A Fast Algorithm for Subgraph Search Problem

Karam Gouda
Faculty of Computers & Informatics
Benha University, Benha, Egypt
karam.gouda@fci.bu.edu.eg

Mosab Hassaan
Faculty of Computers & Informatics
Benha University, Benha, Egypt
mosab.hassaan@fci.bu.edu.eg

Abstract

Graphs are widely used to model complicated data semantics in many applications. In this paper we propose `Fast-ON`*, an efficient algorithm for subgraph isomorphism problem which has proven to be NP-complete.* `Fast-ON` *is based on Ullman algorithm [8]. It improves the search space of Ullman algorithm by considering two effective optimizations. Comparing to the well-known algorithms Ullman and Vflib [3],* `Fast-ON` *achieves up to 1-3 orders of magnitude speed-up.*

1. Introduction

As a popular data structure, graphs have been used to model many complex data objects and their relationships in the real world, such as the chemical compounds [9], entities in images [7], and social networks [1]. For example, in social network, a person i corresponds to a vertex v_i in the graph G, and another person j corresponds to a vertex v_j in the graph G. If persons i and j are acquaintances or they have a business relation, then an edge (v_i, v_j) exists, which connects vertex v_i and v_j. Also in chemistry, a set of atoms combined with designated bonds are used to describe chemical molecules. Due to the wide usage of graphs, it is quite important to retrieve data graphs containing a query graph from graph database efficiently. For example, given a large chemical compound database, a chemist may want to find all chemical compounds having a particular substructure. This type of search is well-known as subgraph search. Formally, given a graph database $\mathcal{D}$ and a query graph q, we need to find all data graphs $g_i \in \mathcal{D}$, where g_i contains the query q, namely, q is subgraph isomorphic to g_i.

Figure 1 shows a running example of subgraph isomorphism problem, where a query graph q and a data graph G are listed. The letter beside the vertex is its id and the letter inside the vertex is its label, and the letter through edge is the edge label. Graph G should be returned as the result, since graph G contains query q.

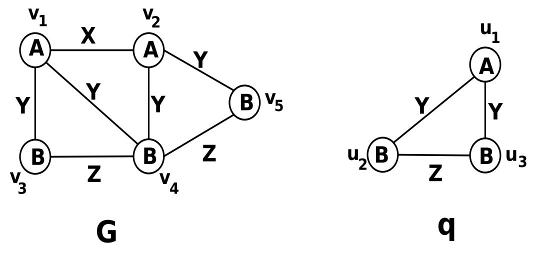

Figure 1. Running Example

Unfortunately, the subgraph search problem is hard in that it requires subgraph isomorphism checking of query q against each data graph g_i, which has proven to be NP-complete problem [4]. Indexing [10, 2, 11] is proposed to alleviate the overhead of pairwise isomorphism checks. In this approach, Indexes are used to quickly filter out data graphs that are not possible in the result and produce candidate graphs. Then the candidate graphs are verified, i.e. whether the query graph is a subgraph of each candidate, by a subgraph isomorphism algorithm. The efficiency of this approach depends on the filtering power of each indexing methodology and how fast it produces candidate graphs. Even with this approach, efficient subgraph checking algorithm is very important since it is required to verify the candidates. Note that there are many scenarios in which all data graphs, or most of them, contain the query, and using any filtering process would return all these graphs as candidates to be finally verified.

Related Work. Ullman [8] and Vflib [3] are two well-known algorithms for subgraph isomorphism problem. Ullman algorithm is developed based on the branch and bound paradigm [6]. It is prohibitively expensive for querying against a very large data graph. The Vflib algorithm is another important algorithm for subgraph isomorphism problem. It uses an optimized serial version of Ullman algorithm. The algorithm proceeds by creating and modifying a match state. The match state contains a matched-set, which is a set of vertex pairs that match between the query graph q and data graph G. If the matched-set contains all of the query graph q, then the algorithm is successful and returns. Otherwise, the algorithm attempts to add a new pair. It does this by tracking the in-set and out-set of each graph, which are the sets of vertices immediately adjacent to the matched-set. These two sets define the potential vertices that can be added to a given state. The only pairs that can be added are either in the in-set of both graphs or the out-set of both graphs. The algorithm uses backtracking search to find either a successful match state, or return a failure.

