `Nuri-NP` are implemented as single-threaded programs using the standard Java 8 distribution. In our experiments, each of these versions was run on a *single core* at 1.90GHz of an Intel(R) Xeon(R) E5-4640 server ($256GB$ memory) running Red Hat Enterprise Linux Server release 7.5. In contrast, Arabesque was executed on a *total of* 40 *cores* from 5 Intel(R) Xeon(R) E5430 servers (each with 8 cores at 2.66GHz and $16GB$ memory), benefiting from its distributed computing capability. In this section, the results obtained from Arabesque are labeled "`Abq40`".

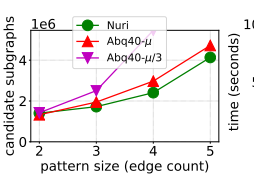
Evaluation Metrics. In terms of *convenience*, Nuri has a clear advantage over *custom* subgraph discovery solutions [31], [26], [32], [9], [7], [11]. For example, Nuri requires only tens of lines of user-provided code to perform CD, PM, and IS computations (Sec. IV). On the other hand, to create custom solutions for these computations, programmers would have to write at least thousands of lines of code for creating and expanding subgraphs, managing subgraphs on disk if they cannot be kept in memory, and, in the case of PM, grouping and aggregating subgraphs. To compare Nuri and Arabesque from the *performance* point of view, we use the following metrics: (1) *number of candidate subgraphs*: Both Arabesque and Nuri examine candidate subgraphs created through expansion until the desired result is obtained. Hence, we consider the number of candidate subgraphs as the basic cost metric which represents the inherent computational load of each system without being affected by the differences in the implementation of the systems. (2) *completion time of subgraph discovery*: This metric allows us to compare different systems/techniques from the user's point of view. It also compensates for the limitation of the first metric, which cannot incorporate the cost of optimization (e.g., time spent for prioritizing subgraphs and evaluating pruning conditions) and Arabesque's ability to speed up subgraph discovery through distributed computing.

B. Clique Discovery Evaluation

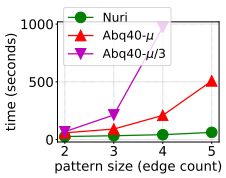
To obtain clique discovery (CD) results from Arabesque, we instrumented it to find all cliques and then select the largest clique(s) among them[2]. We created increasingly denser data graphs using the `Email`, `MiCo`, and `Youtube` datasets by repeatedly adding batches of randomly chosen edges to an empty graph. Denser graphs tend to include more and larger cliques, increasing the complexity of clique discovery.

Fig. 7 shows the CD results obtained for the `Email` dataset. As expected, both the number of candidate subgraphs and completion time increase with the density of the data graph for all of the systems. In the figure, the difference between `Nuri` and `Nuri-NP` demonstrates the benefits of *pruning* and *prioritization* (for 16K edges, `Nuri` examines only $1/26$ of subgraphs and is 29x faster compared to `Nuri-NP`). The gap between `Nuri-NP` and `Abq40` is due to *targeted expansion*, which allows Nuri to explore only relevant subgraphs (cliques) in contrast to Arabesque. Benefiting from targeted expansion and pruning/prioritization, for 15K edges, Nuri can find the largest clique(s) by examining only a small fraction ($1/400$) of

[2]The original implementation finds cliques of a predefined size.

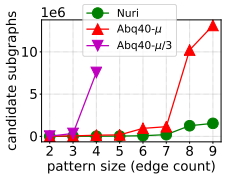


(a) subgraphs

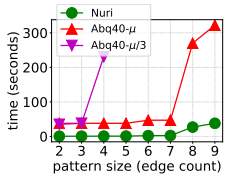


(b) time

Fig. 10: Pattern Mining (Patents)

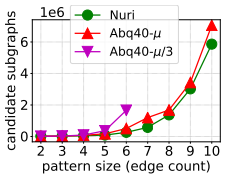


(a) subgraphs

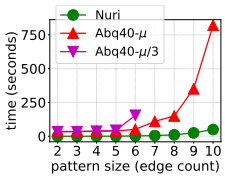


(b) time

Fig. 11: Pattern Mining (CiteSeer)



(a) subgraphs



(b) time

Fig. 12: Pattern Mining (MiCo)

subgraphs compared to Arabesque (Nuri vs. Abq40), resulting in 2 orders of magnitude improvement in completion time although Nuri uses much less computing resources (1 core vs. 40 cores). For 16K edges, Nuri completes its computation within 2 minutes while Arabesque cannot within 10 hours.

