Fast Canonical Labelling for Graph based Data Mining

D. Kavitha
SrAssistant Professor
PVPSIT
Vijayawada, India

V. Kamakshi Prasad
Professor & Head of the Department
PVPSIT

J. V. R. Murthy
Professor
PVPSIT

Abstract: At the core of any frequent subgraph mining algorithm, Graph/Subgraph Isomorphism is required to determine topological equivalence and frequency of a graph. More specifically in biology, frequent subgraphs are used to derive the internal properties and the underlying biological relational information. The main challenge in the enumeration process is internal symmetry present in biological structure which generates an exhaustive list of permutation checking and effects the efficiency of the algorithm itself. In this work, we present an algorithm to discover isomorphic graphs using canonical labelling which overcomes this problem based on the symmetries present in the graph, which eliminates the permutation of symmetric nodes in the canonical labelling process. The proposed technique reduces computation time by reversing the traditional labelling process which starts from the leaves and by eliminating the backtracking. Node signature and edge signature are used to identify symmetric groups present in a graph. For subgraphs with symmetric and non-symmetric structures, the algorithm reduces search space and computation feasible to perform isomorphism testing on large datasets for frequent pattern mining.

Keywords: graph mining, graph isomorphism, canonical labelling, partition refinement, symmetry

I. INTRODUCTION

Studies on Graph Mining have established many approaches for extracting useful knowledge from a large amount of structured data modelled as graphs[1-3]. As the graph structure is widely used in the fields of bio-informatics, chemical compound analysis, text analysis and video indexing [4-14]. Frequent subgraphs are basic to perform graph mining tasks such as clustering, indexing and classification. The process of identification of frequent patterns from a graph database or from a single large graph is known as frequent subgraph mining [15-18]. There have been popular graph mining algorithms based on graph theory are proposed such as AGM (Apriori-based Graph Mining) [19], FSG (Frequent SubGraph discovery) [20], gSpan (graph-based Substructure pattern mining) [21], and FFSM (Fast Frequent Subgraph Mining) [22]. gSpan and FSG needs a lot of time to solve subgraph isomorphism problem. FFSM converts subgraph isomorphism problem into graph isomorphism problem, but testing of graph isomorphism still need lot of time.

Many approaches have been proposed in the literature to find the isomorphic graphs as it is known NP problem[23], however most interesting approaches are direct approach and computing canonical label of a graph. The algorithms in the direct approach proceed by matching the two graphs using backtracking. These algorithms proceed by using a depth-first backtracking and by using techniques to prune the size of the search space. On the other hand, the algorithms in the second category compute canonical label of each graph and then compare the labels directly. Canonical labelling of a graph consists of assigning a unique label to a graph such that the label is invariant under isomorphism. These two approaches may differ in the way they solve the isomorphism problem, but both make use of invariants to do partitioning and refinement.

A. Contribution

In this paper, an algorithm F-CANONICAL (Fast Canonical Labelling for Graph based Data Mining) is proposed to check the isomorphic graphs in a set of simple labelled graphs based on the canonical label. The proposed algorithm makes use of symmetries present in a graph to improve the overall performance. This algorithm uses invariant properties of nodes to find symmetries present in a graph. Encodes the node invariants node label, node degree, edge label of a node and adjacent nodes to find signature of a node and encodes the invariants of an edge as an edge signature. Node signature and edge signature are used to identify symmetric groups present in a graph.

1. The algorithm starts canonical labelling with higher degree nodes to reduce computational complexity of permutations required to construct canonical label.
2. Backtracking is avoided to identify symmetric groups in the search space.

II. RELATED WORK

Lots of well-known isomorphism test algorithms were developed. J.R. Ullmann [24] 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. Another (sub)graph isomorphism algorithm VF2[25] proposed by Cordell et al. also based on depth-first search (DFS) strategy, employed some feasibility rules that prune unpromising node 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[28], is 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 test isomorphism for the individual pairs of graphs.

Coming to another category of graph isomorphism algorithms that uses canonical label to test the graph isomorphism, In [26,27] Babai and Luks proposed a fast algorithm to compute canonical forms of general graphs in \textit{exp}(n/2+O(1)) time. The most powerful algorithm currently available is McKay’s Nauty package [29,30] 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 to find the canonical code for some categories of graphs[32]. Furthermore, NAUTY does not allow graphs to have labels; hence it is not applicable to labelled graphs.

