Mining Rich Graphs: A Graph Transformation Approach

Didi Surian* David Lo† Hong Cheng‡ Ee-Peng Lim§

Abstract

There have been a large number of studies mining graph patterns, however, most of them only process graphs with nodes and edges having a *single label*. Often there is a need to represent a rich dataset as graphs with nodes and edges having *multiple labels* - each label representing a feature in the dataset. For example, in a social network, multiple features could be associated to individuals (e.g., age, address, etc.) and their relationships (e.g., friend, foe, send message, etc.). To analyze these rich datasets, there is a need to extend existing graph mining algorithms to also mine *rich* graphs with nodes and edges having multiple labels.

In this paper, we propose a novel algorithm and framework to transform richly labeled graphs (i.e., graphs with nodes and edges having multiple labels) to an *equivalent* set of simple labeled graphs (i.e., graphs with nodes and edges having single labels). The resultant simple graphs could be fed to most of the available graph mining algorithms to produce simple graph patterns. A reverse translation process is then employed to recover rich graph patterns from the simple graph patterns. We demonstrate that our proposed algorithm is scalable on various synthetic and real datasets. We experiment with three notable graph mining algorithms which are gSpan, CloseGraph, and Top-k LEAP algorithms. We show that our algorithm and framework could complement existing simple graph mining algorithms to allow them to mine rich graphs.

1 Introduction

A graph could be used to represent and model many complicated relationships in a dataset. Due to the generality of a graph as a data structure, there have been many studies that investigate the discovery of knowledge from graph datasets. There have been many algorithms developed to mine frequent subgraphs [20, 15, 16], correlated subgraphs [13], discriminative subgraphs [19], etc. However all the above mentioned studies only mine patterns from graphs with each node and edge having a *single label*, called simple graphs.

In various domains, there is often a need to model data as graphs with nodes and edges having *multiple labels*. Such graphs are called richly labeled graphs or rich graphs. For example, in a social network, several features could be attached to users and their relationships. Graphs representing a social network might consist of nodes corresponding to users, and edges corresponding to relationships among them. Several labels could be attached to each node and edge to add more information about the network. For example, we could label a node with multiple pieces of information such as the length of time a user has joined the network, age, gender, etc. We could also label an edge with information about the number of friends two users have in common, the length of time two users have been friends, etc. As another example, to model collaborations among software developers, we could use a graph with developers as nodes and collaborations between two developers as edges. The nodes may be labeled with the number of past and ongoing projects a developer has, the number of his projects which are successful, etc. The edges may also be labeled with information such as the number of past projects two developers have worked together, the length of time two developers have collaborated, etc. There are many other situations where we need to use *multiple labels* instead of *single labels* to enrich the graph representation of a rich dataset.

Thus, there is a need to enable past algorithms that only mine simple graphs to also mine rich graphs. We fill this need in this work. We propose a generic framework that transforms an existing algorithm that mines a simple graph dataset to handle a rich graph dataset. Our framework treats an existing algorithm as a black box and thus does not require any change to be made to it. This makes our framework desirable, as existing algorithms could be extended without the need to "reinvent the wheel". Many existing implementations

*School of Information Technologies, University of Sydney, Email: dsur5833@uni.sydney.edu.au (The work was done while the author was with School of Information Systems, Singapore Management University)

†School of Information Systems, Singapore Management University. Email: davidlo@smu.edu.sg

‡Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. Email: hcheng@se.cuhk.edu.hk

§School of Information Systems, Singapore Management University. Email: eplim@smu.edu.sg

of popular graph mining algorithms could also be reused to handle rich graphs.

We focus on a family of graph pattern mining algorithms that take as input a graph database and compute the set of graphs in the database where a pattern occurs based on subgraph isomorphism. Some examples of concrete pattern mining algorithms in this family are: frequent subgraph mining algorithms (e.g., [21]), closed subgraph mining algorithms (e.g., [22]), discriminative subgraph mining algorithms (e.g., [5]), etc. We refer to this family of algorithms as *subgraph pattern mining (SPM)* algorithms. In this paper, unless otherwise stated, whenever we mention pattern mining algorithms we refer to this family of algorithms.

Our approach would first translate a richly labeled graph dataset into an *equivalent* simple labeled graph dataset. Many existing graph mining algorithms could then be run on the resultant translated dataset. Running these algorithms would produce a set of simple graph patterns (i.e., subgraph patterns with nodes and edges having single labels). We then provide a reverse translation process to convert these simple graph patterns back to rich graph patterns. We provide several properties and theorems that guarantee our approach is both sound and complete, i.e., our proposed approach does not miss any patterns or add extraneous patterns.

To show the scalability of our approach, we experiment on several synthetic and real rich graph datasets. Real datasets are taken from a mobile-phone-based social network and an open source software development portal. We experiment with three existing pattern mining algorithms, i.e., gSpan [21], CloseGraph [22], and Top-k LEAP [5]. Our goal is to enable these algorithms to mine rich graph patterns from rich graphs.

Our contributions are as follows:

1. We propose a novel framework to enable existing graph mining algorithms to handle rich graphs.
2. We propose a translation algorithm to map a rich graph dataset to an equivalent simple graph dataset. We also introduce a reverse translation algorithm to recover rich graph patterns from the translated simple graph patterns.
3. We prove that our proposed approach, which involves translation and reverse translation processes, is sound and complete.
4. We show that we could successfully integrate our framework with three notable graph mining algorithms, namely, gSpan, CloseGraph, and Top-k LEAP.
5. We show the scalability of our approach by using several synthetic and real rich graph datasets.

The structure of this paper is as follows. In Section 2, we discuss related work. Section 3 describes some concepts and definitions. Section 4 describes our proposed approach to extend existing graph mining solutions to mine richly labeled graph datasets. Section 5 describes our experiments. We conclude our work and describe future work in Section 6.

2 Related Work

There have been many studies that mine frequent patterns from graphs. These studies could be categorized into two: those that analyze a graph database, and those that mine patterns from a single large graph. In the following paragraphs, we would highlight some algorithms that fall into each category. We would also highlight some other algorithms that mine discriminative subgraph patterns.

Graph Database Setting. One of the first pioneers that mined frequent patterns from a graph database is Dehaspe *et al.* [6]. They used inductive logic programming to mine frequent graph substructures. These substructures are used to predict the carcinogenicity of a chemical compound. Their study is extended by Inokuchi *et al.* that proposed an apriori-based approach to mine frequent subgraphs from a graph database [11]. There have been many studies that improve the scalability of frequent subgraph mining algorithms. These include the work by Kuramochi and Karypis which proposes an algorithm named FSG [15], and the work by Yan and Han which proposes an algorithm named gSpan [20]. There are many other algorithms including: MoFa [3], FFSM [10], SPIN [17], and Gaston [16].