Our Contributions. In this paper, we propose an efficient subgraph isomorphism testing algorithm. It is based on Ullman algorithm and reduces the search space as much as possible by following a novel ordering strategy of the query's vertices, and by utilizing the label information of vertex's neighborhood. The new algorithm is called `Fast-ON` (which stands for the bold letters in: **Fast** subgraph testing by **O**rdering the query's vertices and utilizing labeled **N**eighborhood information). Comparing to the well-known algorithms Ullman [8] and Vflib [3], `Fast-ON` achieves up to 1-3 orders of magnitude speed-up.

Organization. This paper is organized as follows. Section 2 defines the preliminary concepts. Section 3 presents Ullman algorithm in details. Section 4 presents our proposed algorithm `Fast-ON`. Section 5 reports the experimental results. Finally, Section 6 conclude the paper.

2. Preliminary Concepts

As a general data structure, labeled graph is used to model complex structured and schema-less data. In labeled graph, vertices and edges represent entity and relationship, respectively. The attributes associated with entities and relationships are called labels. This paper focuses on simple undirected graphs with vertex and edge labels. Below, the terminology used throughout the paper is introduced.

Definition 2.1 (Labeled Graph)
A labeled graph G is defined as a 4-tuples $<V_G, E_G, L_G, l_G>$, where V_G is the set of vertices, E_G is the set of edges, L_G is the set of labels, and l_G is a labeling function that maps each vertex or edge to a label in L_G.

Definition 2.2 (Vertex Neighborhood)
Given a graph G, the neighborhood of $u \in V_G$ is the set $N_G(u) = \{v \in V_G \mid (u, v) \in E_G\}$. The degree of a vertex $v \in V_G$ is defined as $deg(v) = |N_G(v)|$.

Definition 2.3 (Graph Isomorphism)
Given two graphs $G = <V_G, E_G, L_G, l_G>$ and $H = <V_H, E_H, L_H, l_H>$. A graph isomorphism from H to G is a bijection $f : V_H \longmapsto V_G$ such that: (1) for any edge $(u, v) \in E_H$, there is an edge $(f(u), f(v)) \in E_G$, (2) $l_H(u) = l_G(f(u))$ and $l_H(v) = l_G(f(v))$, and (3) $l_H((u, v)) = l_G((f(u), f(v)))$.

The concept of **subgraph isomorphism** can be defined analogously by using an *injection* instead of a *bijection*. A graph H is called a subgraph of another graph G (or G is a supergraph of H), denoted as $H \subseteq G$ (or $G \supseteq H$), if there exists a subgraph isomorphism from H to G.

3. Ullman Algorithm

One of the earliest and highly-cited approaches to the subgraph isomorphism problem is the algorithm proposed by Ullman. Given a query graph q and a data graph G. To check if q is subgraph of G, Ullman's basic approach is to enumerate all possible mappings of vertices in V_q to those in V_G using a depth-first tree-search algorithm. Figure 2 shows a part of the search tree generated from testing the two graphs in Figure 1. At level i of the search tree, a vertex u_i in V_q is mapped to some vertex in V_G (the number j inside each node in the search tree means that this node represents the vertex $v_j \in V_G$). The root node of the search tree represents the starting point of the search, inner nodes of the search tree correspond to partial mappings, and nodes at level $|V_q|$ represent complete – not necessarily sub-isomorphic – mappings. If there exists a complete mapping that preserves adjacency in both q and G, then we have q is subgraph isomorphic to G, otherwise q is not subgraph isomorphic to G. The bold path in Figure 2, (u_1 is mapped to v_1, u_2 is mapped to v_3, and u_3 is mapped to v_4), is a complete mapping that preserves adjacency in q and G, thus q is subgraphs isomorphic to G.