An alternative methods such as FSG and minimum DFS canonical code are popular in frequent graph/subgraph mining. The FSG combines several types of node invariants to partition the nodes into equivalence classes. If the nodes of a graph with n nodes are partitioned into c classes \(\pi_1, \pi_2, ..., \pi_c\), then the number of different permutations need to be considered in order to find that the canonical code is \(\Pi = 1 \sim c(\pi_i)!\), which is substantially faster than the n! Permutations required by the earlier approaches. Xifeng Yan and Jiawei Han 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[31], i.e., string edit distance, 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.

III. PRELIMINARIES

A graph G is made of nodes V and edges E. A graph is simple, connected and labelled. Each node or edge may not have unique label and a label may be assigned to many nodes or many edges. Formally a graph G is defined as

**Definition 1:** A labelled graph is defined as quadruple \(G = (V, E, L, LF)\), where V is a set of nodes, \(E \subseteq V \times V\) is a set of edges, L is a set of labels, and LF is a function that gives a unique label to each node of G.

Two graphs are said to be isomorphic if they have identical behaviour in terms of graph-theoretic properties. Precisely graph isomorphism can be defined as

**Definition 2:** Given two labelled graphs \(G = (V, E, L, LF)\) and \(G' = (V', E', L', LF')\), we say that G is isomorphic to \(G'\), written \(G \cong G'\), if there exists a bijection \(f: V \rightarrow V'\), called isomorphism, such that

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

A. Problem Formulation

The canonical label of a graph \textit{Canonical(G)} can be defined as a unique label (e.g., string) of a graph that is invariant on the ordering of the nodes and edges in the graph. It can be the smallest or the largest string obtained by concatenating all the node labels followed by the columns of the upper triangular entries of the adjacency matrices over all possible permutations of nodes.

Formally, given a class C of graphs which is closed under isomorphism, a canonical labelling algorithm assigns a unique label to each graph in C, in such a way that two graphs in C are isomorphic if and only if the obtained labels of graphs coincide

**Definition 3:** A canonical labelling of graph G is said to be an isomorphism-invariant labelling of G’s nodes, i.e., two graphs G and G’ have the same canonical labelling if and only if they are isomorphic to each other. A function \(C: G \rightarrow G\) is a canonical form if

(i) \(\forall G \in G : C(G) = G\),
(ii) \(\forall G, G' \in G : G \cong G' \Rightarrow C(G) = C(G')\).

Generally, if a graph has \(|V|\) nodes, the complexity of determining canonical label of a graph is \(O(|V|^2)\). Computational complexity can be reduced by lexicographical ordering of nodes. But, the challenge that arise in the ordering of nodes is symmetry (automorphism) detection. Symmetry is a permutation of the graph's nodes that preserves the graph’s edge relation. To do so, the node invariant properties such as degree, label of node, edge relationships between nodes, label of edges, neighbours to identify symmetry etc. are used and defined in the following definitions. Node invariants are some inherent properties of the nodes that are preserved across isomorphism mappings such as node labels and degrees. However, this local information can then be propagated around the graph.

In order to reduce the number of possible permutations, node invariants are used. Nodes with the same values of the node invariants are partitioned into the same equivalence classes. If the nodes of a graph are partitioned into p classes \(\pi_1, \pi_2, ..., \pi_p\), the amount of all possible codes need to be generated in order to obtain the canonical one is \(O(|\Pi| = 1 \sim p!(\pi_i)!)\) in [15].

**Definition 4:** The nodes of a labelled graph G are partitioned into disjoint non empty sub sets classes denoted by \(\Pi\), an ordered partition \(\Pi(G) = \{\pi_1, \pi_2, ..., \pi_p\}\), satisfies the following properties

(1) For all nodeu, v \(\in \pi_\xi(G)\), \(deg(u) = deg(v)\) and \(lbd(u) = lbd(v)\), \(\forall u, v \in \pi_\xi(G)\), \(1 \leq k \leq p\).
(2) For all nodeu \(\in \pi_\xi(G)\) and v \(\in \pi_\xi(G)\), \(deg(u) > deg(v)\) and \(lbd(u) < lbd(v)\), \(if k < l\).
IV. THE PROPOSED METHOD

In this section, pseudocode to generate canonical label of a graph $G$ is presented. Algorithm starts with the equivalence partitioning of nodes as described in the above definition and continues with the added optimization that refine each class before going to next one to compute the $\text{vrtxlbl}$ as the first step in canonical code construction which is in turn used to construct $\text{edglbl}$ and canonical code.