There has also been an interest to mine a compact representation of frequent subgraphs. Yan and Han mined *closed* frequent subgraph patterns [22]. Zeng *et al.* mined frequent subgraph *generators* [23]. Closed patterns are the largest patterns in an equivalence class of patterns supported by the same set of graphs in an input database. Generators are the smallest patterns in the equivalence class.

Our work is orthogonal to the above studies which mine patterns from simple graphs, i.e., graphs where each node and edge is attached with at most a single label. We propose a framework to enable frequent subgraph mining algorithms to handle rich graphs, i.e., graphs where each node and edge can be attached with multiple labels. There are many problem settings that are best modeled as rich graphs.

Single Graph Setting. There are many algorithms that mine patterns that appear frequently in a single large graph. In this setting, multiple embeddings of a pattern in a single graph need to be taken into consideration. One of the first pioneers that mine for patterns from a single graph is Holder *et al.* [9]. They

mined for approximate patterns based on minimum description length property. There are a number of other studies that also mine for frequent patterns in a single graph settings. These include the work by Vanetik *et al.* [18], Ghazizadeh and Chawathe [7], Kuramochi and Karypis [15], and Zhu *et al.* [24].

Different from the above studies, in this work we focus on extending graph mining algorithms that work on the graph database setting to mine from a database of rich graphs.

Mining Discriminative Subgraph Patterns. Yan *et al.* proposed a direct discriminative graph mining approach called LEAP [19]. The approach is later extended by Cheng *et al.* in [5] to mine for top-K most discriminative graph patterns. These patterns are applied in the software engineering domain to localize software bugs. Jin and Wang proposed a technique that leverages search history to improve the scalability of discriminative subgraph mining [12].

This work is orthogonal to the above studies. Our work enables the above algorithms to process rich graphs and mine rich graph patterns.

3 Preliminaries

We introduce the notions of simple graph and rich graph in Definitions 3.1 & 3.2 respectively. The example for a simple graph and rich graph is shown in Figure 1.

DEFINITION 3.1. (**Simple Graph**) *A simple graph is a set of nodes N, edges E, and labels L. Each node and edge could be attached with* one *label from L. Each edge (u,v) is a pair of nodes in N. Each pair of nodes could be linked only by one edge. Given a node n we denote edges incident to it and its label by $n.Edges$ and $n.Label$ respectively. We denote the label of an edge e by $e.Label$. Given an edge e of a node n, we refer to the other node connected to n by e as $e.Target$.*

DEFINITION 3.2. (**Rich Graph**) *A rich graph is a set of nodes N, node placeholders NP, edges E, edge placeholders NE, and labels L. Each node and edge contains* multiple *placeholders each of which contains one label from L. Each edge (u,v) is a pair of nodes in N. Each pair of nodes could be linked by one edge. Given a placeholder np, we denote its labels by $np.Label$. Given a node n we denote edges incident to it and labels contained in its placeholders by $n.Edges$ and $n.Labels$ respectively. We denote the labels of an edge e contained in its placeholders by $e.Label$. Given an edge e of a node n, we refer to the other node connected to n by e as $e.Target$.*

We next formally define sub-graph isomorphism for simple and rich graphs in Definitions 3.3 & 3.4

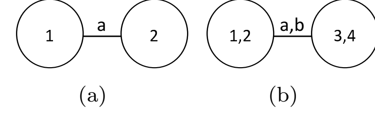

Figure 1: Example: (a) Simple Graph (b) Rich Graph

respectively. Simply put, a graph is a subgraph of another if we could add nodes, edges, and labels to the earlier (without modifying or deleting any nodes, edges, and labels) to form the latter. We give an example of sub-graph isomorphism for simple and rich graphs in Figures 2 and 3 respectively. Corresponding nodes of graphs (a) & (b) are shaded in gray color.

DEFINITION 3.3. (**Subgraph Isomorphism: Simple**) *For two graphs $G = (N, E, L)$ and $G' = (N', E', L')$, a* subgraph isomorphism *is an injective function $f : N \rightarrow N'$, s.t., (1), $\forall n \in N, n.Label = f(n).Label$; and (2), $\forall (u, v) \in E, (f(u), f(v)) \in E'$ and $(u, v).Label = (f(u), f(v)).Label$. f is called an embedding of G in G'.*

DEFINITION 3.4. (**Subgraph Isomorphism: Rich**) *For two graphs $G = (N, NP, E, NE, L)$ and $G' = (N', NP', E', NE', L')$, a* subgraph isomorphism *is an injective function $f : NP \rightarrow NP'$, s.t., (1), $\forall np \in NP, np.Label = f(np).Label$; and (2), it induces a function $g : N \rightarrow N'$ where, (a) $\forall n \in N.\ \forall p \in n.\ \exists p' \in g(n).\ f(p) = p'$, (b) $\forall (u, v) \in E, (g(u), g(v)) \in E'$, and (c) $(u, v).Labels \subseteq (g(u), g(v)).Labels$. f is called an embedding of G in G'.*

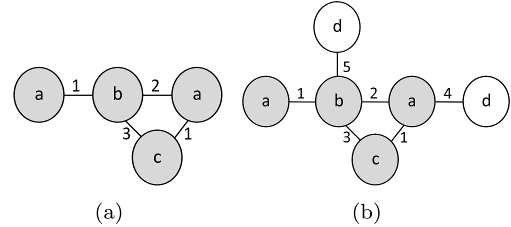

Figure 2: Subgraph Isomorphism: Simple Graphs

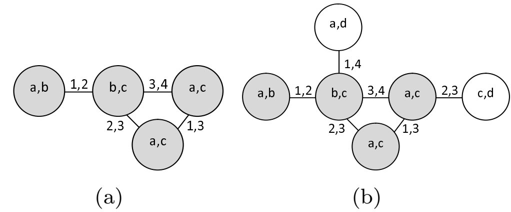

Figure 3: Subgraph Isomorphism: Rich Graphs

4 Proposed Approach

In this section, we first outline our overall framework. We then describe our approach for the translation

from rich graphs to simple graphs. We show that the translation to simple graphs only increases the size of the rich subgraphs by a factor linear to the maximum number of labels per node and the maximum number of labels per edge.

4.1 Overall Framework. We build our approach to accommodate the existing pattern mining algorithms to handle rich graphs. Our framework consists of three steps, which are as follows:

Step 1: Convert rich graphs to simple graphs representation DB_REP.

Step 2: Input these simple graphs to pattern mining algorithms. In this paper we use three pattern mining algorithms, which are gSpan, CloseGraph, and Top-k LEAP. The pattern mining algorithms then mine various patterns.