Figs. 8 and 9 show the CD results obtained for the MiCo and YouTube datasets. Given 500K edges from the MiCo dataset, Nuri examines only $1/85$ of subgraphs compared to Arabesque (Fig. 8a), resulting in a 65x improvement in completion time (Fig. 8b). On 600K edges from the MiCo dataset, Arabesque does not finish its computation within 10 hours whereas Nuri (with pruning and prioritization) completes within 47 minutes. Given 2.8M edges from the YouTube dataset, compared to Arabesque, Nuri examines 2 orders of magnitude fewer candidate subgraphs (Fig. 9a) and is 1 order of magnitude faster (Fig. 9b). Given 2.9M edges, Arabesque does not complete within 10 hours while Nuri finishes in less than 1 hour.

C. Pattern Mining Evaluation

Our pattern mining (PM) implementation (Sec. III-B) finds, given a number M, the most frequent patterns of size M (i.e., M-edge patterns) in the data graph. To obtain the same result, we instrumented Arabesque to (1) find all of the M-edge patterns whose frequency is no lower than a threshold T [16] and then (2) select the most frequent one(s) among these patterns. In real-world use cases, however, it is difficult to appropriately set T for Arabesque since the maximum frequency (denoted μ) over patterns of size M is *not known in advance*. If T is assigned a value lower than μ, Arabesque examines subgraphs that are unnecessary for the purpose of finding the most frequent pattern(s). If T is greater than μ, every pattern of size M is ignored (not reported) since its frequency is lower than T.

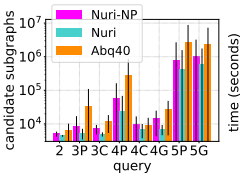
Fig. 10 shows the PM result obtained for the Patents dataset. When T is set to μ, Nuri and Arabesque explore a similar number of subgraphs (Nuri and Abq40-μ in Fig. 10a). The difference in the number of subgraphs between these systems is due to their use of different expansion approaches (Sec. III-B). In Fig. 10b, the gap between Nuri and Abq40-μ shows the benefits of pattern-oriented expansion (Nuri) over edge-oriented expansion (Arabesque). Under pattern-oriented expansion (Sec. III-B), when subgraph s expands into s', the pattern of s' can be quickly obtained by appending only one edge to the pattern of s (*incremental pattern generation*). On the other hand, under edge-oriented expansion, the pattern of each subgraph is always computed from scratch by converting that subgraph into its *canonical form* with high overhead [16]. When T is set to $\mu/3$, Arabesque examines subgraphs unnecessary for the purpose of finding the most frequent patterns, in contrast to Nuri which automatically prunes out such subgraphs. When M is 4 and T is $\mu/3$, Arabesque explores 2.5x more subgraphs than Nuri (Fig. 10a). The benefits of Nuri over Arabesque (particularly, *pruning* and *prioritization*) become more evident as the pattern size increases. In Figs. 11 and 12, similar trends can be seen for the CiteSeer and MiCo datasets, respectively. When T is set to $\mu/3$, Arabesque does not complete due to its high memory demand when the pattern size is 4 for CiteSeer and 6 for MiCo.

D. Subgraph Isomorphism Evaluation

Our top-k subgraph isomorphism (SI) implementation discovers, in the data graph, the k highest-scored subgraphs that are isomorphic to a given query graph, where the score of each subgraph is defined as the sum of the degree of the vertices in that subgraph (Sec. IV-C). This implementation adopts Ullman's algorithm [8] to efficiently find the subgraphs that match the query graph (*targeted expansion*). Since Arabesque does not support targeted expansion, our SI implementation for Arabesque exhaustively explores subgraphs while filtering out subgraphs that do not pass a subgraph isomorphism test (a user-provided implementation). We also extended Arabesque so that it can maintain the k highest-scored subgraphs.