Unfortunately, the number of complete mappings is exponential in the number of nodes of the involved graphs. This means that the running time may be huge even for reasonably small graphs. In order to cope with subgraph isomorphism problem efficiently, Ullman proposed a refinement procedure to prune the search space. It is based on the following three conditions:

1. **Label and degree condition.** A vertex $u \in V_q$ can be mapped to $v \in V_G$ under injective mapping f, i.e $v = f(u)$, if
(i) $l_q(u) = l_G(v)$, and
(ii) $deg(u) \leq deg(v)$.
2. **One-to-One mapping of vertices condition.** Once vertex $u \in V_q$ is mapped to $v \in V_G$, we cannot map any other vertex in V_q to the vertex $v \in V_G$.

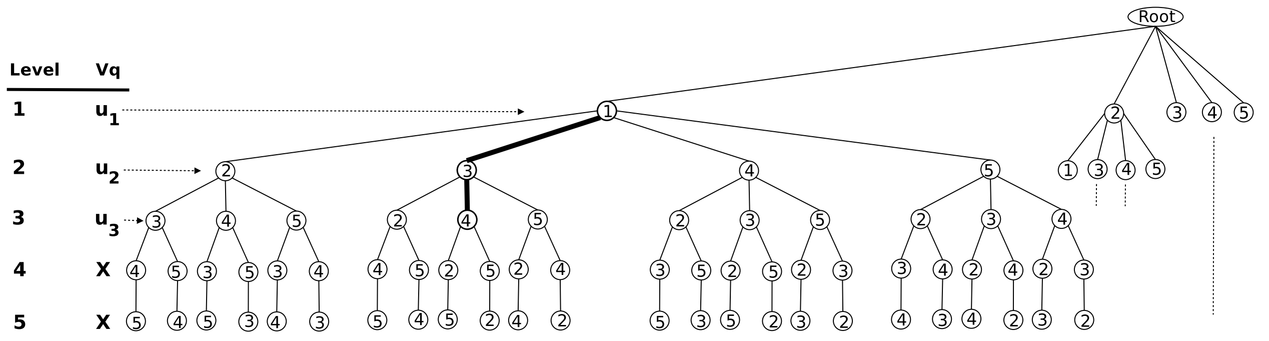

Figure 2. A part of search tree of Ullman algorithm

3. **Neighbor condition.** By this condition Ullman algorithm examines the feasibility of mapping $u \in V_q$ to $v \in V_G$ by considering the preservation of structural connectivity. If there exist edges connecting u with previously explored vertices of q but there are no counterpart edges in G, the mapping test simply fails.

4. `Fast-ON` Algorithm

In this section we propose `Fast-ON`, a new algorithm for subgraph isomorphism problem. `Fast-ON` is based on Ullman algorithm. The search space considered by Ullman algorithm is still huge even after using the refinement procedure. `Fast-ON` explores much smaller space than that of Ullman algorithm by using the following two new optimizations.

4.1 Opt1: Ordering the query vertices

Our first optimization is based on the observation that the search order in Ullman algorithm is random. It depends on the order of query vertices imposed during input. This default ordering of V_q can possibly result in a search order that seriously slows down Ullman Algorithm. Query vertices should be explored in the order that facilitates getting the utmost benefit of applying the third condition. Our approach to order V_q is to require the currently processing query vertex to have high connectivity with the previously explored ones, that is, suppose that $u_i \in V_q$ is the currently processing vertex, then u_i should have the higher connectivity with $u_1, u_2, \ldots, u_{i-1}$ among the remaining ones. Whereas, u_1 is the one with maximum degree. This new ordering forces false mapping to be discarded as early as possible during the search, thus saving much of the time that Ullman algorithm may take on false long partial mappings. Figure 3 outlines this idea.