Step 3: Convert the mined simple graph patterns to its corresponding rich graph patterns.

The steps in our approach are shown in Figure 4.

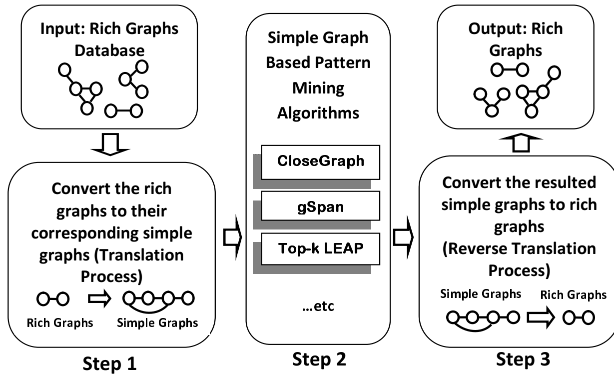

Figure 4: Overall Approach

The next sub-section describes our translation process. A reverse translation process is presented next. We then state some properties and theorems that guarantee the correctness of our translation. Finally, we analyze the bounds on the size of the resultant graph after translation.

4.1.1 Translation Process. A rich graph can have multiple node and edge labels. Our translation process converts a rich graph into a simple graph by performing node and edge *replication* operations. Each replica only retains *one* of the potentially many labels of the original node or edge that it replicates.

We first introduce the notions of translated simple graph and set of sibling nodes in Definitions 4.1 and 4.2 respectively. In a translated simple graph, the replicas of the same rich graph's node (i.e., its *siblings*) are connected together with a special edge.

DEFINITION 4.1. (**Translated Simple Graph**) *A translated simple graph is a regular simple graph with a special edge named sibling-replicated edge (SRE). This edge connects all replicated nodes that are originating from the same node in the original rich graph.*

DEFINITION 4.2. (**Set of Sibling Nodes (SSN)**) *Consider a translated simple graph* $G = (N', E', L)$. *The set of sibling replicated nodes of G is* $\{SNodes \mid \forall n1', n2' \in SNodes, (n1', n2') \in E' \wedge (n1', n2').Label = SRE\}$. *We denote this set as G.SSN.*

Nodes and edges are replicated due to two reasons:

1. There are multiple node labels in a node of a rich graph. This node would be split into multiple simple graph nodes each with a single label.
2. There are multiple edge labels attached to an edge of a rich graph. As a simple graph does not allow for two edges between two nodes, either one of the nodes connected by it would need to be replicated.

We refer to the replicas created by the first reason as *NL-Replicas*. We refer to the ones created due to the latter reason as *EL-Replicas*. Our translation process first creates NL-Replicas. EL-Replicas are constructed next.

Creating NL-Replicas. To create NL-Replicas from a rich graph, for each node, we split it according to the number of labels that it has. The edges of the original node are transferred to each of its replicas. We also add *SRE* edges to connect all the nodes originating from the same rich node. The original rich nodes are then removed from the original graph. After all NL-Replicas have been created and rich nodes removed, all nodes in the graph would each have a single node label. The introduction of new edges ensures that the structures expressed in the rich graph are preserved after the introduction of NL-Replicas.

Figure 5 shows the pseudo-code realizing this. We overlay the NL-Replicas on top of the original graph. We create the NL-Replicas one by one and eliminate the original nodes and edges step-by-step. Note that the order of which the nodes are being processed would not affect the NL-Replicas introduced. If two rich nodes $n1$ and $n2$ are connected, each of the NL-replicas of $n1$ is connected to all the NL-replicas of $n2$.

We illustrate the NL-Replicas creation process in Figure 6.

Creating EL-Replicas. To create EL-Replicas from a rich graph, for each edge with multiple labels, we replicate *one of the two nodes* connected by it. Given a node n, connected to a multi-labeled edge e, we create

Procedure CreateNLReplicas
Inputs:
$G = (N, NP, E, EP, L)$: A rich graph with the set of nodes N, node placeholders NP, edges E, edge placeholders EP, and labels L
Output: G with nodes replaced with NL-Replicas
Method:
1: Let $Orig_N$ = Shallow copy of N
2: For each $n \in Orig_N$
3: Let $NSet[]$ = Create a node array of size $|n.Labels|$
4: For every ith node in $NSet$
5: Let $NSet[i].Label = n.Labels[i]$
6: For every edge e in $n.Edges$
7: Add an edge with labels $e.Labels$ from $NSet[i]$ to $e.Target$
8: Remove all edges from n
9: For every node n_{new} in $NSet[]$
10: Add n_{new} to N
11: Output (N, NP, E, EP, L)

Figure 5: Creation of NL-Replicas

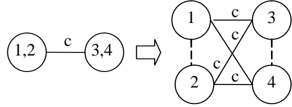

Figure 6: NL-Replicas: Illustration. Edges shown in dashed lines are SREs.

EL-Replicas, by duplicating the node according to the number of labels e has. The original NL-Replica node would be connected with an edge with one of e's labels. Each of the newly introduced EL-Replica nodes would be connected with a new edge with one of the remaining e's labels. The newly introduced EL-Replica nodes are connected to the other replica nodes of the same original rich node by SREs. After all the EL-Replicas are created, the resultant graph would be a simple translated graph composed of NL-Replicas (nodes and edges) and EL-Replicas (nodes and edges).

The remaining ambiguity is on which of the two nodes should be replicated. We use the label of the two nodes to decide. Due to the creation of NL-Replicas, the nodes would have single labels. For a multi-labeled edge connecting two nodes $n1$ and $n2$, there are 3 cases:

1 If $n1$.Label $<$ $n2$.Label, we would create EL-Replicas of $n1$.
2 If $n1$.Label $>$ $n2$.Label, we would create EL-Replicas of $n2$.
3 If $n1$.Label $=$ $n2$.Label, we would create EL-Replicas for both $n1$ and $n2$.

The above cases are used to ensure that the same EL-Replicas are introduced no matter which edges are processed first.

Procedure CreateELReplicas
Inputs:
$G = (N, NP, E, NE, L)$: A graph with NL-Replicas and rich edges
Output: An equivalent translated simple graph with NL- & EL-Replicas
Method:
1: Let $Orig_E$ = Shallow copy of E
2: For each $e \in Orig_E$
3: Let OTR = The node(s) conn. by e to be replicated
4: For every node n of OTR
5: Let $NSet$ = Replicate n, $|e.Labels|$ times
6: For every node n' in $NSet$
7: For every label l in $e.Labels$
8: Add an edge from n' to $e.Target$ with label l
9: For every node n_{new} in $NSet[]$
10: Add n_{new} to N
11: Output (N, E, L)

Figure 7: Creation of EL-Replicas

Figure 7 shows the pseudo-code realizing this. We illustrate the EL-Replicas creation process in Figure 8.

