Approximate Graph Mining with Label Costs

Pranay Anchuri, Mohammed J. Zaki, Omer Barkol, Shahar Golan, Moshe Shamy

HP Laboratories
HPL-2013-36

Keyword(s):

data mining; graph analysis; cmdb; approximation techniques;

Abstract:

Many real-world graphs have complex labels on the nodes and edges. Mining only exact patterns yields limited insights, since it may be hard to find exact matches. However, in many domains it is relatively easy to compute some cost (or distance) between different labels. Using this information, it becomes possible to mine a much richer set of approximate subgraph patterns, which preserve the topol- ogy but allow bounded label mismatches. We present novel and scalable methods to efficiently solve the approximate isomorphism problem. We show that the mined approximate patterns yield interesting patterns in several real-world graphs ranging from IT and protein interaction networks to protein structures.

External Posting Date: June 6, 2013 [Fulltext] Approved for External Publication
Internal Posting Date: June 6, 2013 [Fulltext]

Approximate Graph Mining with Label Costs *

Pranay Anchuri[1], Mohammed J. Zaki[1], Omer Barkol[2], Shahar Golan[2], Moshe Shamy[3]
[1]CS Department, RPI, Troy NY, USA [2]HP Labs, Haifa, Israel [3]HP Software, Yahud, Israel
{anchupa,zaki}@cs.rpi.edu, {omer.barkol, shahar.golan, moshe.shamy}@hp.com

ABSTRACT

Many real-world graphs have complex labels on the nodes and edges. Mining only exact patterns yields limited insights, since it may be hard to find exact matches. However, in many domains it is relatively easy to compute some cost (or distance) between different labels. Using this information, it becomes possible to mine a much richer set of approximate subgraph patterns, which preserve the topology but allow bounded label mismatches. We present novel and scalable methods to efficiently solve the approximate isomorphism problem. We show that the mined approximate patterns yield interesting patterns in several real-world graphs ranging from IT and protein interaction networks to protein structures.

1. INTRODUCTION

Graphs are a natural way to model many of the modern complex datasets that typically have interlinked entities connected with various relationships. Examples include different types of networks, such as social, biological and technological networks. Tools for rapidly querying and mining graph data are therefore in high demand. Our focus is on graph pattern discovery methods that can simultaneously consider both the structure and content (e.g., node labels).

Whereas frequent graph mining has long been a well studied problem, most of the prior work has focused on exact pattern discovery. Graph mining involves two main steps. The first step is to generate non-duplicate candidate patterns, and the second is to compute the frequency of each candidate pattern. The former task requires graph isomorphism testing, whereas the latter requires subgraph isomorphism checking, since we need to count all the occurrences of a smaller graph within a much larger graph (or a set of graphs). Many efficient methods have been proposed for mining exact labeled graph patterns, including both complete search and sampling based approaches [9, 11, 12, 14, 15, 18]. These exact methods require that there be an exact match between the labels of nodes in the candidate pattern and in the database graph. This can potentially miss many patterns where nodes may share a high label similarity, but may not match exactly. This is specially true for more complex labels (e.g., text data), or in cases where the nodes represent some real-world objects (e.g., proteins, IT infrastructure nodes), where it may be possible to easily design a meaningful cost or distance matrix between node "labels". Unfortunately, exact isomorphism based methods cannot leverage the rich information from the cost matrix. What is required is a new class of algorithms that can mine frequent approximate patterns via approximate subgraph isomorphism that satisfies some bound on the overall cost of the match between a candidate and the database graph(s). Only a few methods have tackled this problem [5,13,19], but they typically enumerate all isomorphisms, and are therefore not scalable to large graphs due to the combinatorial explosion in the number of isomorphisms.