Algorithm: $Order\_Vertices(V_q)$

```
Input: $V_q = \{u_1, u_2, \ldots, u_{|V_q|}\}$;
Output: An order of $V_q$, $V'_q = \{u'_1, u'_2, \ldots, u'_{|V_q|}\}$;
1.  $V'_q = \phi$;
2.  for each $u \in V_q$ do calculate $deg(u)$;
3.  $u'_1 = u_k$, $k$ = argmax$_{u \in V_q} deg(u)$;
4.  Add $u'_1$ to $V'_q$ and remove $u_k$ from $V_q$;
5.  for $i = 2 \ldots |V_q|$
6.      $u'_i = u_k$, $k$ = argmax$_{u \in V_q}$ $|\{(u,u') \in E_q : u' \in V'_q\}|$;
7.      Add $u'_i$ to $V'_q$ and remove $u_k$ from $V_q$;
8.  return $V'_q$;
```

Figure 3. Ordering Query Vertices Algorithm

4.2 Opt2: Utilizing Neighborhood Labels

Here, we introduce a novel condition effective in reducing the search space. It is based on the neighborhood labels of matching vertices. This new condition is much stronger than the label and degree condition of the refinement procedure. First, we define the labeled neighborhood of any vertex as follows.

Definition 4.1 (Vertex Labeled Neighborhood) *Given a graph G and a vertex $u \in V_G$, the labeled neighborhood of u is given as $NL_G(u) = \{(l_G(v), l_G((u,v))) : v \in V_G$ and $(u,v) \in E_G\}$.*

The following theorem presents the necessary condition required to map a vertex $u \in V_q$ to a vertex $v \in V_G$.

Theorem 4.1 *Given two graphs q and G such that q is subgraph isomorphic G under injective function f. If $u \in V_q$ is mapped to $v \in V_G$, then $NL_q(u) \subseteq NL_G(v)$*

Thus, according to Theorem 4.1, if the labeled neighborhood of vertex $v \in V_G$ does not contain the labeled neighborhood of vertex $u \in V_q$, u can not be mapped to v. We can reduce the search space by enforcing this inclusion test. Next condition generalizes the first condition of the refinement procedure by adding the new inclusion test.

1. **Label and neighborhood inclusion condition.** A vertex $u \in V_q$ can be mapped to $v \in V_G$ under injective function f, i.e $v = f(u)$, if
   (i) $l_q(u)$ = $l_G(v)$, and
   (ii) $NL_q(u) \subseteq NL_G(v)$.

Note that if $NL_q(u) \subseteq NL_G(v)$ is satisfied, it directly leads to $deg(u) \leq deg(v)$ since $deg(v) = |NL_G(v)|$.

Example 4.1 *Consider the two graphs q and G given in Figure 1. According to the label and neighborhood inclusion condition, we can map vertex $u_1 \in V_q$ to $v_1 \in V_G$ since (i) $l_q(u_1) = l_G(v_1) = A$, and (ii) $NL_q(u_1) = \{(B,Y),(B,Y)\} \subseteq \{(A,X),(B,Y),(B,Y)\} = NL_G(v_1)$.*

Though the Label and neighborhood inclusion condition is effective in reducing the search space, applying the inclusion test is expensive especially for large size graphs with higher average vertex degree. Below, we propose a new method to efficiently apply the inclusion test. The method is based the observation that many vertices in the query or data graph share the same neighborhood. The next example highlights this fact.

Example 4.2 *Consider the query graph q and data graph G given in Figure 1. We have (1) In graph G: $NL_G(v_1)$ = $NL_G(v_2)$ = $\{(A,X),(B,Y),(B,Y)\}$, $NL_G(v_3)$ = $NL_G(v_5)$ = $\{(A,Y),(B,Z)\}$, and $NL_G(v_4)$ = $\{(A,Y),(A,Y),(B,Z),(B,Z)\}$; (2) In query graph q: $NL_q(u_1)$ = $\{(B,Y),(B,Y)\}$, and $NL_q(u_2)$ = $NL_q(u_3)$ = $\{(A,Y),(B,Z)\}$.*

Based on the above observation, we can reduce the cost of the containment checks by cashing most of the repeated computation, as in the following steps:

1. Find the set of distinct labeled neighborhoods for the two graphs q and G, denoted as DLN_G and DLN_q, respectively.

2. Construct a bit matrix $M_{DLN} = (m_{ij})_{\alpha\beta}$ where $\alpha = |DLN_q|$ and $\beta = |DLN_G|$, to maintain the inclusion relationship between distinct neighborhoods of q and G, that is, $m_{ij} = 1$ if $DLN_q[i] \subseteq DLN_G[j]$, otherwise $m_{ij} = 0$.