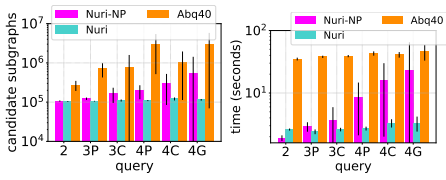
To conduct SI computations, we obtained query graphs of sizes from 2 to 5 by running a sampling algorithm [33] on each data graph constructed from the CiteSeer and MiCo datasets. For 4-vertex subgraphs (and 5-vertex subgraphs), we considered three different types, namely path, clique, and general that are labeled "4P", "4C", and "4G" (and "5P", "5C", and "5G"). We considered, for 3-vertex subgraphs, the path and clique types (labeled "3P" and "3C") and, for 2-vertex subgraphs, only one type (labeled "2") which corresponds to both the path and clique types. For each type of query graph, we ran 10 SI computations each with a different query graph and then calculated the mean values for the number of subgraphs and completion time. For the graph from CiteSeer, we did not consider cliques of size 5 since the graph is sparse and thus contains few cliques of size 5.

For pruning and prioritization, our SI implementation uses an index computed for every vertex in the data graph up to D-hops, where D is the maximum diameter of all query graphs. The index construction time for all 5-vertex query graphs (i.e., for 4-hops) was 1 second for the CiteSeer dataset. For the MiCo dataset, the index construction for all 4-vertex query graphs took 300 minutes using a single core. The index



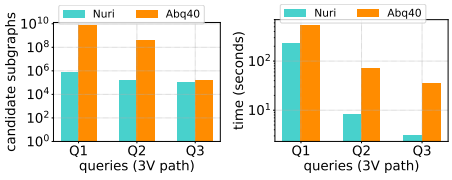
(a) subgraphs (b) time

Fig. 13: SI (CiteSeer)



(a) subgraphs (b) time

Fig. 14: SI (MiCo)

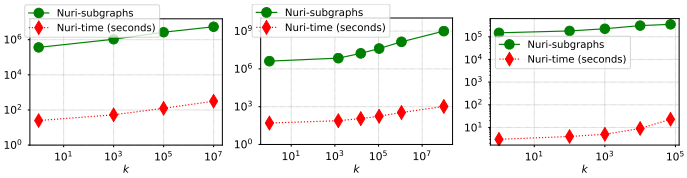


(a) subgraphs (b) time

Fig. 15: Impact of Selectivity

TABLE III: Maximum Memory Usage

computation	CD			PM			SI	
dataset	Email	MiCo	YouTube	Patents	CiteSeer	MiCo	CiteSeer	MiCo
maximum memory usage	3.7G	9.6G	12G	6.2G	4G	8G	6.4G	22G



(a) CD (Email) (b) PM (CiteSeer) (c) SI (CiteSeer)

Fig. 16: Effect of Result Set Size (k)

construction time was reduced to 600 seconds when the index entries for each vertex were created in parallel on 32 cores.

As evident in Figs. 13a and 14a, as the query graph size increases, both Arabesque and Nuri explore more subgraphs for each query type. In the CiteSeer graph, due to the sparsity of the graph, clique queries are very selective and usually explore a smaller number of subgraphs than path and general subgraph queries of the same size. In Figs. 13a and 14a, the difference between Nuri-NP and Abq40 demonstrates the benefits of targeted expansion. Figs. 13b and 14b show that Nuri is substantially faster than Arabesque taking advantage of pruning, prioritization, and targeted expansion despite its use of much less resources (1 core vs. 40 cores).

Fig. 15 illustrates how the selectivity of query graphs affects the SI computation for the CiteSeer dataset. In the figure, Q1 is a non-selective query for which several million matches exist in the data graph. Q2 is a mildly selective query and Q3 is a highly selective query with fewer than 400 matches in the data graph. Fig. 15a shows that, due to prioritization and pruning, Nuri explores much fewer subgraphs than Arabesque (e.g., 4 orders of magnitude fewer subgraphs for Q1) as the query graph has more matches in the data graph. For this reason, Nuri on a single core runs much faster than Arabesque on 40 cores (Fig. 15b). In Fig. 15b, Arabesque's time costs for Q2 and Q3 are mostly caused by its distributed operation (particularly, coordination of multiple workers) rather than actual exploration of subgraphs.

E. Effect of the Result Set Size (k)

The previous evaluations focused on top-1 computations. As the result set size (k) increases, Nuri tends to explore more subgraphs since it can start pruning out subgraphs only after its result set contains k entries. We measured the effect of k on the number of candidate subgraphs and completion time. Fig. 16 shows our results obtained from clique discovery on the Email dataset and pattern mining and subgraph isomorphism computations on the CiteSeer dataset. In the figure, as long as k is smaller than 1000, the number of candidate subgraphs and completion time vary insignificantly. When k is greater than 1000, the number of candidate subgraphs and completion time increase modestly with k.