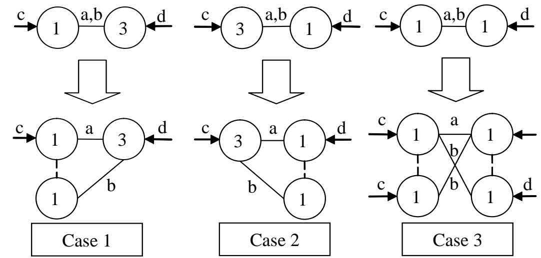

Figure 8: EL-Replicas: Illustration.

Given a rich graph g, its translated simple graph is denoted as TL(g). Also, given a set of rich graphs DB, we denote the corresponding set of translated simple graphs as TL(DB). In the implementation, we combine the NL-Replicas and EL-Replicas creation process so that only one pass through the nodes in the graphs is needed. An end-to-end example of how a rich graph is translated into a simple translated graph is shown in Figure 9.

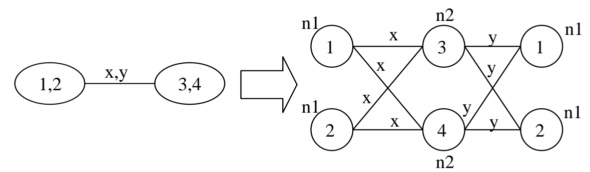

Figure 9: Translation Process

4.1.2 Reverse Translation Process. The reverse translation operation is straightforward. We just need to merge every node connected by SREs together.

These nodes map to the same original rich node. When we merge the nodes we take the union of their node labels. Due to the merging of the nodes, two nodes might have more than one edge connecting them. We would then merge the edges too by again taking the union of their edge labels. Note that as the set union operation is commutative, associative, and distributive, it does not matter as to which nodes and edges are merged first. The reverse translation operation is deterministic: Given one input graph, it would always produce one output graph no matter what nodes and edges are merged first.

We denote the reverse translation operation by RTL. Given a translated simple graph g, the corresponding rich graph after the reverse translation operation is performed is denoted by RTL(g). Also, given a set of translated simple graphs DB, we denote the corresponding set of rich graphs after the reverse translation operations are performed as RTL(DB).

Pattern mining algorithms may produce simple graphs that refer to the same rich graph. This case could happen because the mined patterns may contain several extraneous nodes connected by SREs. An example of this case is shown in Figure 10. As shown in Figure 10, the translated rich graphs in (a) and (b) are isomorphic. To address this issue, we also include graph isomorphism testing in our reverse translation process. We employ the graph isomorphism testing used by Zhu *et al.* [25] and Kim *et al.* [14]. The original algorithms in [14] and [25] are designed to handle subgraph isomorphism testing. In our implementation, we modify their implementations to handle graph isomorphism testing. We perform graph isomorphism testing after the reverse translation operation.

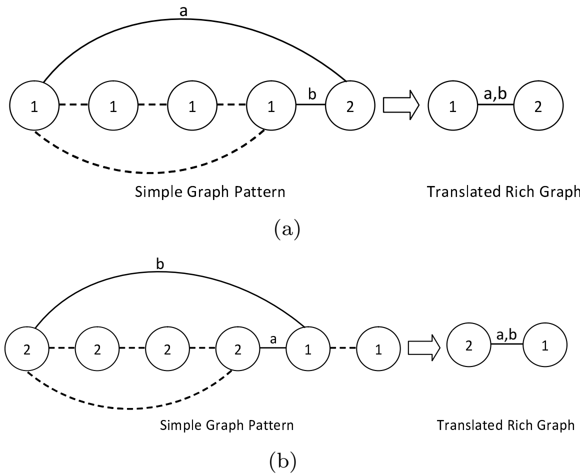

Figure 10: Isomorphic Reverse Translated Rich Graphs

Zhu *et al.* [25] used node signatures to find subgraph isomorphism mappings in large graphs. Nodes signature consists of node label, node degree, and neighbor information. Kim *et al.* [14] extent Zhu *et al.*'s work by considering not just node signatures, but also edge signatures. The signatures for an edge contain the labels of its endpoint nodes, the sum of their degrees, and their neighbors information. In this work, instead of isomorphic subgraphs checking, we modify and extend Kim *et al.*'s approach to check isomorphic rich graphs. Two rich graphs are isomorphic if they have the same number of nodes, the same number of edges, and there is a one-to-one mapping such as corresponding nodes have the same node and edge signatures. Definitions 4.3 gives a formal definition about isomorphic rich graph and Figure 11 shows an example of two rich graphs which are isomorphic.

Definition 4.3. (**Isomorphic Rich Graph**) *Two rich graphs* $G = (N, NP, E, EP, L)$ *and* $G' = (N', NP', E', EP', L')$ *are isomorphic iff (1),* $|N| = |N'|$ *(2),* $|E| = |E'|$ *(3)* $\forall n \in N, n.Signatures = f(n).Signatures$ *(4)* $\forall (u,v) \in E, (f(u), f(v)) \in E' and (u,v).Signatures = (f(u), f(v)).Signatures$*, where* f *is the one-to-one mapping between corresponding nodes in* G *and* G'*.*

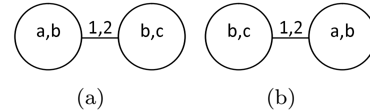

Figure 11: Rich Graph Isomorphism

Node Signatures Matching. Given two nodes u and v, we check their signatures by considering the following conditions:

1. u.Labels = v.Labels
2. u.Degree = v.Degree
3. u.nInfo = v.nInfo

where $u.Labels$ and $v.Labels$ are the labels of u and v respectively, $u.Degree$ and $v.Degree$ are the degrees of u and v. Zhu *et al.* specified a radius parameter r_{max}. For all integer r, where $0 < r < r_{max}$, we count the number of distinct length r simple paths which starts from u and ends at v with label l. In this work, we use the same setting as in [25] where r_{max} is set to 2. Table 1 shows an example of neighbor information for each node of the graph in Figure 12. As shown in Table 1, each node has triples to describe its structural neighbors. For each triple, the first is the number of hops/radius considered, the second is a node label, and the third represents the number of paths ending in a node with the specified label considering the specified number of hops/radius.

Definition 4.4 defines $nInfo$ which is adapted from the definition of $nIndex$ in [25].

DEFINITION 4.4. (**nInfo**) *Given a node v, a maximum radius value r_{max}, and a set of labels L, the nInfo signature of v is $nInfo(v) = \{(r,l,count_{r,l}(v)) \mid 0 < r \leq r_{max} \wedge l \in L\}$. $count_{r,l}(v)$ is the number of length r distinct simple paths connecting node v and the other node with label l.*

Label distribution of distinct paths could partially reflect the neighboring structural characteristic. For example, the signatures of the graph in Figure 12 will change significantly if we add just one edge between nodes v_2 and v_3.