In this paper we present a new approach to mine frequent approximate patterns in a single large graph in the presence of a cost matrix between the labels. In particular we make the following contributions: i) We propose a novel approach to effectively prune the space of approximate labeled isomorphisms. Instead of enumerating all the possible isomorphisms, we maintain a set of *representatives* (mappings of the pattern vertex in the database) that is linear in the database and pattern size. Pruning is applied on this set to narrow down the search to only viable mappings. ii) We propose label based iterative pruning methods to compute the representative sets efficiently. These methods are based on k-hop labels and neighbor concatenated labels. iii) Our method handles both arbitrary as well as binary cost matrices. iv) We place our work within the pattern sampling paradigm, thereby avoiding complete search, which can be practically infeasible in real-world graphs, not to mention the information overload problem.

We study the effectiveness of the proposed methods on three real-world datasets. The first is a configuration management database graph, where the nodes represents entities comprising the IT infrastructure and the link represents relationships between them; approximate mining yields a richer set of de-facto IT policies in the company. The second dataset is a graph dataset representing 3D protein structures; mined patterns represent approximate motifs. The last dataset comes from a protein interaction network, where the nodes are proteins and edges indicate whether they interact physically (i.e., they may bind together or they may be part of the same protein complex); the mined approximate patterns represent molecular subnetworks and molecular machines (the protein complexes) that take part in important cellular processes. We show that our proposed techniques are indeed scalable and fruitful, allowing us to mine interesting approximate graph patterns from large real-world graphs.

2. PRELIMINARIES

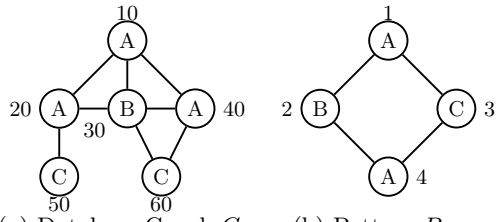
An undirected labeled graph G is represented as a tuple $G = (V_G, E_G, L)$ where V_G is the set of vertices, E_G is the set of edges and L: $V_G \rightarrow \Sigma$ is a function that maps vertices to their labels. The neighbors of a vertex v are given as $N(v) = \{u|(u, v) \in E_G\}$. A walk in a graph is a sequence of vertices $v_0, \ldots, v_k$ such that there is an edge between adjacent pairs of vertices i.e., $(v_i, v_{i+1})E_G$ and its length is k. A walk is a path if every vertex appears at most once in the sequence

*This work was supported in part by an HP Innovation Award and NSF Award CCF-1240646.

i.e., $v_i \neq v_j$, for $i \neq j$. We write $u \xrightarrow{k} v$ if there is a path of length k between u and v.

Cost matrix: We assume that there is a cost matrix $\mathcal{C}$: $\Sigma^2 \rightarrow \mathbb{R}^{\geq 0}$. The entry $\mathcal{C}[l_i][l_j]$ denotes the cost of matching the labels l_i and l_j. Typically $\mathcal{C}$ is usually symmetric and the diagonal entries are 0.

Approximate subgraph isomorphism: A graph $S = (V_S, E_S, L)$ is a subgraph of G, denoted $S \subseteq G$, iff $V_S \subseteq V_G$, and $E_S \subseteq E_G$. Given a database graph G and a pattern graph $P = (V_P, E_P, L)$, a function ϕ: $V_P \rightarrow V_G$ is called an *unlabeled subgraph isomorphism* provided ϕ is an injective (or one-to-one) mapping such that $\forall (u, v) \in E_P$, we have $(\phi(u), \phi(v)) \in E_G$. That is, ϕ preserves the topology of P in G. Define the cost of the isomorphism as follows: $\mathcal{C}(\phi) = \sum_{u \in V_P} \mathcal{C}[L(u)][L(\phi(u))]$, that is, the sum of the costs of matching the node labels in P to the corresponding node labels in G. We say that ϕ is an *approximate subgraph isomorphism* from P to G provided its cost $\mathcal{C}(\phi) \leq \alpha$, where α is a user-specified threshold on the total cost. In this case we also call P an approximate pattern in G. Note that if $\alpha = 0$, then ϕ is an exact subgraph isomorphism between P and G. From now on, *isomorphism* refers to *approximate subgraph isomorphism* unless specified otherwise.



