Journal of Engineering and Applied Sciences 11 (7): 1586-1597, 2016
ISSN: 1816-949X

Fast Graph Isomorphism Testing for Graph Based Data Mining with Improved Canonical Labelling

[1]D. Kavitha, [2]V. Kamakshi Prasad and [3]J.V.R. Murthy
[1]PVPSIT, Vijayawada, Andhra Pradesh, India
[2]Department of Computer Science and Engineering, JNTUH College of Engineering, JNTU, Hyderabad, India
[3]Department of Computer Science, JNTU, Kakinada, India

Abstract: In graph based data mining, graph/subgraph isomorphism testing used in mining frequent subgraphs plays key role and is time consuming. In a wide range of real applications, graph Isomorphism has significant role in retrieving the isomorphic graphs from a set of graphs. Canonical labelling of the graph has major impact on the efficiency of graph isomorphism testing. In this study, an algorithm is proposed to find canonical labelling in an efficient way and there by efficient isomorphism testing of labelled graphs. The proposed algorithm reduces search space based on the symmetries present in the graph there by making computation feasible to perform isomorphism testing on large databases for pattern mining.

Key words: Graph mining, graph isomorphism, canonical labelling, partition refinement, symmetry

INTRODUCTION

Graph mining aims at extracting useful knowledge from a large amount of structured data modelled as graphs (Cook and Holder, 2006). Attributed/labelled graphs are more appropriate to represent structured data as they offer a powerful way for modelling data of manyscientific and commercial applications (Gyssens *et al*., 1994; Kumar *et al*., 2000; Binucci *et al*., 2005; Amuthavalli, 2010; Frick *et al*., 1994). Labels of vertices and edges can represent different attributes of entities and relations among them. Labelled graphs have been used to address the problems of shape analysis, chemical data analysis, computational biology, social network analysis, web link and document analysis and computer networks due to the clarity in representation and efficient use in finding solutions (Sander, 1999; Maio and Maltoni, 1996; Zhou and Pei, 2008; Deshpande *et al*., 2005; Chartrand and Zhang, 2006; Koyuturk *et al*., 2004). Popular graph mining algorithms based on graph theory based approach are AGM (Apriori-based Graph Mining) (Inokuchi *et al*., 2003), FSG (Frequent sub-Graph discovery) (Washio and Motoda, 2003; Kuramochi and Karypis, 2004), gSpan (graph-based Substructure pattern mining) (Yan and Han, 2002), FFSM (Fast Frequent Sub-graph Mining) (Huan *et al*., 2003), etc.In order to extract knowledge from graphs, frequent subgraphs are very basic ones that can be discovered in a set of graphs that are useful at analysing and characterizing graph data, discriminating different groups of graphs, classifying and clustering graphs and building graph indices (Wale and Karypis, 2007; Flake *et al*., 2004; Yan *et al*., 2004). Mining frequent subgraphs (Bringmann and Nijssen, 2008; Jin *et al*., 2005) is a basic activity to find frequent subgraphs over a collection of graphs in a graph database or from a large single graph. The graph/subgraph isomorphism testing plays key role in the process of frequent subgraph (pattern) mining (Cordella *et al*., 2004). A graph isomorphism is a classic P or NP complete (Garey and Jhonson, 1979) problem in finding frequent graphs/subgraphs in graph mining. There are different approaches to solve the problem of finding isomorphisms of a general graph (Gross and Yellen, 2004), however most practical algorithms available in the literature are sub-divisible into two different categories. The algorithms in the first category proceed directly by taking the two graphs to be compared for isomorphism and try to find a match between them. These algorithms proceed by using a depth-first backtracking (Cordella *et al*., 2004; Ullmann, 1976) and by using heuristics to reduce the size of the search tree. On the other hand, the algorithms in the second category proceed by considering one graph at a time. These algorithms take a single graph, say Gand compute a function Cl (G) which returns a certificate or a canonical label of the graph. Canonical labelling of a graph consists of assigning a unique label to a graph such that the label is invariant under isomorphism.After obtaining the canonical label for each graph the isomorphism can be found by comparing the canonical labels. These two classes of algorithms, even though they differ in the way

Corresponding Author: D. Kavitha, PVPSIT, Vijayawada, Andhra Pradesh, India

they solve the isomorphism problem, make use of invariants. Practical applications of graph isomorphism testing do not confine to checking the individual pairs of graph. Often one must decide whether a certain graph is isomorphic to any of collection of graphs or one has a collection of graphs and needs to identify isomorphism classes in it. Such applications are not well served by the algorithms that can test graphs in pairs only.

The challenges that arise in the second category of isomorphism testing are canonical labelling and symmetry (automorphism) detection. Identifying canonical label itself is exponential. If a graph has |V| vertices, the complexity of determining canonical label of a graph is O(|V|!). Symmetry is a permutation of the graph's vertices that preserves the graph's edge relation. The symmetries of a graph map each labelling to another labelling. If all symmetries are extracted it may be sufficient to visit only one labelling from each equivalence class. This main feature is being exploredin this study is to eliminate the permutation computations involved in canonization.

Lots of well-known isomorphism test algorithms were developed (Ullmann, 1976) proposed a backtracking method that significantly reduces the size of the search tree and it is still one of the popular algorithm for exact graph matching (Foggia *et al*., 2001). Another (sub)graph isomorphism algorithm VF2 proposed by Cordell *et al*. (2004),also based on Depth-first Search (DFS) strategy, employed some feasibility rules that prune unpromising vertex pairs in the search space. VF2 is more efficient, especially for large graph sizes. VF2 as well as Ullmann, both can work efficiently for large labelled graphs with and without imposing any restrictions on the graph structure.The major drawback of these algorithms is that these algorithms are slow when the graphs being tested have many automorphisms, since they do not detect them. Conauto is an another direct algorithm tries to find a mapping between the two graphs using backtracking and prune the search tree using automorphisms in the graphs like canonical labelling algorithms process but without necessarily computing the whole automorphism group. All these algorithms are confined to check isomorphism for the individual pairs of graphs.

Coming to another category of graph isomorphism algorithms that uses canonical label to test the graph isomorphism. Babai and Kucera (1979) proposed a fast algorithm to compute canonical forms of general graphs in exp(n1/2+O(1) time. The most powerful algorithm currently available is McKay's Nauty package (McKay and Piperno, 2014) which is considered to be the first practical algorithm that employ the idea of Babai. Even it is one of the fastest graph isomorphism algorithm, it takes exponential time for some categories of groups. Furthermore, NAUTY does not allow graphs to have labels; hence it is not applicable to labelled graphs.