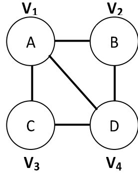

Figure 12: Graph Example

Node	Neighbor Information
v_1	$(1, B, 1), (1, C, 1), (1, D, 1), (2, B, 1), (2, C, 1), (2, D, 2)$
v_2	$(1, A, 1), (1, D, 1), (2, A, 1), (2, C, 1), (2, C, 2), (2, D, 1)$
v_3	$(1, A, 1), (1, D, 1), (2, A, 1), (2, B, 2), (2, D, 1)$
v_4	$(1, A, 1), (1, B, 1), (1, C, 1), (2, A, 2), (2, B, 1), (2, C, 1)$

Table 1: Neighbor Information for Each Node of the Example Graph

Edge Signatures Matching. Given two edges (u, v) and (u', v'), we check their signatures by considering the following conditions:

1. $u.Labels = u'.Labels \wedge v.Labels = v'.Labels$
2. $u.Degree + v.Degree = u'.Degree + v'.Degree$

4.1.3 Properties and Theorems

We first state a property on the canonicity of the translation and reverse translation process in Lemma 4.1.

LEMMA 4.1. (**Canonical**) *One rich graph can only be translated to one translated simple graph. One translated simple graph can only be reverse translated to one rich graph.*

Proof. Note that our translation process is agnostic to the order the nodes and edges are processed. Thus there is only one possible translated simple graph for a rich graph. Similarly, our reverse translation process is agnostic to the order the nodes and edges are merged. Thus there is also one possible rich graph for a translated simple graph. □

Lemma 4.2 states that the translation operation preserves the subgraph relationship between a graph and its subgraph.

LEMMA 4.2. (**Containment**) *Consider an arbitrary rich graph RG, and its subgraph SGP. SGP is a subgraph of RG* if and only if *$TL(SGP)$ is a subgraph of $TL(RG)$.*

Proof. Forward direction: RG has the nodes, edges, and labels of SGP and more. By construction, when processing RG the translation operation would create the NL-replicas of SGP, the EL-replicas of SGP, and more. We could thus add nodes and edges to $TL(SGP)$ for it to form $TL(RG)$. By definition, a graph g is a subgraph of g' iff we could add nodes or edges or labels to it to form g'. Thus, if SGP is a subgraph of RG, then $TL(SGP)$ is a subgraph of $TL(RG)$.

Backward Direction: Assume for contradiction that $TL(SGP)$ is a subgraph of $TL(RG)$, but SGP is not a subgraph of RG. Since $TL(SGP)$ is a subgraph of $TL(RG)$, there are nodes in $TL(RG)$ that could be mapped to nodes in $TL(SGP)$. Let us perform a partial reverse translation on these nodes. Doing this on $TL(RG)$ would create the rich graph SGP connected with the remaining simple nodes and edges. We could continue to perform reverse translation operations to result in a graph SGP'.

Note that for this case, when we perform reverse translation operations, we either: (1) do not affect SGP, (2) add placeholders and edges to nodes in SGP, or (3) merge two nodes in SGP. Let's denote the successive SGPs created by performing these operations as $SGP_1 \ldots SGP_n$. For cases (1) and (2), it is clear that SGP_i is a subgraph of SGP_{i+1}. For case (3), SGP_i is still a subgraph of SGP_{i+1}. Let's consider the case where we merge nodes $n1$ and $n2$ to form node m. We can find the embedding of SPG_i in SPG_{i+1} where the placeholders in unmerged nodes of SPG_i are mapped to the placeholders in the corresponding nodes in SGP_{i+1}, and placeholders in $n1$ and $n2$ are mapped to the corresponding placeholders in m. Merging nodes $n1$ and $n2$ into m simply replaces edge $(x, n1)$ or $(x, n2)$ into (x, m) for all arbitrary node x. These would satisfy the subgraph isomorphism conditions in Definition 3.4.

Thus SGP is a subgraph of SGP'. Since, RG is not a supergraph of SGP, SGP' and RG must be two different rich graphs. This is a contradiction as a translated simple graph could only be reverse translated to a unique rich graph. Thus, the backward direction holds; if $TL(SGP)$ is a subgraph of $TL(RG)$, then SGP is a subgraph of RG. □

Theorem 1 assures the correctness of our approach which involves translation and reverse translation processes. We start by translating rich graphs to simple graphs, and then perform mining operation using a specific *subgraph pattern mining (SPM)* algorithm, and fi-

nally reverse translate the mined patterns to rich patterns. The whole process is sound and complete: all significant rich patterns are mined, and all mined patterns are significant, where significant patterns are defined based on the pattern mining algorithm employed.

THEOREM 1. (**Sound & Complete**) *Consider an arbitrary* subgraph pattern mining (SPM) *algorithm* PMA. *It must be the case that the set of rich graph patterns deemed significant by* PMA *is the set:*

$$RTL(PMA(TL(DB_{rich})))$$

Proof. By definition (see Section 1), all *subgraph pattern mining* algorithms analyze a graph database and compute the set of graphs in the database that contains a pattern based on subgraph isomorphism relation. Following Lemma 4.2, for an arbitrary pattern P and an arbitrary graph g in DB_{rich}, P is a subgraph of g *if and only if* $TL(P)$ is a subgraph of $TL(g)$. This guarantees that if a rich pattern is deemed significant by PMA (it is a subgraph of a significant set of graphs in DB_{rich}), it will also be in the set $RTL(PMA(TL(DB_{rich})))$. Similarly, if a graph is in the set $RTL(PMA(TL(DB_{rich})))$ it will also be deemed significant by PMA (it is a subgraph of a significant set of graphs in DB_{rich}). □

4.2 Analysis. Consider an arbitrary node n with the maximum number of labels. Let e be the edge with the most labels in n. For this n, at most $n.Labels \times e.Labels$ new nodes are introduced. Thus, the number of nodes in the new translated graph grows linearly to the maximum number of labels per node and the maximum number of labels per edge.

We give an example for a translation that involves a two-node rich graph with three labels per node and per edge respectively. Figure 13 shows the original rich graph. Figure 14 shows that after the translation process, the translated rich graph contains twelve nodes. In terms of the number of nodes, the translated graph becomes six times larger than the original one, but less than the number of node labels multiplied by the number of edge labels, which is in this case, 3 x 3 = 9.

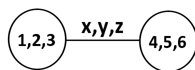

Figure 13: 3-Label Rich Graph

5 Experiments

In this section, we describe our experimental settings, datasets, and results.