(a) Database Graph G (b) Pattern P

$\mathcal{C}$	A	B	C	D
A	0	0.2	0.6	0.1
B	0.2	0	0.4	0.5
C	0.6	0.4	0	0.2
D	0.1	0.5	0.2	0

(c) Cost Matrix

	approx. isomorphisms Φ				
P	1	2	3	4	cost
ϕ_1	30	10	60	40	0.4
ϕ_2	40	10	60	30	0.4

(d) Approximate Embeddings

Figure 1: (a): database graph G, (b): approximate pattern P. (c): cost matrix. (d): approximate embeddings of P.

Pattern support: Given a (large) database graph G, a pattern graph P, and the set of all approximate subgraph isomorphisms Φ from P to G, the *support* of P is some *anti-monotonic* function $sup(P, \Phi)$, i.e., $sup(P, \Phi_P) \leq sup(Q, \Phi_Q)$ for all subgraphs Q of P. We discuss some of the common support definitions in Sec 4.2. A pattern P is called *frequent* if $sup(P) \geq minsup$, where $minsup$ is a user defined support threshold. P is *maximal* iff P is frequent and there does not exist any supergraph of P that is frequent in G.

Representative set : Given a node $u \in V_P$, its *representative set* in the database graph G is the set

$$R(u) = \{v \in V_G | \ \exists \phi, \text{ such that } \mathcal{C}(\phi) \leq \alpha \text{ and } \phi(u) = v\}$$

That is, the representative set of u comprises all nodes v in G that u is mapped to in some isomorphism ϕ. Figure 1 shows an example database, a cost matrix, an approximate pattern, and its approximate subgraph isomorphisms for $\alpha = 0.5$. There are only two possible approximate isomorphisms from P to G, as specified by ϕ_1 and ϕ_2. For example, for ϕ_1, we have $\phi_1(1) \rightarrow 30$, $\phi_1(2) \rightarrow 10$, $\phi_1(3) \rightarrow 60$, and $\phi_1(4) \rightarrow 40$, as seen in Table 1d. The cost of the isomorphism is $\mathcal{C}(\phi_1) = 0.4$, since $\mathcal{C}(L(1), L(30)) + \mathcal{C}(L(2), L(10)) + \mathcal{C}(L(3), L(60)) + \mathcal{C}(L(4), L(40)) = \mathcal{C}(A, B) + \mathcal{C}(B, A) + \mathcal{C}(C, C) + \mathcal{C}(A, A) = 0.2 + 0.2 + 0 + 0 = 0.4$. The representative set for node $1 \in V_P$ is $R(1) = \{30, 40\}$.

Outline of our Approach: The two main steps in approximate graph mining are candidate generation and support computation. With candidate generation the search space of the frequent patterns is explored. For each candidate that is generated, we can check whether it is frequent by computing its support. Given a pattern with k vertices, the maximum number of possible isomorphisms are $k! \times \binom{|V_G|}{k}$. It is therefore infeasible to either enumerate or store the complete set of isomorphisms. Computing and storing the representative sets is a compromise that will enable us to decide efficiently if a candidate is frequent.

3. COMPUTING REPRESENTATIVE SETS

Representative vertex v of a pattern vertex u implies that there exists an isomorphism ϕ for which $\phi(u) = v$. One way to interpret it is that the neighborhood of u matches with that of v. By comparing the neighborhoods we can find vertices that are not valid representatives of u without trying to find an isomorphism exhaustively. Therefore, to compute the representative sets we will start with a candidate representative set denoted by $R'(u)$ and iteratively prune some of the vertices if the neighborhoods cannot be matched. The candidate set is a super set of the representative set, $R'(u) \supseteq R(u)$. An example of a candidate set is $R'(u) = \{v | v \in V_G, \mathcal{C}[L(u)][L(v)] \leq \alpha\}$, i.e., all the isomorphisms of the single vertex pattern with label $L(u)$. In this section, we will describe different notions of neighborhood and show how they help us in computing the final representative set of vertices in a pattern. Note that the problem of checking whether a vertex $v \in R(u)$ is an NP-Hard problem, since it involves the approximate subgraph isomorphism problem. The pruning methods typically do not prune all the invalid vertices. So, we use a final enumeration-based verification step to prune the remaining invalid vertices and reduce $R'(u)$ to the true $R(u)$ (as described in Sec. 3.3).