Algorithm

**Input:** A labelled graph $G$

**Output:** Canonical label of a graph $G$: $\text{Canonical}(G)$

Begin

1. Make ordered partition $\pi(G) \leftarrow (\pi_1, \pi_2, \ldots, \pi_p)$ of $G$
2. for each class $\pi_i$
3. If $|\pi_i| = 1$
4. Append $v \in \pi_i$ to $\text{vrtxlbl}$
5. else if $|\pi_i| > 1$ and $\text{vrtxlbl}$ is null
6. symmetry($\pi_i$)
7. else
8. $\text{refine_class}(\pi_i)$
9. Construct $\text{edglbl}$
10. if ($v_i, v_j) \in E(G)$
11. $\text{edglbl} \leftarrow \text{edgedglbl} \| l(v_i, v_j)$
12. else $\text{edglbl} \leftarrow \text{edglbl} \| 0$
13. canonical $\leftarrow \text{vrtxlbl} \| \text{edglbl}$

end

The first step of computing $\text{Canonical}(G)$ of a graph $G$ is to identify the node label $\text{vrtxlbl}$ that contains labels of nodes as a string in the order of nodes such that the canonical label is lexicographically small/high. In step 1, predefined node invariants, degree and label of the node is used to create an initial ordered partition $\pi(G)$ of the nodes of the graph $G$ into $p$ classes as stated in definition 4. A trivial class is a class of size one. If the ordered partition $\pi(G)$ contains discrete and trivial classes, append the elements of each class to the $\text{vrtxlbl}$ in the order of partitioning done as nodes in different classes of $\pi(G)$ have already been distinguished from each other with the properties as stated in definition 4. In the usual canonical labelling procedure, permutation of nodes is used to get the order of nodes of a non-trivial class $\pi_1$. To avoid the permutations computation of identical nodes inside $p$ classes, to further refine each class and to find the symmetries present in a class, algorithms $\text{refine_class}()$ and $\text{symmetry}()$ are used. Step 2 to step 8 in the algorithm defines this process and is the first step of canonical label construction. This yields a $\text{vrtxlbl}$ string that is enough to construct canonical label of graph without permutations. String obtained by concatenating the columns of the upper triangular entries of the adjacency matrix was referred as $\text{edglbl}$. Canonical label be a string that was obtained by concatenating $\text{vrtxlbl}$ and $\text{edglbl}$. Step 9 to step 13 constructs edge label based on the order of nodes of constructed $\text{vrtxlbl}$ and edge relations between them i.e., second step of labelling. Obtain the canonical label of graph $\text{Canonical}(G)$ by concatenating the string $\text{vrtxlbl}$ followed by the string $\text{edglbl}$.

The main algorithm started by forming an equitable ordered partition of nodes, thereby extracting all the degree and label information. The children of an equitable ordered partition in the search tree do not correspond to all possible splitting of $\pi(G)$. In order to distinguish nodes of $\pi_i$ and to find the symmetry, other graph theoretical information such as edge relations between nodes are exploited using the algorithm $\text{refine_class}()$.

Algorithm: $\text{refine_class}()$

**Input:** A class of graph $\pi_i (G)$

**Output:** Refined class

Begin

for each node $v_a$ in $\pi_i$ do
for each node $v_b$ in $\text{vrtxlbl}$ do
if $(v_a, v_b) \in E$
    $R \leftarrow R \cup v_a$
else
    $\text{UR} \leftarrow \text{UR} \cup v_a$
if $\text{UC}$ is singleton
    $\text{vrtxlbl} \leftarrow \text{vrtxlbl} \cup v \in \text{UR}$
else if $|\text{UR}| > 1$
    $\text{symmetry}(\text{UR})$

end
append(R)
end.

To discard numerous permutation possibilities in each class \( \pi_i \), refine_class() algorithm invokes partition refinement to propagate the constraints of the graph i.e. the graphs node degrees, node labels, edge labels and edge relations, as the edge relations with the nodes in the \( vrtxlbl \) has an influence on the lexicographic order, divide the nodes of a non-singleton class are divided into unrelated(\( UR \)) and related(\( R \)) set of nodes based on the edge relations between the nodes of the class to the nodes in the \( vrtxlbl \) till constructed. Data structures \( UR \) and \( R \) are designed to maintain list of connected and unconnected nodes that are resultant of first level of refinement and make available for further refinement of nodes of each class \( \pi_i \). Set of nodes that have edge relations with the nodes of \( vrtxlbl \) constructed till are stored in set \( R \) and \( UR \) contains that haven’t. Symmetry() and append() checks the symmetries present in the elements of the non-singleton sets \( UR \) and \( R \) to identify the order of nodes.

A. Symmetry detection

While the isomorphism problem is NP-complete, several basic techniques have been developed to minimise the required work. These techniques and how they are customised in \( I-Canonical \) are explained in this section.