3. For a graph g – g is q or G – construct an array of pointers P_g of size $|V_g|$, called position array, where each slot u holds the index of the vertex u labeled neighborhood at DLN_g.

```
Algorithm: $Fast-ON(q,G)$

Input: $q$: a query graph and $G$: a data graph.
Output: Boolean: $q$ is a subgraph of $G$.
Boolean Test = FALSE;                          /* Global Variable */
1.  $V'_q = Order\_Vertices(V_q)$;                    /* Opt1 */
2.  Construct $DLN_G, DLN_q$ and $M_{DLN}$;
3.  Construct both $P_q$ and $P_G$;
4.  for each $u \in V'_q$ do
5.      $C(u)$ = $\{v : v \in V_G, l_q(u) = l_G(v)$, and
                      $m_{P_q(u)P_G(v)} = 1\}$;   /* Opt2 */
6.  $Recursive\_Search(u_1)$;
7.  return Test;
Procedure $Recursive\_Search(u_i)$
1.  if NOT Test then
2.      for $v \in C(u_i)$ and $v$ is unmatched do    /* Cond. 2 */
3.          if NOT $Matchable(u_i, v)$ then continue;
4.          $f(u_i) = v$; $v$ = matched;
5.          if $i < |V'_q|$ then
6.              $Recursive\_Search(u_{i+1})$;
7.          else
8.              Test = TRUE;
9.              return;
10.         $f(u_i) =$ NULL; $v$ = unmatched;     /* Backtrack */
Function $Matchable(u_i, v)$                     /* Cond. 3 */
1.  for each $(u_i, u_j) \in E_q, j < i$ do
2.      if $(v, f(u_j)) \notin E_G$ then return FALSE;
3.  return TRUE;
```

Figure 4. `Fast-ON` **Algorithm**

Now we can say that, for each $u \in V_q$ and $v \in V_G$, we have $NL_q(u) \subseteq NL_G(v)$ iff $m_{P_q(u)P_G(v)} = 1$. Thus, the test (ii) in label and neighborhood inclusion condition can be replaced by testing if $m_{P_q(u)P_G(v)} = 1$.

In subgraph search problem, cashing the repeated computation as above is very useful since real graph datasets tend to share commonality, that is, a vertex may appear in many data graphs. This happens because the real data come from the same application domain. Note that in the experimental section, subgraph search is used for testing `Fast-ON` algorithm.

4.3 `Fast-ON` Algorithm

Figure 4 outlines `Fast-ON` algorithm. Line 1 applies the first optimization Opt1, whereas lines 2-5 outline the second optimization Opt2. In line 5, for each query vertex $u \in V_q$, data graph vertices $v \in V_G$ that satisfy the modified first condition are collected into a set called candidate set