3.1 k-hop Label

k-hop label is defined as the set of vertices that are reachable via a simple path of length k. In other words, k-hop label contains all vertices that are reachable in k-hops starting from u and by visiting each vertex at most once. Note that, we use the word label even though we refer to a set of vertices. Formally, the k-hop label of a vertex u in graph G, $h_k(u, G) = \{v | v \in G, u \xrightarrow{k} v\}$. We simply write it as $h_k(u)$ when the graph is evident from the context. For example, for pattern P in Fig. 2a, the 0-hop label of vertex 5 is $h_0(5) = 5$, its 1-hop label is $h_1(5) = 2, 4, 6$ (we omit the set notation for convenience) and its 2-hop label $h_2(5) = 1, 3$. The minimum cost of matching k-hop labels $h_k(u)$ and $h_k(v)$ is

$$\mathcal{C}_k[h_k(u)][h_k(v)] = \min_f \sum_{u' \in h_k(u)} \mathcal{C}[L(u')][L(f(u'))] \qquad (1)$$

where the minimization is over all injective functions f: $h_k(u) \rightarrow h_k(v)$ and $\mathcal{C}[L(u')][L(f(u'))]$ is the cost of matching the vertex labels. In other words, it is the minimum total cost of matching the vertices present in the k-hop labels. The following theorem places an upper bound on the minimum cost of matching the k-hop label of a pattern vertex and any of its representative vertices.

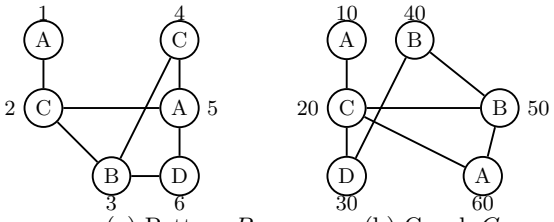
Theorem 1. *Given any pattern vertex u, a representative vertex $v \in R(u)$ and cost threshold α, then*

$$\mathcal{C}_k[h_k(u)][h_k(v)] \leq \alpha, \textit{for all } k \geq 0$$

Proof: *Omitted due to lack of space.*

Based on the above theorem, a vertex v is not a representative vertex of u if $\mathcal{C}_k[h_k(u)][h_k(v)] > \alpha$ for any $k \geq 0$. However, in practice, it enough to check the condition only for $k \leq |V_P| - 1$ because $h_k(u)$ is the null set $\forall k \geq |V_P|$ and the condition is trivially satisfied.

Figure 2 shows an example for the k-hop label based pruning of the candidate representative set where the threshold $\alpha = 0.5$. Consider vertex $2 \in V_P$ and vertex $20 \in V_G$, we have, $\mathcal{C}_0[h_0(2)][h_0(20)] = 0$, since the cost of matching vertex labels $\mathcal{C}[L(2)][L(20)] = 0$, as per the label matching matrix C in Fig. 2c. The k-hop labels for $k = 1, 2, 3$ and the minimum of cost matching them are as shown in the Table 1, and it can be verified that the minimum cost is within the threshold α. Thus far, we cannot prune node 20 from $R'(2)$. However, $h_4(2) = 4, 6$ and $h_4(20) = 30, 60$ and the minimum cost of matching them is $0.6 > \alpha$. Thus, from Theorem 1 we conclude that $20 \notin R'(2)$. This example illustrates that k-hop labels can help prune the candidate representative sets.