An alternative methods such as FSG (Kuramochi and Karypis, 2004) and minimum DFS (Yan and Han, 2002) canonical code are popular in frequent graph/subgraph mining. The FSG combines several types of vertex invariants to partition the vertices into equivalence classes. If the vertices of a graph with n vertices are partitioned into c classes $\pi_1, \pi_2, \ldots, \pi_c$, then the number of different permutations need to be considered in order to find that the canonical code is $\prod_i = 1 \sim c(|\pi_i|!)$ which is substantially faster than the n! Permutations required by the earlier approaches (Yan and Han, 2002) proposed a canonical labeling named DFS Lexicographic Order that searches a graph using Depth-first Search (DFS) strategy, they traverse the graphs and label them canonically with the minimum DFS code. Then, after extracting these codes and filtering them, instead of GED measurements, they used Levenshtein distance (i.e., string edit distance, SED) to measure the similarity between two graphs i.e., between the source graph and graphs in database. This is used in frequent subgraph mining algorithm gSpan.

Definition 1: A labelled graph is defined as quadruple G = (V, E, L, LF), where V is a set of vertices $E \subseteq V \times V$ is a set of edges, L is a set of labels and LF is a functionthat gives a unique label to each vertex of G. Given two labelled graphs G = (V, E, L, LF) and G' = (V', E', L', LF'), we say that G is isomorphic to G', written' $G \simeq G'$, if there exists a bijection $f: V \rightarrow V'$, called isomorphism such that:

1. $\forall (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$
2. $\forall v \in V, L(v) = L'(f(v))$

Definition 2: A canonical labelling of graph G is said to be an isomorphism-invariant labelling of G'svertices, i.e., two graphs G and G' have the same canonical labelling if and only if they areisomorphic to each other.A function $C: G \rightarrow G$ is a canonical form iff:

(i) $\forall G \in G: C(G) \simeq G$
(ii) $\forall G, G' \in G: G \simeq G' \Leftrightarrow C(G) = C(G')$

Formally, given a class K of graphs which is closed under isomorphism, a canonical labelling algorithm assigns a unique label to each graph in K, in such a way that two graphs in K are isomorphic if and only if the obtained labelsof graphs coincide. One way to define a canonical isomorphism function is to specify a total order on graphs with n vertices. The ordering is a lexicographic total order induced by a fixed total ordering on. Unfortunately Chas the drawback of being difficult to compute. Typically there is no alternative to checking essentially every permutation of the vertices to see whether it produces the graph that is than greatest. One

reason for the weakness of this sort of approach is that it does not exploit any graph theoretical information. One starting point might be to have vertices in the canonical isomorph appear in increasing order of degree. But starting label withsmallest degree vertices requires more complex computation as they are large in number in a general graph and computation of automorphism at this level is quiet complex and of less use. So, this algorithm starts its canonical label with highest degree vertices. With regarding this, the vertices of a labelled graph are partitioned with the properties as defined in definition 3. Clearly this is not sufficient to determine the canonical isomorph. However, this local information can then be propagated around the graph. For instance, if there is a unique vertex vof some particular degree, then the neighbours of vcan be distinguished from the non-neighbours. Iterating this idea, the second neighbourhood of vcan then be distinguished and so on. To do so,the vertex invariant properties such as degree, label of vertex, edge relationships between vertices, label of edges, neighbours to identify symmetry etc. areused and defined in the following definitions.

Definition 3: The vertices of a labelled graph G are partitioned intopdisjoint non empty sub sets denoted by, an ordered partition $T(G) = \{\pi_1, \pi_2, \ldots, \pi_p\}$ if, satisfies the following properties:

(1) For all vertex u, v $\epsilon\pi_{ik}(G)$, deg (u) = deg (v), $\forall\mu,v \epsilon \pi_k$ (G) $1 \leq k \geq p$
(2) For all vertex $\mu \epsilon\pi_k$ (G) and v $\epsilon\pi_l$ (G), deg (u) >deg(v) if k<l

Definition 4: Let be an equitable ordered partition of n vertices with a nontrivialpart π_i, the equitable refinement $R(\pi_i)$ of the ordered partition (π_i) is $R(\pi_i) = \pi_{i1}, \pi_{i2}, \ldots, \pi_{ic}$ if satisfies the following properties:

(1) For all vertexu, u, $v\epsilon\pi_{ik}(G)$, lbl(u) = lbl (v)$\forall\mu,v \epsilon\pi_{ik}$ (G), $1 \leq k \geq c$
(2) For all vertex $\mu \epsilon\pi_{ak}$ (π_k), $v\epsilon\pi_{bk}(\pi_k)$, lbl(μ), <lbl (v), if a<b

Definition 5: The set of symmetries of G forms a group under functional composition and is called the symmetry group of G and is denoted by Sym (G). An automorphism (a symmetry) of graph G is a permutation of G's vertices that preserves G's edge relation, i.e., G = G. The automorphism group Aut(G) of a graph G is theset of all automorphisms of G with permutation composition as group operation.

Lemma 1: A graph G is vertex-transitive if for every vertex pair u,v $\in V_G$ there is an automorphism that maps u to v.

Lemma 2: A graph G is edge-transitive if for every edge pair d,e $\in E_G$ there is an automorphism that maps d to e. Vertex orbits are the equivalence classes of the vertices of a graph G identified by the automorphism. The equivalence classes of the edges are called edge orbits.

All vertices in the same orbit have the exact same degree, label and neighbours.All edges in the same orbit have the same pair of degreesat their endpoints. Anautomorphism of graph G is a structure-preserving permutation on$V_G$along with a (consistent) permutation γ_E on E_G . We may writeγ_v,γ_E. The adjacency set of a vertex v of a graph g, denoted as adj (v) is a set of vertices directly connected (adjacent) to v.