F. Space Overhead and External Priority Queue

Table III shows the memory usage results from our experiments when all subgraphs are kept in memory. When subgraphs no longer fit into the memory, our external priority queue implementation can efficiently manage them on disk (Sec. V). In our experiments, despite disk usage, its overall enqueue/dequeue time costs were only at most 1.8 times higher compared to the standard Java PriorityQueue implementation which was given a much larger memory space and managed all of its elements in that space. For further details, we refer the reader to an extended version of this paper [25].

VII. Related Work

Existing subgraph discovery systems are discussed in Sec. II-B. This section summarizes additional related work.

Maximum Clique Discovery. Our maximum clique discovery implementation (Sec. III-A and IV-A) is based on the CP algorithm [11] which calculates, for each clique s, an upper bound on the size of the cliques that s can expand into and then prunes out s if its size upper bound is smaller than the size of largest clique(s) discovered. Researchers have also developed algorithms that can outperform CP by finding a tighter upper bound [13], [14]. We leave the implementation of these algorithms for Nuri as our future work.

Top-k Pattern Mining. Given a graph, our top-k pattern mining implementation finds the k *most frequent* patterns of a certain size (Sec. III-B and IV-B). The closest work that we are aware of [10] instead finds the k *largest* patterns that are as frequent as a given threshold. An earlier work called GRAMI [9] seeks, in contrast to our implementation, all patterns whose frequency is no less than a threshold. Han et al. also developed a threshold-based frequent pattern mining solution for a different frequency metric [34].

Top-k Subgraph Isomorphism. Our top-k subgraph isomorphism implementation is motivated by Gupta et al.'s work [7]

which uses an index to quickly identify subgraphs whose expansions cannot produce any of the desired subgraphs. Zou et al. developed another indexing solution [26]. We intend to implement and evaluate this solution for Nuri.

Custom Top-k Subgraph Discovery. There are also subgraph discovery techniques that are designed to quickly obtain the k subgraphs of highest preference according to a user-specified criterion [31], [32], [7], [35], [36], [37]. Our future work includes implementation of these techniques for Nuri.

Top-k Query Processing. In the context of database systems, various techniques for top-k queries have been developed [38]. These techniques cannot adequately support subgraph discovery computations since they are mainly designed for queries on relations rather than a large collection of subgraphs that need to be expanded according to their priorities.

VIII. Conclusions

We presented Nuri, a new system for efficient top-k subgraph discovery in large graphs. Nuri's API allows users to specify application-specific criteria for exploring and prioritizing subgraphs. Nuri also proactively discards subgraphs from which the desired subgraphs cannot be obtained (pruning). For discovery computations with high space overhead, it provides efficient on-disk management of subgraphs. We evaluated Nuri on real-world datasets of various sizes for three example computations: maximum clique discovery, subgraph isomorphism search, and pattern mining. Nuri consistently outperformed the closest state-of-the-art alternative, achieving at least 2 orders of magnitude improvement for clique discovery and 1 order of magnitude improvement for subgraph isomorphism search and pattern mining, while utilizing $1/40$ of the computational resources compared to the closest alternative.

Acknowledgements

This work is in part supported by NSF under CAREER award IIS-1149372 and NSF SCC-1831547.

References

[1] M. Chau and J. Xu, "Mining Communities and Their Relationships in Blogs: A Study of Online Hate Groups," *International Journal of Human-Computer Studies*, vol. 65, no. 1, pp. 57–70, 2007.

[2] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou, "Mining Coherent Dense Subgraphs across Massive Biological Networks for Functional Discovery," in *ISMB*, 2005, pp. 213–221.

[3] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov, "BotGrep: Finding P2P Bots with Structured Graph Analysis," in *USENIX Security Symposium*, 2010, pp. 95–110.

[4] W. Eberle, J. Graves, and L. Holder, "Insider Threat Detection Using a Graph-Based Approach," *Journal of Applied Security Research*, vol. 6, no. 1, pp. 32–81, 2010.

[5] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities based on Ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[6] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo$_{\text{iso}}$: Towards UltraFast and Robust Subgraph Isomorphism Search in Large Graph Databases," *PVLDB*, pp. 517–528, 2014.

[7] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han, "Top-K Interesting Subgraph Discovery in Information Networks," in *ICDE*, 2014, pp. 820–831.