(a) Pattern P (b) Graph G

$\mathcal{C}$	A	B	C	D
A	0	0.7	0.6	0.1
B	0.7	0	0.3	1
C	0.6	0.3	0	0.8
D	0.1	1	0.8	0

(c) Cost Matrix

Figure 2: Pattern (a), db graph (b), and cost matrix (c).

k	$h_k(2)$	$h_k(20)$	$\mathcal{C}_k[h_k(2)][h_k(20)]$
1	1, 3, 5	10, 30, 50, 60	0
2	4, 6	40, 50, 60	0.4
3	3, 5	40, 30, 50	0.1

Table 1: k-hop label of vertices 2 and 20

k	$h_k(3)$	$h_k(50)$	$\mathcal{C}_k[h_k(3)][h_k(50)]$
0	3	50	0
1	2, 4, 6	20, 40, 60	0.4
2	1, 5	10, 20 , 30, 60	0
3	2, 4, 6	10, 20, 30, 40	0.3
4	1	10, 40, 60	0

Table 2: k-hop labels of vertices 3 and 50.

3.2 Neighbor Concatenated Label

In neighbor concatenated label (NL), the information regarding the candidates of a neighbor that were pruned in the previous iteration is used along with the current k-hop label to prune candidates in the current iteration. In contrast, the k-hop label pruning strategy for a vertex u works independently of the result of k-hop label pruning of other vertices in the pattern. This leads us to the following recursive formulation for NL.

The NL of a vertex in the $k+1^{th}$ iteration, $\eta_{k+1}(u)$, is defined as the tuple $(\{\eta_k(u')|u' \in N(u)\}, h_{k+1}(u))$. The first element (denoted X) of the tuple is the set of NL of the neighbors of the vertex u in the previous iteration k, and the second element (denoted Y) is exactly same as the (k+1)-hop label defined in the Sec. 3.1. We say that $\eta_{k+1}(u)$ is dominated by $\eta_{k+1}(v)$, denoted as $\eta_{k+1}(u) = (X, Y) \preceq \eta_{k+1}(v) = (X', Y')$, iff i) $\mathcal{C}_{k+1}[Y][Y'] \leq \alpha$, i.e., the minimum cost of matching the (k+1)-hop labels is within α, and ii) there exits an injective function $g\colon X \to X'$ such that $x \preceq g(x)$ for all $x \in X$ i.e., there is a one to one mapping between the NL labels (of the previous iteration k) of neighbors of u and v. The base case $\eta_0(u) \preceq \eta_0(v)$ iff $\mathcal{C}[L(u)][L(v)] \leq \alpha$. For example, in Fig. 2, $\eta_1(2) \preceq \eta_1(20)$ because $\mathcal{C}_1[h_1(2)][h_1(20)] \leq \alpha$ and the NL labels of vertices $1, 3, 5$ are dominated by the NL labels of vertices $10, 50, 30$ or $10, 50, 60$ respectively. The following theorem states that the NL of a pattern vertex u is dominated by the NL of any of its representative vertices $v \in R(u)$.

Theorem 2. *Given any pattern vertex u, a representative vertex $v \in R(u)$, and cost threshold α, then*

$$\eta_k(u) \preceq \eta_k(v), \text{ for all } k \geq 0$$

Proof: *Omitted due to lack of space.*

Based on the above theorem, a vertex v can be pruned from $R'(u)$ if $\eta_k(u) \not\preceq \eta_k(v)$ for some $k \geq 0$. In Fig. 2, consider the vertices $3 \in P$, $50 \in G$ and let $\alpha = 0.5$. The NL labels, $\eta_0(3) \preceq \eta_0(50)$ as $\mathcal{C}[B][B] = 0 \leq \alpha$. Similarly it is also true for the pairs $(2, 20)$, $(4, 40)$ etc. It follows that $\eta_1(3) \preceq \eta_1(50)$ as the neighbors $2, 4, 6$ can be mapped to $20, 40, 60$ respectively and the minimum cost of the matching the 1-hop label is 0.4 which is less than the α threshold. In the next iteration the NL labels of vertices 3 and 50 are $(\{\eta_1(2), \eta_1(4), \eta_1(6)\}, \{1, 5\})$ and $(\{\eta_1(20), \eta_1(40), \eta_1(60)\}, \{10, 20, 30, 60\})$ respectively. But $\eta_2(3) \not\preceq \eta_2(50)$ because the NL label $\eta_1(6)$ is not dominated by any of $\eta_1(20), \eta_1(40), \eta_1(60)$. So, there is no mapping between the neighbors of vertices 3 and 50 in the second iteration. Hence, $50 \notin R(3)$. Note that using the k-hop label in the same example will not prune the vertex 50 because the minimum cost of matching the k-hop labels is within α as shown in Table 2. Therefore, NL label is more efficient compared to k-hop label as it subsumes the latter label.