Definition 6; vertex triplet: For a vertex v with madjacent vertices $u_1, u_2, \ldots, u_m$, vertex tripletTri (v) is a string defined as: $dsig_v$+ $lsig_v$+ $esig_v$ where "+" is string concatenation and which satisfies the following criteria:

(1) For each v∈S, $dsig_v = (d(u_1), d(u_2), \ldots, d(u_m))$ where deg (u_k) = deg $(u_{(k+1)})$, for all k, $1 \leq k \leq m-1$
(2) For each v∈S, $lsig_v = (l(u_1)\| l(u_2)\| \ldots \| l(u_m))$ where u_i is the in the order of $dsig_v$
(3) For each v∈S, a sequence $esig_v = (l(u_1, v)\| l(u_2 v)\| \ldots \| l(u_m v))$ where u_i is the in the order of $dsig_v$

Therefore, the triplet of vertex v with m adjacent vertices is composed of the triples in the form of adjacent vertices degree ($dsig_v$), adjacent vertices label ($lsig_v$) and adjacent vertices edge label ($esig_v$). These triples are ordered first by vertices degree, then by vertexlabels and then by edge labels. Notice that the vertex triplet is not obtained by calculating all possiblepermutations; it is attained by sorting the degrees of adjacent vertices as described in the above definition 6 rule 1. And then adjacent vertices label and adjacent vertices edge label are obtained just by concatenating the labels in the order of vertices obtained with rule 1. The intention behind the design of vertextriplet is to combine the degree, label and the edge label of adjacent vertices information of a vertex into a string-based representation. Therefore, the lexicographic order of vertex triplets is totally ordered as normal strings.

Lemma 1: Given two vertex triplets Tri (v_1) and Tri (v_2) of vertices v_1 and v_2 of a labelled graph G, respectively, v_1 is not symmetric to v_2 if Tri (v_1)≠Tri (v_2).

Proof: We prove this lemma by showing that if v_1 is symmetric to v_2 then Tri (v_1) = Tri (v_2). Since, v_1v_2, there

exists abijection f that maps each adjacent vertex v_i of v_1 to a vertex $f(v_i)$, namely w_i, of v_2. Otherwise, the adjacency relations preservations such as adjacent vertex degrees, vertex labels and edge labels must be violated between these twovertices. Since the lexicographic order of adjacent vertices relationships are totally ordered and the triplet of a vertex is the stringobtained by concatenation, these two vertex triplets must be equal. Notice that Lemma 1states a necessary condition to identify symmetry between two vertices. This symmetry property of vertices can be used to avoid permutation when generating canonical label of a graph that have symmetrical vertices.

The main difference between the proposed algorithm and previous works (Ullmann, 1976; McKay and Piperno, 2014) is that the search tree of those works is global, i.e., the backtracking of the entire matching process may cross the boundaries of partition classes such that the maximal number of nodes in this global search tree becomes the product of all the possible matching enumerations within each partition class. In this algorithm, search space exists locally for the vertices that correspond to a single partition class and there is no backtrackingsand further the automorphism property is used to eliminate the permutation computations.

MATERIALS AND METHODS

The proposed method: In this study, a new algorithm Fast-CL (Fast Graph Isomorphism testing for graph based data mining with improved canonical labelling) is proposed to perform the isomorphism testing of simple labelled graphs based on the canonical label. Our algorithm makes use of invariant properties of labelled graphs tofind symmetries in the graph and to improve the overall performance. The key features of the proposed algorithm are as follows:

- This algorithm does not use backtrack to traverse the search space
- Starts labelling with higher degree vertices to reduce complexity involved in computation of permutations required to construct canonical label and to use automorphisms efficiently in label construction
- Encodes the vertex invariants label, degree and adjacentvertices into vertex triplet and use of adjacency lists to find symmetry
- Narrows down the search space that reduces the time needed for constructing canonical label by using symmetries in a graph
- Automorphisms of graph G determined at non terminal nodes are used to get rid of permutations

Construction of canonical label: The canonical label of a graph Cl(G) is defined to be a unique code (e.g., string) that is invariant on the ordering of the vertices and edges in the graph.It can be the smallest or the largest stringobtained by concatenating all the vertex labels followed by the columns of the upper triangular entries of the adjacency matricesover all possible permutations of vertices.

In order to reduce the number of possible permutations, vertex invariants are used. Vertex invariants are some inherent properties of the vertices that arepreserved across isomorphism mappings such as vertex labels and degrees. Vertices with the same values of the vertex invariantsare partitioned into the same equivalence partitions. If the vertices of a graph are partitioned into p partitions $\pi_1, \pi_2,\ldots,\pi_p$, the amount of all possible codes need to be generated in order to obtain the canonical one is $O(\Pi i = 1 \sim p(|\pi_i|!))$ by Kuramochi and Karypis (2004).

Here is the basic algorithm to generate canonical label of a graph G. It is basically based on theequivalence partitioning of vertices with the added optimization that refine each part before going to next one. This algorithmcomputes the vlbl as the first step in canonical code construction which is in turn used to construct elbland canonical code.

Algorithm

Input: A labelled graph G
Output: Canonical label of a graph G-Cl(G)
Begin
1. $(\pi_1,\pi_2,\ldots,\pi_p)$ /*Make ordered partition Π (G)*/
2. p←| (G) | /* Let p be the number of vertex parts of
3. fori = 1 to p do
4. If ($|\pi_i|$==1)
5. vlbl←vlbl ||v (π_i) /*vlblconstruction*/
6. else if($|\pi_i|$>1)
7. refine_partition(π_i)
8. end for
9. for j = 2 to n do
10. for k = 1 to n do
11. v_k←v∈vlbl
12. v_j v←vlbl
13. if (v_k, v_j) ∈ E(G)
14. elbl←elbl ||l(v_k, v_j)
15. elseelbl elbl|| 0
16. end for
17. end for
18. clbl←vlbl ||elbl
end

This is the algorithm to construct the canonical label of a graph Cl(G).The first step is to identify the vertex label vlblthat contains labels of vertices as a string in the order of vertices such that the canonical label is lexicographically small/high. String obtained by concatenating the columns of the upper triangular entries of the adjacency matrix was referred as elbl. Canonical label be a string that was obtained by concatenating vlbl and elbl.

In step1, predefined vertex invariant, degree of the vertex is used to create an initial orderedpartition (Π) of the vertices of the graph G into p parts as stated in definition 3. A trivial part is a part of size one. If the ordered partition (Π) contains discrete and trivial parts, append the elements of each part to the vlbl in the order of partitioning done as vertices in different parts of have already been distinguished from each other with the properties as stated in definition 3. In the usual canonical labelling procedure, permutation of vertices is used to get the order of vertices of a non-trivial part. To reduce the complexity in computation involved in finding all possible permutations of identical vertices inside p parts, an algorithmrefine_partition () is developed that further refines each part using vertex invariants and edge relations. Step 3-8 in the algorith mdefine this process and is the first step of canonical label construction. This yields a vlbl string that is enough to construct canonical label of graph without permutations.