5.1 Experimental Settings. The translation and reverse translation experiments are performed on an

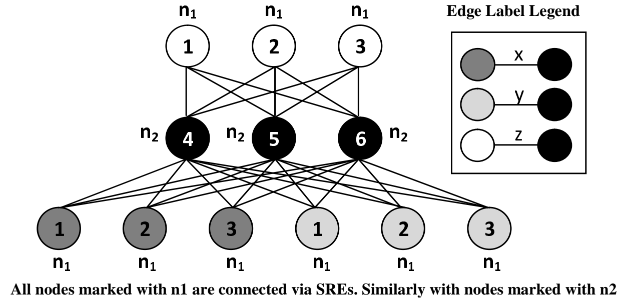

Figure 14: The Translated 3-Label Rich Graph

Intel Core 2 Duo 2.00 GHz Laptop with 1.75 GB of RAM. Algorithms are written in Visual C#.Net on Windows XP Service Pack 3. CloseGraph and gSpan experiments are performed on Intel Xeon Dell X540 Server with 24 GB RAM and 64-bit CentOS Linux release 5.5. Top-k LEAP experiment is run on a Dell PowerEdge R900 server with 2.67GHz six-core CPU and 8GB main memory running RedHat Linux.

5.1.1 CloseGraph & gSpan Algorithms. gSpan [21] is designed to mine frequent patterns from a graph database, while CloseGraph [22] is designed to mine closed frequent graph patterns from a graph database. For these two pattern mining algorithms, we use synthetic datasets and real data extracted from myGamma mobile social network dataset [1].

Synthetic Datasets. We develop a synthetic data generator to generate synthetic datasets with various variables. Our synthetic data generator takes in several user-defined parameters, i.e. G (the number of graphs to be generated), $NODE_{max}$ (the maximum number of nodes in a graph), $EDGE_{max}$ (the maximum number of edges per node in a graph), and $LABEL_{max}$ (the maximum number of labels per node/edge). Based on the above mentioned parameters, our data generator would then generate rich graphs. We generate two different synthetic datasets, i.e., 10,000 ($S10$) and 20,000 ($S20$) rich graphs with $NODE_{max}$=5, $EDGE_{max}$=4, and $LABEL_{max}$=3.

myGamma Dataset. We use myGamma user's friendships information to create one *egocentric graph* for each user. We select the user's top 10 most recently contacted friends to form the graph. A node represents the user and an edge represents a relationship between the user and his friend. We form 10,000 *egocentric graph* taken from 5,000 most active users and 5,000 least active users[1]. The nodes of the graphs are then labeled with information on: (a) the user's length of member-

[1] We use the number of messages a user has sent to his friends as his activity score.

ship, and (b) the number of friends the user has. The edges between users are labeled with: (a) the difference in the users length of membership, and (b) the number of common friends. We discretize each value for the node and edge labels into 4-5 ranges. Thus in total we have 10,000 rich graphs with 2 labels attached to every node and edge. These 10,000 rich graphs have an average size of 4.27 nodes and 3.27 edges, with maximum number of nodes and edges of 11 and 10 respectively.

5.1.2 Top-k LEAP Algorithm.

Top-k LEAP [5] algorithm is designed to mine top-k discriminative subgraphs and requires two sets of contrasting graph databases as its input. For this purpose, we analyze SourceForge.Net, the largest open source software development portal.

SourceForge Dataset. We use the database dumps of SourceForge.Net collected by Madey *et al.* [2]. We categorize projects in SourceForge.Net into two groups, i.e. successful projects and failed projects. From February 2005 onwards, Madey *et al.* collect SourceForge.Net database dumps monthly. We take 64 snapshots which are the dumps for the period starting from February 2005 until May 2010. Each snapshot has many tables and we focus on those containing the information on the various projects hosted in SourceForge.Net and the developers that work on those projects.

We initiate our experiment by extracting projects that have at least one developer from May 2010 snapshot. There are in total 227,922 projects with 289,316 registered developers. We divide the 227,922 projects into three groups: successful projects, failed projects, and others. We use the number of downloads to categorize projects. Projects with more than 100,000 downloads are considered successful. Projects with less than 100 downloads are considered failed. Projects with number of downloads between 100 and 100,000 are considered as another separate group[2]. Unfortunately, SourceForge.net database dumps do not contain any accurate information on the number of downloads. Thus, we crawl the SourceForge.Net website to obtain the download history of each project. We find 2,448 projects (1.07% of all May 2010 projects) are categorized as successful, and 140,796 projects (61.77%) are categorized as failed.

We filter out projects with only one developer. From the 2,448 successful projects, 1,859 projects (75.94% of the successful projects) have more than one developer. Moreover from the 140,796 failed projects, 28,802 projects (20.46% of failed projects) have more than one developer. We also filter projects that exist on February 2005 as for those projects we could not ascertain the time the contributors join the project[3]. After we exclude those projects, we have 224 successful projects and 3,826 failed projects.

For each project in every group, we create a graph consisting of nodes which represent developers working on that project and edges which represent collaboration among developers in the project. To form rich graphs, we extract features representing the socio-technical aspects of the developers working on it. To extract the features, we need to first determine the time when a developer joins a project. This information is not directly available from the dump. Fortunately, we have the monthly snapshots and by contrasting the reported developers in two consecutive months, we could find the *month* a developer joins a project. For consistency reason, for all features, a granularity of one month is used.

Each node is labeled with information on: (a) number of past successful projects a developer has before he joins the current project, Past Successful Projects (PSP), (b) number of past unsuccessful projects a developer has before he joins the current project, Past Failed Projects (PFP), and (c) the length of time a developer has joined SourceForge.Net at the time he joins a project, Length of Membership (LOM).

Each edge is labeled with information on: (a) number of past successful collaborations two developers have before they start to work together in the current project, Past Successful Collaborations (PSC), (b) number of past failed projects two developers have before they start to collaborate in the current project, Past Failed Collaborations (PFC), and (c) the length of time that has passed, since the first time two developers worked together prior to the current project, Length of Collaboration History (LCH).

Four features: Past Successful Projects (PSP), Past Failed Projects (PFP), Past Successful Collaborations (PSC), and Past Failed Collaborations (PFC), could be obtained by analyzing the monthly dumps one by one. We just need to compare the month a developer or a pair of developers join a past project with the month the developer or the pair of developers join the current project. If a past project under comparison is either successful or failed, the counts of the corresponding features among the 4 are updated.

Following our definition of Length of Membership (LOM), we count the period of time that has passed since a developer first registered in SourceForge.Net until he joins the current project. SourceForge database dumps provide the time when a developer becomes a member of SourceForge.Net. To compute the feature:

[2] We exclude this group from our analysis

[3] This information is not recorded in SourceForge.Net dumps.