The definition 5 describes the definition of symmetry.

**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 \( Sym(G) \). A symmetry group of graph \( G \) is a permutation of \( G \)'s nodes that preserves \( G \)'s edge relation, i.e., \( G = G' \).

Preposition 1: A graph \( G \) is node-transitive if for every node pair \( u, v \in V_G \) there is an automorphism that maps \( u \) to \( v \).

Preposition 2: A graph \( G \) is edge-transitive if for every edge pair \( x, y \in E_G \), there is a symmetry that maps \( x \) to \( y \).

All nodes in the same orbit have the exact same degree, label and neighbours. All edges in the same orbit have the same pair of degrees at their endpoints. 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 explored in this paper is to eliminate the permutation computations involved in canonization. At each stage, the first non-trivial class of \( \pi_i \) is chosen to find symmetries. We record in the search tree not only the current equitable ordered partition, but also the sequence of nodes used for splitting.

**Definition 6:** Nodesig: For a node \( v \) with adjacent nodes \( u_1, u_2, ..., u_m \), \( nodesig(v) \) is a string that contains invariant information of neighbourhood. The neighbours of a node \( v \) of a graph \( g \), denoted as \( S \) and the string \( nodesig(v) \) contains the degree, node and edge labels of neighbours as strings.

(1) \( nodesig(v) \) is \( ndegree,Unlbl,Unelbl, \)
(2) For each \( v \in S \), \( ndegree\), = (\( degree(u_1), degree(u_2), ..., degree(u_m) \)) where \( degree(u_k) \geq degree(u_{k+1}), \) for all \( k \leq \) \( m - 1 \)
(3) For each \( v \in S \), \( nlbl\), = (\( lbl(u_1) \) || \( lbl(u_2) \) || ... || \( lbl(u_m) \)), where \( u_k \) is in the order of \( ndegree \).
(4) For each \( v \in S \), \( nelbl\), = (\( lbl(u_1, v) \) || \( lbl(u_2, v) \) || ... || \( lbl(u_m, v) \)) where \( u_k \) is in the order of \( ndegree \).

Therefore, the \( nodesig \) of a node with \( m \) adjacent nodes is composed of the triples in the form of adjacent nodes degree (\( ndegree \)), adjacent nodes label (\( nlbl \)) and adjacent nodes edge label (\( nelbl \)). These triples are ordered first by degree nodes, then by node labels and then by edge labels. Notice that the invariant of \( nodesig \) not obtained by calculating all possible permutations; it is attained by sorting the degrees of adjacent nodes as described in the above definition 6 rule 1. And then adjacent nodes label and adjacent nodes edge label are obtained just by concatenating the labels in the order of nodes obtained with Rule 1. The intention behind the design of \( nodesig \) is to combine the degree, label and the edge label of adjacent nodes information of a node into a string-based representation. Therefore, the lexicographic order of \( nodesig \) is totally ordered as normal strings.

Preposition 1. Given two \( nodesig(v_1) \) and \( nodesig(v_2) \) of nodes \( v_1 \) and \( v_2 \) of a labelled graph \( G \), respectively, \( v_1 \) is not symmetric to \( v_2 \) if \( nodesig(v_1) \neq nodesig(v_2) \).

Proof: We prove this, by showing that if \( v_1 \) is symmetric to \( v_2 \) then \( nodesig(v_1) = nodesig(v_2) \). Since, \( v_1 \neq v_2 \), there exists an assignment that maps each adjacent node of \( v_1 \) to a node of \( v_2 \), namely \( w_i \) to \( v_2 \). Otherwise the adjacency relations preservations such as adjacent nodes degrees, node labels and edge labels must be violated between these two nodes. Since the lexicographic order of adjacent nodes relationships are totally ordered, and the \( nodesig \) of a node is the string obtained by concatenation, these two \( nodesig \) must be equal. This symmetry property of nodes can be used to avoid permutation when generating canonical label of a graph that have symmetrical nodes.

The identification of symmetrical 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 nodes[31]. The \( symmetry() \) algorithm differs with Nauty in recognising and using symmetries that may appear both at leaf and non-leaf terminals that are inferred from \( nodesig \) of nodes and edge relations i.e. structure of the partition which results same adjacency matrix.