Step 9-15 constructs edge label based on the order of vertices of constructed vlbl and edge relations between them i.e. second step of labelling. Obtain the canonical label of graph Cl (G) by concatenating the string vlbl followed by the string elbl.

Refinement using vertex invariants: In the remainder of this section, the procedurerefine_partition() used to refine an equitable partition is presented. The procedure acceptsan ordered part as input and returns refinedclasses π_{i1}, π_{i2}, ... π_{ic} of π_i. These classes $\pi_i$are further distinguished using automorphismproperty present in the class.

Initially, the main algorithmstarted by forming anequitable ordered partition of vertices, thereby extracting all the degree information. The children of an equitable ordered partition in the search tree do not correspond to all possible splittings of . In order to distinguish verticesof π_i and to find the symmetry, other graph theoreticalinformationsuch as label of vertex and edge relations between vertices are exploited. The definition 4 describes the way to identify these distinctions thus to make further partition refinement. At each stage, the first non-trivial partof π is chosen to split. We record in the search tree not only the current equitable ordered partition but also the sequence of vertices used for splitting.

Algorithm: Refine_partition ()
Input: A partition of graph ((G))
Output: Refined partition
C←Connected set of vertices in a class $\pi_{ic}$to the vertexof vlbl
UC←Unconnected set of vertices in a class $\delta_{ic}$to the vertexof vlbl
Begin
1. Make $\pi_i$into classes π_{i1}, π_{i2}.... π_{ic} based onL (v) inπ_i
2. $\pi_i\pi_{i1}$, π_{i2}....π_{ic}
3. for each class π_{ij} inπ_i do
4. if class $\pi_{ij}$is singleton
5. add operand v ofπ_{ij} to vlbl
6. else if vlbl is null
7. find_symmetry(π_{ij})
8. else
9. for each vertex v_a inπ_{ij} do
10. for each vertex $v_b$invlbldo
11. if L (va,vb) ϵE
12. C←C U v_a
13. else
14. UC UC U v_a
15. end if
16. end for
17. end for
18. end if
19. if UC is singleton
20. add operand UC_v of UC to vlbl
21. else if |UC|> 1
22. find_symmetry(UC)
23. end if
24. refine_connected(C)
25. end for
end

Vertices in different parts Π of have already been distinguished from each other with the properties as stated in definition 3 butnot the vertices in the same part π_i. In labelling computation to discard numerous permutation possibilities in each part π_i, refine_partition () algorithm invokes partition refinement to propagate the constraints of the graph, i.e., the graphs vertex degrees, vertex labels, edge labels and edge relations. Step 1 classifies each part π_i into classes π_{i1}, π_{i2}, ... π_{ic} based on the label of vertices.If the classes are discrete and singleton, append the vertices of the classes to the vlbl in the order of partitioning. Divide the vertices in a non-singleton class into Unconnected (UC) and Connected (C) set of vertices based on the edge relations between the vertices of the class to the vertices in the vlbl till constructed as the edge relation with the vertices in the vlblhas profound influence on the canonical label. It is important to employ well-devised data structures for performing the computation ofdifferent set of vertices that have different properties, Data structures UC and C are designed tomaintain list of connected and unconnected vertices that are resultant of first level of refinement and make available for further refinement of vertices of each class π_{ij}. Ccomprises set of vertices that have edge relations with the vertices of vlbl constructed till and UC contains that haven't. As the result of step 6-16, if the number of elements in the unconnected set UC is one, there is no scope for further refinement and hence append the element of UC set to vlbl. The elements in thenon-singleton set needs to be explored to find automorphism. The algorithmfind_symmetry () is used to find automorphisms between set of vertices. Vertices of connected set CS are refined and added to vlbl by invoking the algorithm refine_connected ().

Automorphism discovery: The identification of automorphic groups and using them in eliminating the permutations is the significant step employed in this algorithm. Nauty recognizes an automorphism if two different leaf partitions result in the same adjacency matrix after relabeling the vertices (McKay and Piperno, 2014). Thefind_symmetry () algorithm differs with Nauty in recognising and using symmetriesthat may appear both at leaf and non-leaf terminals that areinferred from triplet of vertices and edge relations i.e., structure of the partitionwhich results same adjacency matrix.

Algorithm: find_symmetry( )
Input: A set of vertices
Output: Symmetric group
UCset of vertices in an orbit to discover automorphism
S←set of adjacent vertices to a vertex $v_i \in$ UC
RU←set of vertices need to refine
begin
for each v ∈ Ucdo
S← adj(v)
compute Tri_v ($dsig_v$, $lsig_v$, $esig_v$)
end for
split UC into number of subsetsU_1, $U_{2,...}$ U_k for all $v_i \in U_k$ and Tri_{vi}
Tri_{vj}
for each U_i do
if ($U_i$is singleton)
append the vertex v to vlbl
else
compute_auto(U_i)
append any of the vertex from U_{ij} to vlbl
if ($U_{ij}$is singleton)
append the vertex v $U_{ij}$to vlbl
else
refine (U_{ij})
end if
end for
return
end

Let adj (v) returns the list of neighbours of v that are in the set S, we could then define a symmetry function.As stated in definition 6, for each vertex compute vertex tripletTri$_v$ which comprises three parts dsig, lsig, esig. String dsig is the sorted order of degrees of its adjacent vertices, string lsig is the corresponding labelsof adjacent vertices and string esig is the respective edge labels. Finding this triplet is itself time consuming process. While constructing the adjacency list in this algorithm, adjacent vertices are added to the list in the required sorted order as per definition 6 and hencein this algorithm time complexity is reduced. Subsequently the amount of work required to compute automorphisms is reduced. After finding Tri$_v$, splitting the vertices into subsets based on automorphism and appending vertices of subsets to vlblis same as explained in the above algorithms. The algorithm refine() is used to find edge relations for the set of vertices that are in automorphic group and refine_connected() algorithm is used to append vertices that have edge relation with vertices of vlbl.

Compute_auto() verifies the preservations of actual adjacent vertices for the vertices of $U_i$and assures the complete automorphism by checking the symmetry of adjacent neighbour vertices of automorphic groups.