Length of Collaboration History (LCH), we analyze the monthly dumps in chronological order and for relevant pairs of developers, we find the month that they first work together in a single project.

At the end of the above process, we have 224 rich graphs corresponding to successful projects and 3,826 rich graphs corresponding to failed ones. All these rich graphs have three labels in the nodes and edges. For the 224 rich graphs, the maximum numbers of nodes and edges a rich graph has are 32 and 195 respectively, and the average numbers of nodes and edges are 3.76 and 5.92 respectively. For the 3,826 rich graphs, the maximum numbers of nodes and edges a rich graph has are 73 and 1,081 respectively, and the average numbers of nodes and edges are 2.86 and 3.98 respectively.

Translation & Reverse Translation. Following our framework description in Section 4, we first perform translation from the rich graphs (synthetic, myGamma, & SourceForge) to simple graphs before running the pattern mining algorithms (CloseGraph, gSpan, and Top-k LEAP). Finally, we perform reverse translation to recover rich subgraph patterns from the resultant mined simple subgraph patterns.

5.2 Experimental Results.

CloseGraph & gSpan. We run our translation algorithm on synthetic and real datasets. For the synthetic datasets, we use above mentioned generated datasets, i.e., $S10$ and $S20$ and for the real dataset, we use myGamma dataset.

For $S10$ dataset, the translation process takes *23.03* seconds to translate from synthetic rich graphs to translated simple graphs. The rich graphs from $S10$ dataset have an average size of 3.03 nodes and 2.59 edges; while the translated simple graphs have an average size of 11.40 nodes and 43.56 edges.

For $S20$ dataset, the translation process takes *23.50* seconds to translate from synthetic rich graphs to translated simple graphs. The rich graphs from $S20$ dataset have an average size of 3.04 nodes and 2.60 edges; while the translated simple graphs have an average size of 11.47 nodes and 43.96 edges.

We then run CloseGraph and gSpan algorithms on the translated simple graphs from $S10$ and $S20$ datasets. For CloseGraph, we use various supports, i.e., 60, 80, and 100. We also run gSpan with various supports, i.e., 3%, 4%, and 5%. Tables 2 and 3 show the results of our experiments on the synthetic datasets. Columns *Algorithm*, *Support*, and *Time* specify the graph pattern mining algorithms used, the support values used, and the runtime of each graph pattern mining algorithms. The inputs are the simple graphs resulted from the translation process. Column |*Graph Patterns*| describes the number of simple subgraph patterns mined by the graph pattern mining algorithms. These patterns are then translated to rich subgraph patterns by the reverse translation process. We present the running time of our reverse translation process in column *RTL Time*. The final number of resultant graphs are shown in column |*Final Graphs*|. The last column shows us the percentage of the translation & reverse translation runtime to that of the whole process.

The translation process translates 10,000 rich graphs from myGamma dataset in *8.1875 seconds*. The myGamma rich graphs have an average size of 4.27 nodes and 3.27 edges; while the translated simple graphs have an average size of 15.55 nodes and 48.25 edges.

For myGamma dataset, we also run CloseGraph and gSpan algorithm by varying their supports. For CloseGraph we use support 500, 600, and 700. For gSpan we set the support to 20%, 30%, and 40%. Figure 15 shows the results for myGamma dataset using CloseGraph algorithm. We show the results for myGamma dataset using gSpan algorithm in Figure 16.

Top-k LEAP.
The translation process for 224 successful projects takes *0.97 seconds*. For the 3,826 failed projects, the translation process takes *12.78 seconds*. The 224 translated simple graphs for the successful projects have an average size of 32.33 nodes and 303.09 edges. The 3,826 translated simple graphs for the failed projects have an average size of 24.62 nodes and 216.76 edges.

We run Top-k LEAP algorithm by Cheng *et al.* [5] on the translated simple graphs. We vary k value to mine top 50 and 100 discriminative subgraphs. Table 5 shows the running time of Top-k LEAP and Reverse Translation process for k=50 and k=100 respectively.

k	Disc. Patterns	Top-k LEAP Time	RTL Time
50	50	9,506 s	0.234 s
100	100	21,371 s	0.047 s

Table 5: Running Time: Top-k LEAP & Reverse Translation

5.3 Analysis. In terms of the number of nodes, after translation, the graphs grow by 3.76 times ($S10$ dataset), 3.77 times ($S20$ dataset), 3.64 times (myGamma dataset), 8.37-8.39 times (SourceForge dataset), which are less than the number of node labels multiplied by the number of edge labels (e.g., $2 \times 2 = 4$ for myGamma dataset and $3 \times 3 = 9$ for SourceForge dataset). All of these results are in line with our analysis presented in Section 4.2.

5.4 Applicability of Mined Rich Patterns for Classification. We use the mined discriminative rich

Dataset $S10$ (10,000 Rich Graphs)						
Translation (TL) Time: 23.03 s						
Algorithms	Support	Time	$\|GraphPatterns\|$	RTL Time	$\|FinalGraphs\|$	TL+RTL Portion
CloseGraph	60	28.65 s	13,244	24.39 s	4,500	62.34%
	80	18.5 s	4,675	1.42 s	749	56.93%
	100	17.61 s	4,229	1.17 s	635	57.88%
gSpan	3%	2,311.64 s	23,040	3.52 s	423	1.13%
	4%	380.05 s	841	0.06 s	66	5.73%
	5%	28.04 s	564	0.05 s	55	45.15%

Table 2: Experimental Result: Synthetic Datasets $S10$

Dataset $S20$ (20,000 Rich Graphs)						
Translation (TL) Time: 23.499 s						
Algorithms	Support	Time	$\|GraphPatterns\|$	RTL Time	$\|FinalGraphs\|$	TL+RTL Portion
CloseGraph	60	1,377.19 s	147,080	1,698.53 s	20,724	55.56%
	80	259.51 s	75,219	570.92 s	14,944	69.61%
	100	119.96 s	36,829	186 s	10,208	63.59%
gSpan	3%	3,872.92 s	21,836	3.45 s	433	0.69%
	4%	89.72 s	880	0.08 s	67	20.81%
	5%	59.52 s	529	0.05 s	55	28.35%