3.3 Candidate set verification

The pruning methods based on the k-hop and the NL labels start with a $R'(u)$ and prune some of the candidate vertices based on Theorems 1 and 2. The verification step reduces $R'(u)$ to $R(u)$ by retaining only those vertices v for which there exists an isomorphism ϕ in which $\phi(u) = v$. Informally, it does this by checking if the pattern P can be embedded at v such that total cost of label mismatch is at most α.

A vertex $v \in R(u)$ iff for any walk $W_p = u_0, u_1, \ldots, u_m$ (with $u = u_0$) that covers all the edges in pattern P, there exists a walk $W_d = v_0, v_1, \ldots, v_m$ (with $v = v_0$) in the database graph G, which satisfies the following conditions: i) $u_i = u_j \implies v_i = v_j$, ii) $(v_i, v_{i+1}) \in E_G$, and iii) $\sum \mathcal{C}[L(u_i)][L(v_i)] \leq \alpha, u_i \in \{W_p\}$ i.e, the summation is over each unique vertex u_i. Unlike the NL label condition, the above conditions are necessary and sufficient and follow directly from the definition of approximate isomorphism.

To check whether $v \in R(u)$, we first map u to v and subtract the cost of $\mathcal{C}[L(u)][L(v)]$ from the threshold α. We then try to map the remaining vertices in P by following W_p one edge at a time. In any step (u_i, u_{i+1}), if u_i and u_{i+1} are mapped to x and y in G, respectively, then we ensure that $(x, y) \in E_G$ (condition ii). If on the other hand, u_{i+1} is not mapped then we map it to some vertex in $y \in R'(u_{i+1})$ and subtract the cost $\mathcal{C}[L(u_{i+1})][L(y)]$ from the

remaining α threshold. We backtrack if any of the conditions is violated. The vertex $v \in R(u)$, if we can complete the walk W_p satisfying the above three conditions.

Consider whether vertex $30 \in R(1)$ for the pattern in Fig. 1b, and let $\alpha = 0.5$. The sequence $W_p = 1, 2, 4, 3, 1$ is a walk in the pattern that covers all the edges. In general, finding a walk that covers all the edges in a graph is a special case of Chinese postman problem [6] when the edge weights are one. We first map 1 to 30 an subtract the cost $\mathcal{C}[L(1)][L(30)] = 0.2$ from 0.5. For the first edge, $(1, 2)$, since 2 is not mapped we map it to some vertex, say 20. The cost of the mapping is 0.2 and the remaining threshold is $0.3 - 0.2 = 0.1$. It can be verified that these mappings cannot complete the walk W_p. So we backtrack and map 2 to another vertex say 10. This walk can be completed with the mappings as in ϕ_1 in Table 1d and the remaining cost is 0.1. The mappings of the pattern vertices not only implies that $30 \in R(1)$, it also tells us that $10, 60, 40$ represent vertices $2, 3, 4$ respectively. The above procedure can be easily extended to enumerate all the isomorphisms of the pattern.

3.4 Label costs and dominance checking

Candidate representative vertices are pruned by checking for dominance relation between the NL labels of pattern vertex and that of candidate vertex in the database. Comparing the NL labels requires i) computing the cost of matching the k-hop labels ii) matching the neighbors of a pattern vertex with neighbors of a candidate vertex such that the NL of the candidate vertex dominates that of pattern vertex. First problem can be formulated as a minimum cost maximum flow in a network, and the second as maximum matching in a bipartite graph.