Algorithm: compute_auto( )
Input: a set of vertices U_i
Output: automorphic vertices groups U_{ic}
begin
for each $u_i$inU_i do
find l(v)based on edge relation with the vertices
end for
split U_i into sub sets U_{i1},U_{i2},...U_{ic}based on edge relations
return
end

Algorithm: Refine( )
Input: a set of vertices C
Output: Append vertices to vlbl otherwise proceed to refine_connected () algorithm
begin
for each $u_i$in Ru do
find edge relation with the vertices of vlbl
if (no edge relation)
append to vlbl
else
set status of u
append to Cu
end for
refine_connected(Cu)
return
end

Algorithm: refine_connected()
Input: a set of connected vertices CS
Output: Append vertices to vlbl otherwise proceed to find_auto () algorithm
begin
while (CS is not empty)
set status of v_c
split C into sub sets C_1,C_2,...$C_c$based on status
if (|C_i|>1)
find_auto(C_i)
else
append$v_c$of $C_i$to vlbl
end while
return
end

Vertex individualization is an important task in a symmetrical group. Compute_auto() algorithm substantiates find_symmetry() algorithm in individualizing vertices in a symmetrical groupusing the vertex transitive and edge transitive properties of symmetry. Generally, the individualization of a vertex corresponds to select a subgroup of permutations in fixing that vertex which was avoided in this algorithm. What would be the next node in the vlbl is selected based on the local properties of adjacencies of nodes and these are algorithmised in refine ( ) and refine_connected ().

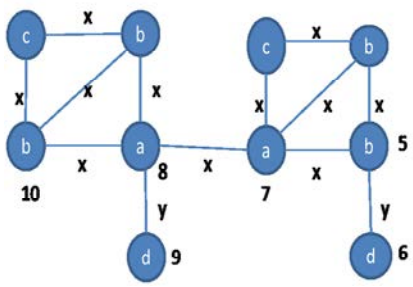
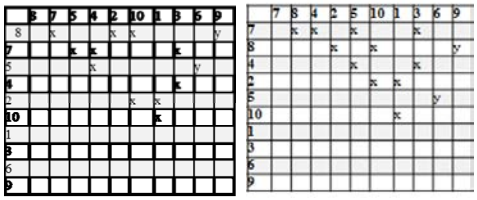


Fig. 1: A labelled graph G



(a) (b)

Fig. 2: Finding canonical label using adjacency matrix

Canonical label construction using the knowledge of symmetry: This section presents the behaviour of the proposed algorithm for unique code construction of graph G shown in Fig. 1. Let G be the graph with 10 vertices and 13 edges. Simple way of defining the canonical label of a graph is to obtain the string by concatenating the upper triangular entries of the graph's adjacency matrix when this matrix has been symmetrically permuted so that this string becomes the lexicographically largest (or smallest) over the strings that can be obtained from all such permutations. Two permutations of its adjacency matrices are illustrated in Fig. 2 a, b leads to its canonical label "aabbbbccddx0x0xxx000x000x0000xx0x0x00000y00000y00 000000".

In this code, "aabbbbccdd" is obtained by concatenating the vertex-labels in the order that they appear in the adjacency matrix and "x0x0xxx000x000x0000xx0x0x00000y00000y00000000" is obtained by concatenating the columns of the upper triangular portion of the matrix. If a graph has |V| vertices, the complexity of determining its canonical label using this scheme is in O(|V|!) i.e., 10! making it impractical even for moderate size graphs. Even the FSG algorithm requires (2!X4!X2!X2!) permutations to compute canonical label for labelled graphs.

The typical behaviour of algorithm is exemplified in Fig. 3. Coming to Fast-CL's algorithm, code construction of graph G starts by forming the equitable partitioning (using algorithm 1), thereby extracting all of the initial degree information. For graph G, the equitable partitioning is shown in step 1. Having extracted an equitable partition, apply further refinement to split the non-trivial parts. At each stage, nontrivial parts are chosen to further refine in the order of partitioning done. An algorithm refine_partition() is applied on the first non-trivial part π_1. As labels of nodes 7 and 8 are same, find_symmetry () algorithm is invoked and 8|7 is the order of vertices (by computing Tri_v) , then edge connectivity with vlbl vertices information is used in next refinement as edge relation with vlbl vertices results bigger code than that hasn't. Symmetry checking is used as the last step in the refinement process to identify symmetrical vertices. In the above graph (G), the vertices 7 and 8 in the partition $\bar{u}_1$ are distinguished at this stage and ordered as 8, 7 in vlbl. Coming to refinement of next part π_2, nodes 2 and 10 are connected to node 8 and nodes 4 and 5 are connected to 7. As nodes 2 and 10 connected to node 8 the first node of vlbl, nodes 5 and 4 obtain higher priority to refine and got the next positions as 8|7|5|4. The remaining nodes of this part are automorphic and takes either order 2|10 or 10|2. The nodes in other parts are refined in the same manner and the vlbl order for the graph G is 8|7|5|4|2|10|1|3|6|9 i.e., aabbbbccdd and the canonical label isaabbbbccddx0x0xxx000x000x0000xx0x0x00000y00000y0 0000000.

Graph isomorphism testingusing fast-CL: Let's test the performance of the algorithm to identify isomorphic and non-isomorphic graphs shown in Fig 4. The ordered partitions of G_1 are (1 4| 2 3). As vertices 1 and 4 of part π_1 are symmetrical the vlbl order may be either 14 or 41. The same appeared for nodes 2 and 3 when refining part π_2. So the constructed vlblfor $G_1$is 1423, i.e., aabb leads to canonical label aabbxxxxxx. For the graph G_2 the ordered partitions are (1 3|2 4) and by refining the partitions we got the vlbl1324 and canonical label aabbxxxxxx. Coming to the graph G_3 the ordered partitions are (1 4| 2 3) and the vlblorder after refinement of π_1 is 14 as the node 1 has $esig_v$ xxx and node 4 has xxy. The same applied for refinement of π_2 and finally the label of graph G is aabbxxxxyx. By comparing the labels, we can observe that graphs G_1 and G_2 are isomorphic graphs and graph $G_3$is non-isomorphic graph.