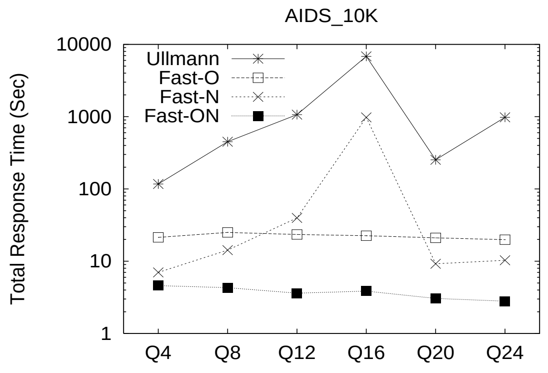

Figure 5. Effects of optimizations

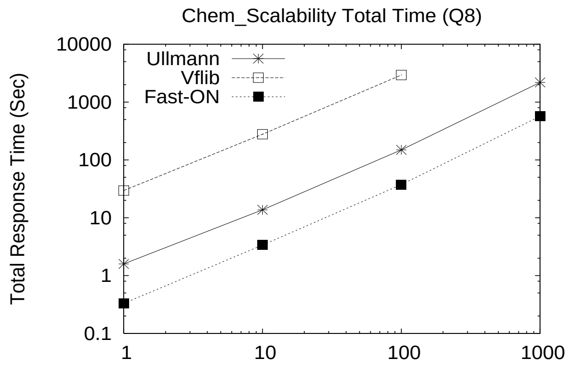

Figure 7. Scalability on dataset size (# Graphs in K)

$C(u)$. The procedure $Recursive\_Search$ matches u_i over $C(u_i)$ (line 5) and proceeds step-by-step by recursively matching the subsequent vertex u_{i+1} over $C(u_{i+1})$ (lines 6-7), or sets Test to true value and returns if every vertex of q has counterpart in G (line 9). If u_i exhausts all vertices in $C(u_i)$ and still cannot find matching, Recursive_Search backtracks to the previous state for further exploration (line 11). The procedure Matchable applies the third condition.

Note that according to Opt2, for each u, $C(u)$ is as small as possible. Consequently `Fast-ON` explores much smaller space than Ullman algorithm. Moreover, according to Opt1, false mappings are discarded as early as possible, saving much of the computation spent by Ullman algorithm.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of `Fast-ON` on real and synthetic graphs. `Fast-ON` is implemented in standard C++ with STL library support and compiled with GNU GCC. Experiments were run on a PC with Intel 3GHz dual Core CPU and 4G memory running Linux. In experiments, we consider vertex-labeled and edge-labeled simple graphs.

5.1 Datasets

Experimental evaluation are performed on a group of real and synthetic datasets as follows.

Real Datasets. The first real dataset, referred to as AIDS_10k, consists of 10,000 graphs that are randomly drawn from the AIDS Antiviral screen database [1]. These graphs have 25 vertices and 27 edges on average. There are totally 62 distinct vertex labels in the dataset but the majority of these labels are C, O and N. The total number of distinct edge labels is 3. In order to study the scalability of `Fast-ON` against different dataset size, we use a large real chemical compound dataset as a second real dataset, denoted as Chem_1M. Chem_1M is a subset of the PubChem database [2], and consists of one million graphs. Chem_1M has 23.98 vertices and 25.76 edges on average. The number of distinct vertex and distinct edge labels are 81 and 3 respectively. For this study, we derive subsets from Chem_1M, each of which consists of N graphs and called Chem_N dataset.

Synthetic Datasets. The synthetic graph dataset is generated as follows: first, a set of S seed fragments (seed of a small subgraphs) is generated randomly, whose size is determined by a Poisson distribution with mean I. The size of each graph is a Poisson random variable with mean T. Seed fragments are then randomly selected and inserted into a graph one by one until the graph reaches its size. More details about the synthetic data generator are available in [5]. A typical dataset may have the following setting: it has 10,000 graphs and uses 100 seed fragments ($S = 100$) with distinct vertex labels, $L_V = 3$ and distinct edge labels, $L_E = 2$. On average, each graph has 50 edges ($T = 50$) and each seed fragment has 15 edges ($I = 15$). This dataset is denoted by Syn_10K.

Query Sets. There are six query sets Q4, Q8, Q12, Q16, Q20 and Q24. Each set Qi consists of 1000 query graphs with i edges. For AIDS_10k, we adopt the query set from [10]. In order to generate query sets for other datasets, a set of 1000 graphs whose size larger than or equal to 24 are randomly selected from the dataset. Then, edges are removed from graphs such that the remaining graphs still connected. These graphs constitute Qi when all graphs are of size i.