Computing k-hop label cost: The minimum cost of matching the k-hop labels $h_k(u)$ and $h_k(v)$ is equal to minimum cost maximum flow in a network F defined as follows. Each edge in F is associated with a maximum capacity and a cost for sending one unit of flow across it. The network contains a vertex for each label $l_u = L(u')$ where $u' \in h_k(u)$ and a vertex for each label $l_v = L(v')$ where $v' \in h_k(v)$. There is a directed edge between between source vertex (s) and each l_u with zero cost and a capacity equal to the multiplicity of the l_u, i.e., the number of vertices in $h_k(u)$ that have the label l_u. Similarly there is a directed edge between l_v and the sink node (t). In addition, there is a directed edge from l_u to l_v with a cost equal to $\mathcal{C}[l_u][l_v]$ and a capacity equal to the multiplicity of l_u. The cost between the k-hop labels is equal to the minimum cost for maximum flow if the maximum flow is equal to $|h_k(u)|$ and ∞ otherwise.

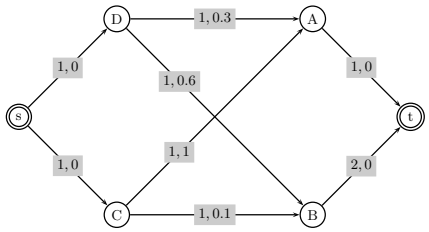


Figure 3: Flow network for $h_2(2)$ and $h_2(20)$

Fig. 3 shows the flow network required to compute the minimum cost of matching the k-hop labels $h_2(2) = 4, 6$ and $h_2(20) = 40, 50, 60$ as shown in Table 1. The labels of vertices in the k-hop labels are C, D and B, B, A respectively. There is an edge from s to each of C, D with zero cost and maximum capacity of one. Similarly, there is an edge from each of A, B to the sink vertex t with zero cost and maximum capacity of one and two respectively. The capacity of the edge between B and t is two because both the vertices 40 and 50 have the same label B. There is an edge from C, D to each of A, B with cost equal to the corresponding entry in the cost matrix C. The maximum flow in the network is two and the minimum cost of sending two units of flow 0.4 is achieved by pushing a unit flow along the paths s, C, B, t and a, D, A, t. Therefore, the cost of matching the labels $h_2(2)$ and $h_2(20)$ is 0.4. Thus, vertex 4 with label C can be matched to either 40 or 50 and the vertex 6 to 60.

Dominance check: Consider the NL labels $\eta_{k+1}(u) = (X, Y)$ and $\eta_{k+1}(v) = (X', Y')$, the cost of matching the k-hop labels Y and Y' can be computed using the above the network formulation. Finding an injective function $f\colon X \to X'$ such that $x \preceq f(x)$, is equivalent to finding a matching of size $|N(u)|$ in the bipartite graph with edges (x, x'), for all $x \in X$ and $x' \preceq X'$. The NL label $\eta_k(u)$ is therefore dominated by $\eta_k(v)$ if the cost between the k-hop labels is within α and the size of maximum bipartite matching is $|N(u)|$.

Optimization: The candidate pattern may contain groups of symmetric vertices that are indistinguishable with respect to the k-hop label. In such a scenario, the candidate representative sets of all these vertices are exactly the same. Utilizing the symmetry, we can apply the label pruning strategy only on one vertex per symmetry group and replicate the results for all other vertices in the group. For example, the vertices 1 and 4 in Fig. 1b are symmetric and the representative sets $R(1)$ and $R(4)$ are exactly the same. In abstract algebra terms such groups are called orbits of the graph and can be computed by using the Nauty algorithm [16]. Even though computing the orbits is expensive, we can avoid $(|g| - 1) \times |R'(u)|$ NL dominance checks (g is the size of an orbit) due to the fact that NL dominance checks are performed only on one vertex in each group. Note that we find the orbits only for the pattern which is usually very small compared to the database graph.

3.5 Precomputing database k-hop labels

The k-hop label of a database vertex is independent of the candidate pattern. Also, the flow network to compute the cost of matching the k-hop labels requires only the aggregate information about the multiplicity of the vertex label in the k-hop label. Hence, we can precompute the k-hop label of the database vertices and store them in memory. The following theorem proves that computing k-hop label is expensive.