Time complexity analysis: The time needed for Algorithm 1 is the time needed for making ordered partitioning . In algorithm refine_partition (), the time required to make refinement of parts i.e., to make classes and to add vertices to the vlbl is O (π_{ij}). The dominating part is finding symmetric groups using find_symmetry () and compute_auto() algorithms. In these routines, the complexity involved in computing the signature Tri_v for

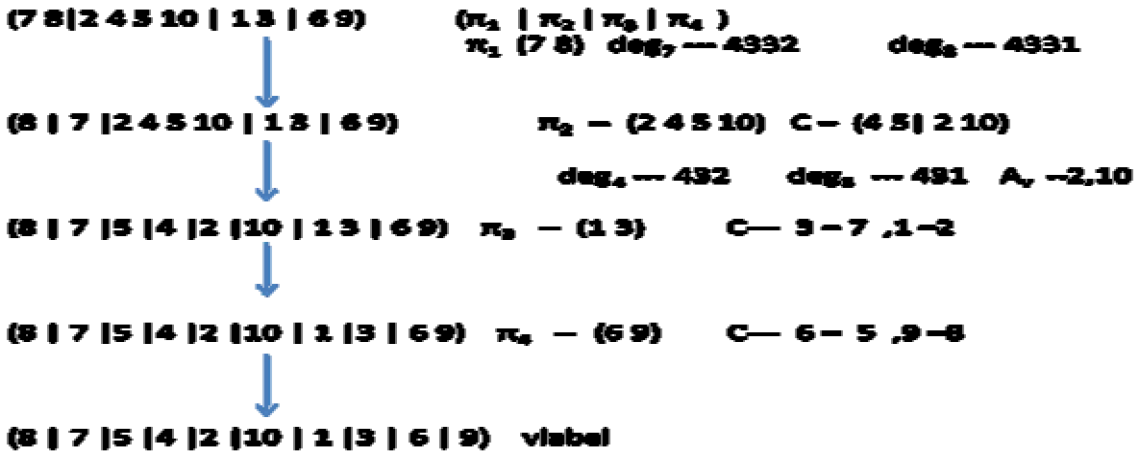


Fig. 3: Generation of vertex label using Fast-CL algorithm for the graph G

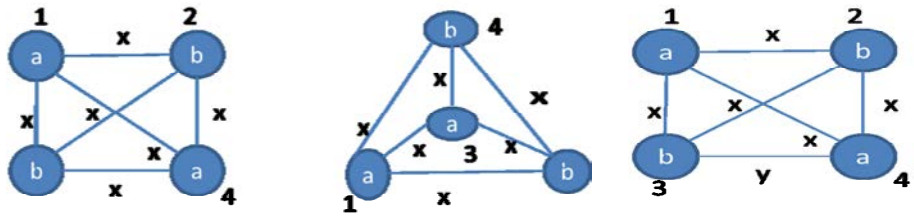


Fig. 4: Test for graph isomorphism

each vertex in a class was avoided at the time of graph construction by representing graph as an ordered adjacency list. In other algorithms finding edge relations and arranging them in an order requires order of the elements in a set. So the time needed to compute order of vertices for each part is $O(\pi_i)$ and hence the time needed to compute vlbl is $O(\pi_i)$.

The proposed method will produce the same canonical code for all isomorphic graphs and different canonical codes for non-isomorphic graphs. The presented algorithm can also be used to find isomorphism between pairs of graphs by comparing the partitions and vlbl resulted at each step.

RESULTS AND DISCUSSION

Experimental results: Fast graph isomorphism testing for graph based data mining with improved canonical Labellingimplements a general purpose algorithm aimed at reducing the search space associatedin constructing canonical label with the knowledge of the automorphism group of a graph. Fast-CL was implemented in C language. The experiments were carried out on a Intel® Pentium® Dual CPU T3400 @ 2.17 GHzwith 4GB RAM. A comprehensive performance study has been conducted in experiments on both synthetic and real world data sets. Synthetic graphs are selected from the library of benchmarks (Santo *et al*., 2003). These are randomly connected graphs that each vertex pair has a probability of η that characterizes the density of the graphs. A graph with n vertices has $n^2\eta$ edges,and each vertex has $n\eta$ connected edges in average. In Fig. 4, the results obtained on randomly connected graphs are presented using two distinct values for the parameter η.

The plots show the average execution time (in seconds) as a function of the number of nodes. By observing the results of Fig. 5a and b, dense graphs required little bit of much time. However the time required to compute canonical label of graphs is considerably very less. To evaluate the performance on real datasets, we used the data sets in a standard graph library available at. It provides information on the anti-cancer screen tests with different cancer cell lines. Each dataset belongs to a certain type of cancer screen with the outcome active or inactive. The dataset is sparse, containing 66 vertices types and four types of edges. The largest graph has 214 vertices and 214 edges, on an average 43 vertices and 45 edges. Tests are conducted on Yeast and MCF-7 graphs. This dataset is useful to evaluate the heuristics of symmetry for eliminating permutations during label computation (Fig. 6).

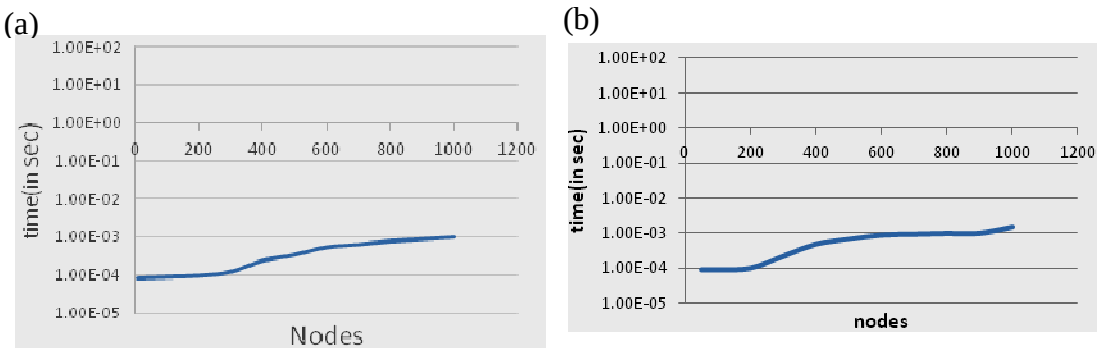


Fig. 5: a) Randomly connected graphs-η = 0.01, b) randomly connected graphs η = 0.1