[1]http://dtp.nci.gov/.
[2]ftp://ftp.ncbi.nlm.nih.gov/pubchem/.

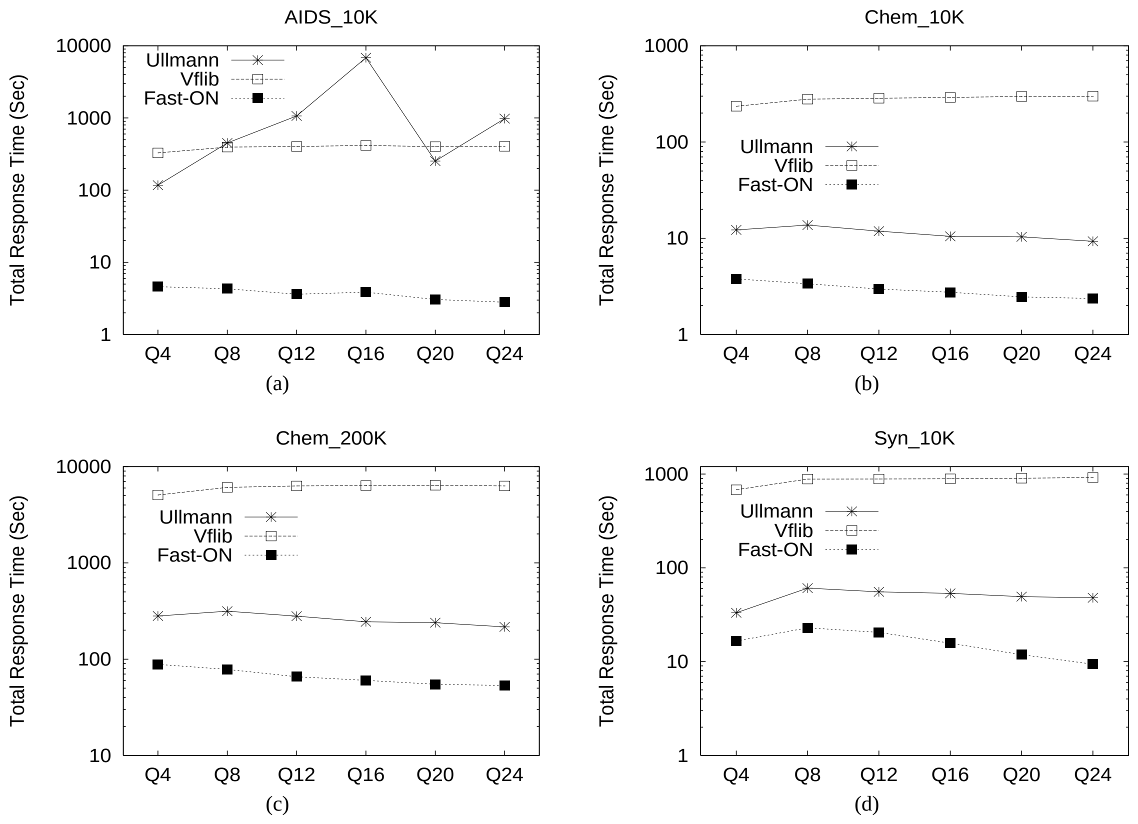

Figure 6. Total Response Time on Various Datasets

Datasets	`Fast-ON` **faster than Ullman**	`Fast-ON` **faster than Vflib**	$\lvert DLN_{\mathcal{D}}\rvert$
AIDS_10K	by 1-3 orders of magnitude	by 1 order of magnitude	796
Chem_10K	by 4 factors	by 1-2 orders of magnitude	352
Chem_200K	by 3-4 factors	by 1-2 orders of magnitude	1173
Syn_10K	by 2-5 factors	by 1-2 order of magnitude	287

Table 1. `Fast-ON` **performance against Ullman and Vflib, and # of distinct neighborhoods.**

5.2. Performance Study

Here, we compare the performance results of `Fast-ON` algorithm with those obtained on the same dataset by Ullman [3] and Vflib [4].