Table 3: Experimental Result: Synthetic Datasets $S20$

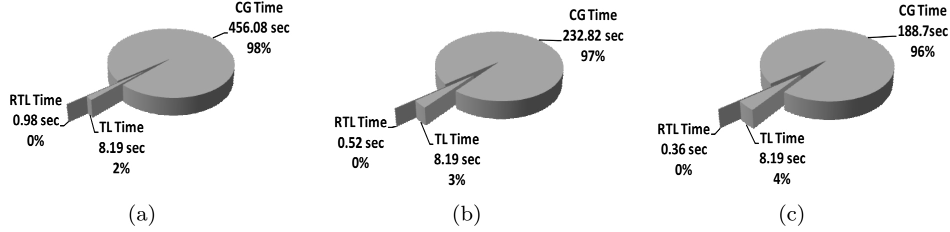

Figure 15: CloseGraph: myGamma Dataset, Support (a) 500 (b) 600 (c) 700

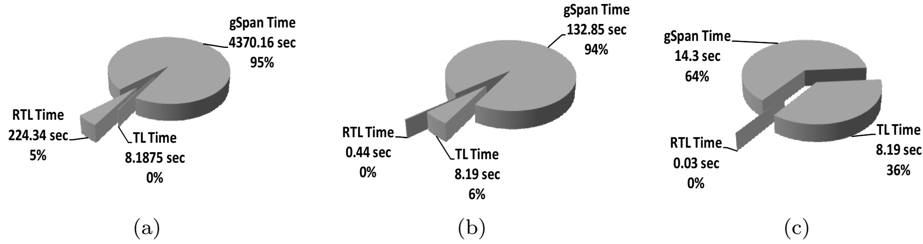

Figure 16: gSpan: myGamma Dataset, Support (a) 20% (b) 30% (c) 40%

myGamma Dataset (10,000 Rich Graphs)					
Translation (TL) Time: 8.1875 s					
Pattern Mining Alg.	Support	Time (s)	$\|GraphPatterns\|$	RTL Time	$\|FinalRichGraphs\|$
CloseGraph	500	456.08 s	2,831	0.98 s	1,227
	600	232.82 s	2,001	0.52 s	723
	700	188.7 s	1,664	0.36 s	532
gSpan	10%	4,370.16 s	242,551	224.34 s	5,944
	20%	132.85 s	3,354	0.44 s	181
	30%	14.30 s	182	0.03 s	21

Table 4: Experimental Result: myGamma Dataset

subgraph patterns as features for effective classification. We split our SourceForge dataset into two groups: training and testing. Given a training data containing a set of projects labeled as *successful* or *failed*, we mine for

patterns and learn a discriminative model. This model in turn is used to classify a set of projects with unknown labels.

We use LIBSVM [4] with probability estimates as the classification model. Classification accuracy, defined as the percentage of test cases correctly classified, is used as one measure. Due to the skewed class distribution, the measure AUC which is the area under a ROC curve is also used. ROC curve shows the trade-off between true positive rate and false positive rate for a given classifier [8]. A good classifier would produce a ROC curve as close to the top-left corner as possible. AUC is a measure of the model accuracy, in the range of $[0, 1.0]$. The best possible classifier would generate an optimal AUC value of 1.0.

We perform 10-fold cross validation, where for each we keep 1/10 of the data for testing and the other for training. We are able to predict the labels of the projects with 94.99% accuracy and 0.86 AUC.

6 Conclusion and Future Work

In this work, we propose a novel framework to transform rich graphs to simple graphs. We also introduce an algorithm to do reverse translation from rich graphs to simple graphs. Our work answers the need to mine graphs with nodes and edges having *multiple labels*. The problem is interesting as many datasets from various domains, e.g., social networks, are best represented using rich graphs. Our proposed framework works in three steps. First, the translation process translates graphs with *multiple* node and edge labels to corresponding graphs with *single* node and edge label. Second, the translated simple graphs would become inputs to the pattern mining algorithms. At the last step, we translate back the resultant simple graphs output by the pattern mining algorithms to rich graphs. By integrating our proposed framework with the simple graph based pattern mining algorithms, the simple graph based pattern mining algorithms now have a capability to handle rich graphs.

We experiment our approach with three pattern mining algorithms, i.e., CloseGraph, gSpan, and Top-k LEAP. We use various datasets as inputs to investigate the running time and scalability of our framework. From the results, we show that our proposed framework only adds a reasonable amount of time to perform translation from rich graphs to simple graphs and reverse translation from simple graphs back to rich graphs. Experiments on synthetic, myGamma, and SourceForge.Net datasets show that our framework could work and integrate very well with the existing pattern mining algorithms.

For future work, we plan to apply our algorithm to a variety of problems that are best modeled as a set of rich graphs. We also plan to extend our graph transformation approach to also support algorithms that mine patterns in a single graph setting.

Acknowledgement. We would like to thank Greg Madey for sharing with us the SourceForge.Net dataset. We also would like to thank BuzzCity for sharing the mobile social network dataset.

References

[1] myGamma: Mobile Social Networking Service. http://m.mygamma.com.
[2] M.V. Antwerp and G. Madey. Advancess in the sourceforge research data archive (SRDA). In *OSS*, 2008.
[3] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 211–218, 2002.
[4] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001.
[5] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *ISSTA*, 2009.
[6] L. Dehaspe, H. Toivonen, and R. King. Finding frequent substructures in chemical compounds. In *KDD*, pages 30–36, 1998.
[7] S. Ghazizadeh and S. S. Chawathe. Seus: Structure extraction using summaries. In *Proc. of the Int. Conf. on Discovery Science*, 2002.
[8] J. Han and M. Kamber. *Data Mining: Concepts and Techniques* (2nd ed.). Morgan Kaufmann, 2006.
[9] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the subdue system. In *KDD*, 1994.
[10] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.
[11] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 1998.
[12] N. Jin and W. Wang. Lts: Discriminative subgraph mining by learning from search history. In *ICDE*, 2011.
[13] Y. Ke, J. Cheng, and W. Ng. Correlation search in graph databases. In *KDD*, 2007.
[14] S. Kim, I. Song, and Y. J. Lee. An edge-based framework for fast subgraph matching in a large graph. In *DASFAA*, April 2011.
[15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, 2001.
[16] S. Nijssen and J. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.
[17] J. Prins, J. Yang, J. Huan, and W. Wang. Spin: Mining maximal frequent subgraphs from graph databases. In *KDD*, 2004.
[18] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *ICDM*, pages 458–465, 2002.
[19] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by scalable leap search. In *SIGMOD*, pages 433–444, 2008.
[20] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. of ICDM*, pages 721–724, 2002.
[21] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
[22] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *KDD*, 2003.
[23] Z. Zeng, J. Wang, J. Zhang, and L. Zhou. Fogger: an algorithm for graph generator discovery. In *EDBT*, pages 517–528, 2009.
[24] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining top-k large structural patterns in massive networks. In *VLDB*, 2011.
[25] K. Zhu, Y. Zhang, X. Lin, G. Zhu, and W. Wang. Nova: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs. In *DASFAA*, April 2010.