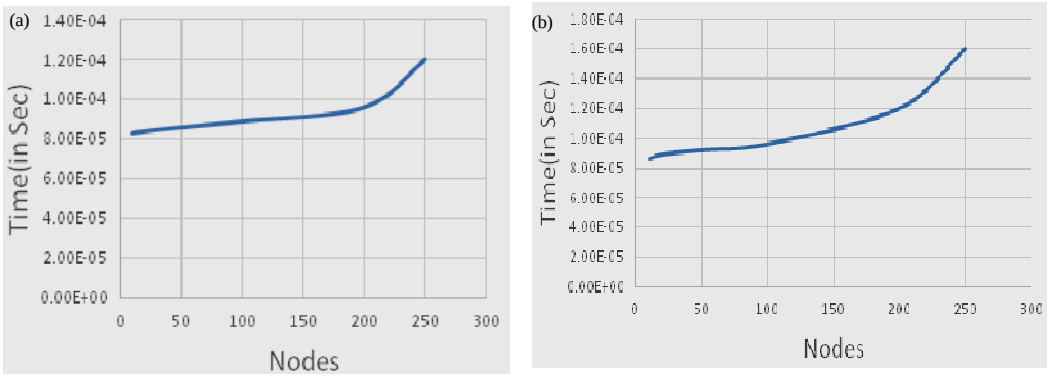


Fig. 6: a) Yeast and b) MCF-7

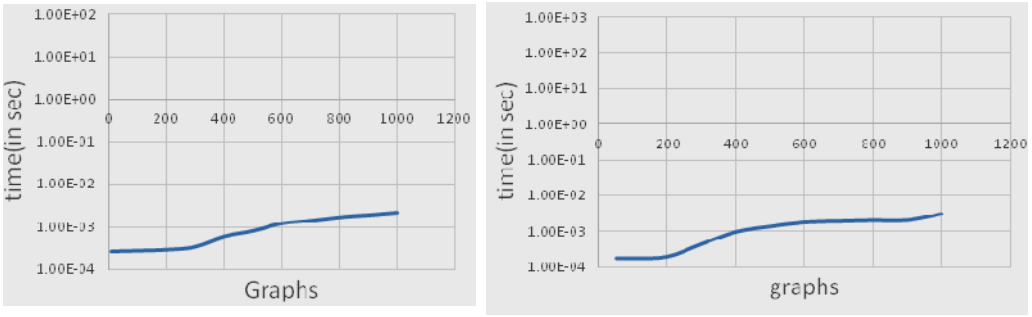


Fig. 7: Canonical label computation on aviation data

Aviation (ailab.wsu.edu/subdue). This dataset contains a list of records extracted from the aviation safety reporting system database. Each record corresponds to an event and information is represented by a graph having two typesof nodes and edges. The first type of nodes represents the events (and are labeled with the ids of the event) while the second represents information regarding event. Aviation consists of100K nodes and 133 K edges. Note that Aviation is a fundamentally different dataset when compared with the previous ones. TheAviation graph has on average one edge per node, thus, it is verysparse. Also it has a very large number of distinct node labels. Directed edges are converted into undirected and experiments are conducted on randomly sampled parts of graph and results are plotted in Fig. 7. The performance of the algorithm has been evaluated on multiple instances of graphs from the above defined data sets. Observe that there are no significant differences in the execution times for different cases. The algorithm is fast and consistent for different families of graphs. In order to verify the effectiveness of theproposed algorithm on set of graphs,the experiments are conducted on MCF-7 database that has nearly 40 K graphs. The graphs are randomly chosen from the active

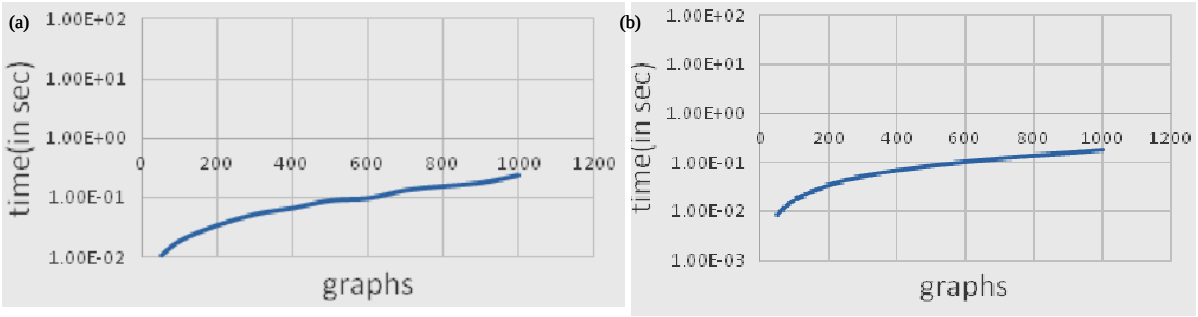


Fig. 8: Time taken to compute canonical label of graphs in a graph database; a) active and b) inactive

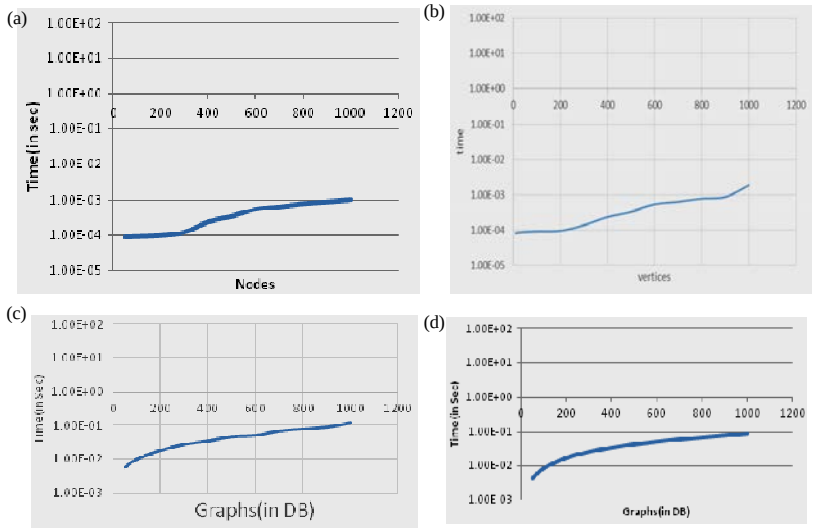


Fig. 9: Performance evaluation for testing isomorphism; a) random connected graphs $\eta = 0.01$; b) randomly connected graphs $\eta = 0.1$; c) active and d) inactive

and inactive sets. The time required to compute canonical label of set of graphs on MCF-7 database graphs is shown in Fig. 8. The compound datasets used in the experiments are useful in characterizing the effectiveness of heuristics tofind symmetry and thus to eliminate permutations in canonical label computation.

The average time required to perform isomorphism testing on pairs of randomly connected graphs is shown in Fig. 9a and b. Finally, to verify the effectiveness of the proposed algorithm on set of graphs as the main focus of the algorithm is to conduct isomorphism testing on a set of graphs, the experiments are conducted on MCF-7 database plotted in Fig. 9c, d. From the results of Fig. 9, it appears that the proposed algorithm is more convenient to perform isomorphism testing on set of graphs. Observe that the proposed algorithm is fast and consistent to perform isomorphism testing on labelled graphs of different families.