5.2.1 Effects of Optimizations

In this experiment we show the effect of each optimization independently, and the effect of both of them combined, on the performance of `Fast-ON`. For this purpose, we implemented three versions of `Fast-ON`, namely, `Fast-O` that uses only the first optimization Opt1, `Fast-N` that uses only the second optimization Opt2, and `Fast-ON` that uses both of the two optimizations.

Figure 5 plots the results obtained by running the three versions and Ullman algorithm on AIDS_10K for the different query sets. The figure shows that `Fast-N` is faster than `Fast-O` except for Q12 and Q16, where `Fast-O` shows the best performance. In addition to its influence on speed, the first optimization makes the algorithm less sensitive to query size. `Fast-ON` shows the best performance, it outperforms both `Fast-O` and `Fast-N`. This result confirm the fact that the two optimizations are neither independent nor conflicting, but they are complementary to each other. Finally, the figure shows how our new optimizations scale Ullman algorithm. `Fast-ON` outperforms Ullman algorithm by 1-3 orders of magnitude.

5.2.2 `Fast-ON` vs. Ullman and Vflib

Figure 6 reports the total response time obtained by running Ullman, Vflib, and `Fast-ON` on various datasets (AIDS_10K: Figure 6(a), Chem_10K: Figure 6(b), Chem_200K: Figure 6(c), and Syn_10K: Figure 6(d)). Table 1 reports how much `Fast-ON` is faster than Ullman and Vflib. The Table also reports the size of distinct labeled neighborhood of each dataset $\mathcal{D}$, $|DLN_{\mathcal{D}}|$. Notice that $|DLN_{\mathcal{D}}|$ is small for all datasets compared to the number of graphs $|\mathcal{D}|$. Thus, the containment cost of the labeled neighborhoods in `Fast-ON` is minimal. We can see that `Fast-ON` always spends less time compared with Ullman and Vflib. This happens because `Fast-ON` has better optimizations.

5.2.3 Scalability

Figure 7 shows the scalability of Ullman, Vflib and `Fast-ON` with respect to the number of graphs using the dataset Chem_1M and $Q8$. The figure shows that the three algorithms scale linearly. However, `Fast-ON` outperforms Ullman by factor three, and Vflib by more than one order of magnitude. Moreover, Vflib is not shown for 1000K graphs, since it failed to run on large datasets.

[3] Ullman Algorithm is also implemented in standard C++ with STL library support and compiled with GNU GCC.

[4] http://amalfi.dis.unina.it/graph/db/vflib-2.0/.

6. Conclusion

In this paper, we presented `Fast-ON`, an efficient algorithm for testing subgraph isomorphism problem which has proven to be NP-complete problem. `Fast-ON` is based on Ullman algorithm and reduces the search space as much as possible by ordering the vertices of query graph and by using the labeled neighborhood information. Experimental results on real and synthetic datasets demonstrate that `Fast-ON` outperforms Ullman and Vflib by 1-3 order of magnitude. Also `Fast-ON` has excellent scale-up properties with respect to the number of graphs.

References

[1] D. Cai, Z. Shao, X. He, X. Yan, and J. Han. Community mining from multi-relational networks. *Proc. of PKDD*, 2005.

[2] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. *SIGMOD*, pages 857–872, 2007.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE transaction on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[4] M. R. Garey and D. S. Johnson. Computers and intractability; guide to the theory of NP-completeness. *W. H. Freeman & Co.*, 1990.

[5] M. Kuramochi and G. Karypis. Frequent subgraph discovery. *Proc. of ICDM*, pages 313–320, 2001.

[6] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497520.

[7] E. G. M. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *IEEE transactions on knowledge and data enginnering*, 9(3), 1997.

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

[9] P. Willett. Chemical similarity searching. *J. Chem. Inf. Computer Science*, 38(6), 1998.

[10] X. Yan, S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. *SIGMOD*, pages 335–346, 2004.

[11] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta <= graph. *VLDB*, pages 938–949, 2007.