[8] J. R. Ullmann, "An Algorithm for Subgraph Isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[9] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph," *PVLDB*, pp. 517–528, 2014.

[10] Z. Feida, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu., "Mining Top-K Large Structural Patterns in a Massive Network," *PVLDB*, vol. 4, no. 11, pp. 807–818, 2011.

[11] R. Carraghan and P. M. Pardalos, "An Exact Algorithm for the Maximum Clique Problem," *Operations Research Letters*, vol. 9, pp. 375–382, 1990.

[12] Q. Wu and J. Hao, "A Review on Algorithms for Maximum Clique Problems," *European Journal of Operational Research*, vol. 242, no. 3, pp. 693–709, 2015.

[13] C. Bron and J. Kerbosch, "Finding All Cliques of an Undirected Graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[14] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast Algorithms for Maximal Clique Enumeration with Limited Memory," in *SIGKDD*, 2012, pp. 1240–1248.

[15] R. Andersen, "A Local Algorithm for Finding Dense Subgraphs," in *SODA*, 2008, pp. 1003–1009.

[16] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A System for Distributed Graph Mining," in *SOSP*, 2015, pp. 425–440.

[17] A. Quamar, A. Deshpande, and J. Lin, "NScale: Neighborhood-centric Analytics on Large Graphs," *PVLDB*, vol. 7, no. 13, pp. 1673–1676, 2014.

[18] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-Miner: An Efficient Task-Oriented Graph Mining System," in *EuroSys*, 2018, pp. 32:1–32:12.

[19] B. Bringmann and S. Nijssen, "What is Frequent in a Single Graph?" in *PAKDD*, 2008, pp. 858–863.

[20] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *SIGMOD*, 2010, pp. 135–146.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Framework For Parallel Machine Learning," in *UAI*, 2010, pp. 340–349.

[22] W. Han, S. Lee, K. Park, J. Lee, M. Kim, J. Kim, and H. Yu, "TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC," in *SIGKDD*, 2013, pp. 77–85.

[23] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks and ISDN Systems*, vol. 30, pp. 107–117, 1998.

[24] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," in *ICDM*, 2002, pp. 721–724.

[25] A. Joshi, Y. Zhang, P. Bogdanov, and J. Hwang, "An Efficient System for Subgraph Discovery," *CoRR*, vol. abs/1807.08888, 2018.

[26] L. Zou, L. Chen, and Y. Lu, "Top-K Subgraph Matching Query in A Large Graph," in *CIKM Ph. D. Workshop*, 2007, pp. 139–146.

[27] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[28] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph Evolution: Densification and Shrinking Diameters," *TKDD*, vol. 1, no. 1, p. 2, 2007.

[29] X. Yan, P. S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-based Approach*," in *SIGMOD*, 2004, pp. 335–346.

[30] B. H. Hall, A. B. Jaffe, and M. Trajtenberg, "The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools," http://www.nber.org/patents/, 2011.

[31] Y. Wu, S. Yang, and X. Yan, "Ontology-based Subgraph Querying," in *ICDE*, 2013, pp. 697–708.

[32] S. Yang, F. Han, Y. Wu, and X. Yan, "Fast Top-K Search in Knowledge Graphs," in *ICDE*, 2016, pp. 990–1001.

[33] R.-H. Li, J. X. Yu, L. Qin, R. Mao, and T. Jin, "On Random Walk Based Graph Sampling," in *ICDE*, 2015, pp. 927–938.

[34] J. Han and J.-R. Wen, "Mining Frequent Neighborhood Patterns in a Large Labeled Graph," in *CIKM*, 2013, pp. 259–268.

[35] P. Bogdanov, B. Baumer, P. Basu, A. Bar-Noy, and A. K. Singh, "As Strong as the Weakest Link: Mining Diverse Cliques in Weighted Graphs," in *ECML/PKDD (1)*, 2013, pp. 525–540.

[36] P. Bogdanov, M. Mongiovì, and A. K. Singh, "Mining Heavy Subgraphs in Time-Evolving Networks," in *ICDM*, 2011, pp. 81–90.

[37] D. J. DiTursi, G. Ghosh, and P. Bogdanov, "Local Community Detection in Dynamic Networks," in *ICDM*, 2017, pp. 847–852.

[38] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A Survey of Top-*k* Query Processing Techniques in Relational Database Systems," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 11:1–11:58, 2008.