CONCLUSION

The emphasis of proposed algorithm is to provide an efficient canonical labelling for labelled graphs thus make available to perform fast and proficient isomorphism testing for a set of graphs in a graph database.

The results obtained in preliminary tests confirmed the effectiveness of the proposed approach. The algorithm is able to construct canonical label without using permutations for labelled graphs that have automorphisms.

The algorithm can also be applied to unlabelled graphs as considering the label of all vertices and edges as single. This algorithm can also be modified to find isomorphism between a pair of graphs just by comparing the results at each and every step of our algorithm.

REFERENCES

Amuthavalli, K., 2010. Graph labeling and its applications-some generalization of odd mean labelling. Ph.D. Thesis, Mother Teresa Womens University, Kodaikanal, India.

Babai, L. and L. Kucera, 1979. Canonical labelling of graphs in linear average time. Proceedings of the 20th Annual Symposium on Foundations of Computer Science, October 29-31, 1979, IEEE, New York, USA., pp: 39-46.

Binucci, C., W. Didimo, G. Liotta and M. Nonato, 2005. Orthogonal drawings of graphs with vertex and edge labels. Comput. Geometry, 32: 71-114.

Bringmann, B. and S. Nijssen, 2008. What is Frequent in a Single Graph?. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, T. Washio., E. Suzuki, K.M. Ting and A. Inokuchi (Eds.). Springer, Berlin, Germany, ISBN:978-3-540-68125-0, pp: 858-863.

Chartrand, G. and P. Zhang, 2006. Introduction to Graph Theory. McGraw-Hill, New York, USA., ISBN: 978-0-07-061608-0, Pages: 447.

Cook, D.J. and L.B. Holder, 2006. Mining Graph Data. John Wiley & Sons, Hoboken, New Jersey, ISBN:978-0-471-73190-0, Pages: 469.

Cordella, L.P., P. Foggia, C. Sansone and M. Vento, 2004. A (sub) graph isomorphism algorithm for matching large graphs. IEEE. Trans. Pattern Anal. Mach. Intell., 26: 1367-1372.

Deshpande, M., M. Kuramochi, N. Wale and G. Karypis, 2005. Frequent substructure-based approaches for classifying chemical compounds. IEEE Trans. Knowl. Data Eng., 17: 1036-1050.

Flake, G.W., R.E. Tarjan and K. Tsioutsiouliklis, 2004. Graph clustering and minimum cut trees. Internet Math., 1: 385-408.

Foggia, P., C. Sansone and M. Vento, 2001. A performance comparison of five algorithms for graph isomorphism. Proceedings of the 3rd IAPR-TC15 Workshop Graph-Based Representations in Pattern Recognition, May, 2001, Italy, pp: 188-199.

Frick, A., A. Ludwig and H. Mehldau, 1994. A Fast Adaptive Layout Algorithm for Undirected Graphs (Extended Abstract and System Demonstration). In: Graph Drawing, Tamassia, R. and I.G. Tollis (Eds.). Springer, Berlin, Germany, ISBN:978-3-540-49155-2, pp: 388-403.

Garey, M. and D. Johnson, 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. 11th Edn., W.H. Freeman and Company, New York, ISBN-10: 0716710455.

Gross, J. and J. Yellen, 2004. Handbook of Graph Theory. CRC Press, Florida, USA.

Gyssens, M., J. Paredaens, V.D.J. Bussche and V.D. Gucht, 1994. A graph-oriented object database model. IEEE. Trans. Knowl. Data Eng., 6: 572-586.

Huan, J., W. Wang and J. Prins, 2003. Efficient mining of frequent subgraph in the presence of isomorphism. Proceedings of the 3rd IEEE International Conference on Data Mining, Nov. 19-22, Melbourne, FL., pp: 549-552.

Inokuchi, A., T. Washio and H. Motoda, 2003. Complete mining of frequent patterns from graphs: Mining graph data. Mach. Learn., 50: 321-354.

Jin, R., C. Wang, D. Polshakov, S. Parthasarathy and G. Agrawal, 2005. Discovering frequent topological structures from graph datasets. Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Databases, Aug. 21-24, Chicago, IL., pp: 606-611.

Koyuturk, M., A. Grama and W. Szpankowski, 2004. An efficient algorithm for detecting frequent subgraphs in biological networks. Bioinf., 20: 200-207.

Kumar, R., P. Raghavan, S. Rajagopalan, D. Sivakumar and A. Tompkins *et al.*, 2000. The web as a graph. Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-18, 2000, ACM, Dallas, Texas, USA, ISBN:1-58113-214-X, pp: 1-10.

Kuramochi, M. and G. Karypis, 2004. An efficient algorithm for discovering frequent subgraphs. IEEE. Trans. Knowl. Data Eng., 16: 1038-1051.

Maio, D. and D. Maltoni, 1996. A structural approach to fingerprint classification. Proceedings of the 13th International Conference on Pattern Recognition, August 25-29, 1996, IEEE, New York, USA., ISBN:0-8186-7282-X, pp: 578-585.

McKay, B.D. and A. Piperno, 2014. Practical graph isomorphism, II. J. Symbolic Comput., 60: 94-112.

Sander, G., 1999. Graph layout for applications in compiler construction. Theor. Comput. Sci., 217: 175-184.

Santo, M.D., P. Foggia, C. Sansone and M. Vento, 2003. A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recognit. Lett., 24: 1067-1079.

Ullmann, J.R., 1976. An algorithm for subgraph isomorphism. J. ACM., 23: 31-42.

Wale, N.G. and I.A.W. Karypis, 2007. Method for effective virtual screening and scaffold-hopping in chemical compounds. Comput. Syst. Bioinf. Conf., 6: 403-414.

Washio, T. and H. Motoda, 2003. State of the art of graph-based data mining. ACM SIGKDD Explorat. Newslett., 5: 59-68.

Yan, X. and J. Han, 2002. gSpan: Graph-based substructure pattern mining. Proceedings of the 2002 IEEE International Conference on Data Mining, December 9-12, 2002, Maebashi, Japan, pp: 721-724.

Yan, X., P. Yu and J. Han, 2004. Graph indexing: A frequent structure based approach. Proceedings of the Special Interest Group on Management of Data, June 13-18, 2004, Paris, France, pp: 335-346.

Zhou, B. and J. Pei, 2008. Preserving privacy in social networks against neighborhood attacks. Proceedings of the 24th IEEE International Conference on Data Engineering, April 7-12, 2008, Cancun, pp: 506-515.