**Algorithm:** symmetry()

**Input:** A set of nodes

**Output:** Symmetric groups

**UR** -> set of nodes in an orbit to discover automorphism

RU -> set of nodes need to refine
As stated in definition 6, for each node computenodesig. Finding this nodesig itself time consuming process. While constructing the adjacency list in this algorithm, adjacent nodes are added to the list in the required sorted order as per definition 6 and hence this algorithm time complexity is reduced. Subsequently the amount of work required to compute symmetric groups is reduced. After finding nodesig, split the nodes into subsets based on symmetries and appendnodes of subsets to vrtxlbl is same as explained in the above algorithms. The algorithm append() is used to append nodes that have edge relation with nodes of vrtxlblAs in the previous algorithm.

Algorithm: Append()
Input: a set of nodes
Output: Append nodes to vrtxlbl otherwise proceed
begin
for each u ∈ Ru do
    if(u in Ru) then
        find edge relation with the nodes of vrtxlbl
        if(no edge relation) then
            append to vrtxlbl
        else
            set status of u_i
            append to Cu
        append(Cu)
    return
end

Node individualization is an important task in a symmetrical group. Append() algorithm substantiates symmetry() algorithm in individualizing nodes in a symmetrical group using the node transitive and edge transitive properties of symmetry. Generally, the individualization of a node corresponds to select a subgroup of permutations in fixing that node which was avoided in this algorithm. What would be the next node in the vrtxlbl is selected based on the local properties of adjacencies of nodes and these are algorithmised in append().

V. EXPERIMENTAL RESULTS

Improved CanonicalLabelling for Graph based Data Mining implements a general purpose algorithm aimed at reducing the search space associated in constructing canonical label with the knowledge of the symmetrical groups of a graph. F-CANONICAL was implemented in C language. The experiments were carried out on a Intel® Pentium® Dual CPU T3400 @2.17 GHz with 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 [33]. These are Randomly Connected graphs that each node pair has a probability of η that characterizes the density of the graphs. A graph with n nodes has n^2 edges, and each node has η connected edges in average. In Fig 1 the results obtained on randomly connected graphs are presented using two distinct values for the parameter η.

Fig1: (a)Randomly Connected Graphs - η = 0.01 (b) Randomly Connected Graphs - η = 0.1
The plots show the average execution time (in seconds) as a function of the number of nodes. By observing the results of fig 1(a) 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 show the advantages of incorporating the symmetric information in the search on the real datasets, we used the data sets in a standard graph library available at[34]. 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 nodes types and four types of edges. The largest graph has 214 nodes and 214 edges, on an average 43 nodes 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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{(a) Yeast (b) MCF-7}
\end{figure}

Aviation (ailab.wsu.edu/subdue). This dataset contains a list of records extracted from the aviation safety reporting system database [35]. Each record corresponds to an event and information is represented by a graph having two types of 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 of 100K nodes and 133K edges. Note that Aviation is a fundamentally different dataset when compared with the previous ones. The Aviation graph has on average one edge per node, thus, it is very sparse. Also it has a very large number of distinct node labels. Directed edges are converted into undirected and Experiments are conducted on randomly sampled classes of graph and results are plotted in Figure 3.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{Canonical label computation on aviation data}
\end{figure}

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 the proposed algorithm on set of graphs, the experiments are conducted on MCF-7 database that has nearly 40K graphs. The graphs are randomly chosen from the active and inactive sets. The time required to compute canonical label of set of graphs on MCF-7 database graphs is shown in figure 4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{(a) Active (b) Inactive}
\end{figure}
The compound datasets used in the experiments are useful in characterizing the effectiveness of heuristics to find symmetry and thus to eliminate permutations in canonical label computation.

![Graphs](image-url)

(a) Randomly Connected Graphs - $\eta = 0.01$

(b) Randomly Connected Graphs - $\eta = 0.1$

(c) Active

(d) Inactive

Fig 5: Performance evaluation for testing isomorphism

The average time required to perform isomorphism testing on pairs of randomly connected graphs is shown in figure 5(a) 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 figure 5(c) and (d). From the results of above figure, 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.

### VI. CONCLUSION

In this paper, we present F-CANONICAL --Fast Canonical Labelling for Graph based Data Mining based on the symmetries present in a graph. The algorithm considers all symmetries present in a graph as it is an important tool in pruning the search space. By identifying the symmetric groups, nodes relative and nodes unrelated, we have reduced the action of permutation on combinatorial nodes and total order of nodes to find canonical label. 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 symmetric groups. The factors that influencing the performance of the algorithm are the degree, relativity and complexity of symmetry in the subgraph. The algorithm can also be applied to unlabelled graphs as considering the label of all nodes 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

[28] Conauto-2.0: Fast isomorphism testing and automorphism group computation
[34] https://www.cs.ucsb.edu/~xyan/dataset.htm
[35] SUBDUE databases; http://ailab.wsu.edu/subdue/