Theorem 3. *Given a graph G, k, and $u \in V_G$, then computing $h_k(u)$ is NP-Hard.*
Proof: *Omitted due to lack of space.*

To compute k-hop label of a vertex u, we check for each vertex v whether $v \in u \xrightarrow{k} v$ by enumerating all possible k length paths until a path is found. This procedure is exponential, we therefore fix a maximum value k_{max} and use the NL label based pruning only for values of $k \leq k_{max}$. It only takes a small amount of time to compute the k-hop label for $k \leq 6$ for all the vertices in the database graph; significantly less than the overall run time of the mining algorithm. Once $h_k(u)$ is computed we store in memory only the tuples (l, m) where m is the multiplicity of the label $l = L(u')$. The total amount of main memory required to store the precomputed k-hop labels is $\mathrm{O}(|V_G| \times |\Sigma| \times k_{max})$.

4. MINING ALGORITHM

Having described the key contributions of label based pruning and candidate representative set verification, we now

briefly describe our algorithm for mining approximate subgraphs in the presence of a label cost matrix. The main steps of the mining algorithm include candidate generation and support computation. The representative set for each pattern vertex comprise a compact view of all the isomorphisms of the pattern in the input graph. We now show how the representative sets can be used in conjunction with different candidate generation and support computation techniques to yield approximate graph mining algorithms with different properties.

4.1 Candidate Generation

The search space of the frequent patterns forms a partial order. It can be explored in a depth first or breadth first order but doing so requires computing canonical code to avoid duplicates. Since the search space is exponential, sampling methods have gained traction in recent times [4, 9]. In our algorithm we employ the depth-first random edge extension strategy we proposed in [3], i.e., we employ random walks over the chains of the frequent subgraph partial order. Each random walk starts with an empty pattern and repeatedly add a new edge to a new vertex, or connects two existing vertices in the pattern to generate a new candidate. More precisely, at any stage of the walk let Q be the current frequent pattern. A candidate pattern P is generated from Q either by adding a new vertex with label l or by connecting two existing vertices $u, v \in V_Q$. For any vertex u, if $u \in V_P \cap V_Q$ then the candidate representative set $R'(u)$ in P is the same as the representative set $R(u)$ verified for Q. Otherwise $u \in V_P \setminus V_Q$, and the candidate representative set is $R'(u) = \{v | v \in V_G, \mathcal{C}[L(u)][L(v)] \leq \alpha\}$, i.e., we start with the current representatives if the vertex is already present, otherwise it is the set of vertices in G whose label matching cost is within α. Using the label pruning and verification mechanism we compute the representatives of P. Then we decide if the pattern is frequent using the support function that we will define in Sec. 4.2. If the candidate pattern P is frequent, then we continue the walk by extending P. Otherwise, we try another random edge extension from Q. If no extension of Q leads to a frequent pattern then by definition Q is maximal and we terminate the current random walk. Using an input parameter K, our algorithm performs K random walks (by default), or outputs K distinct maximal approximate patterns (if desired). Furthermore, if the application requires a complete set of maximal patterns an ordered exploration of the search space may be employed.

4.2 Support Computation

The support of a pattern is an anti-monotonic function on the set of isomorphisms of the pattern. The anti-monotonicity means that the support of a pattern cannot be greater than the support of any of its subgraphs. Therefore, if a candidate pattern is found to be infrequent we can prune the entire subtree under it from the search space. This helps in pruning the otherwise exponential search space.

When mining from a database of graphs, a function as simple as the total number of graphs having at least one isomorphism is anti-monotonic. This approach cannot be used when mining from a single graph as it leads to a binary support function which is not very informative. On the other hand, counting the number of isomorphisms is not anti-monotonic because a graph can have more isomorphisms compared to its subgraph.

An anti-monotonic support function for a single graph is the maximum number of vertex disjoint isomorphisms. However, this requires computing the maximum independent set (MIS) in a graph where a vertex represents an isomorphism, and an edge exists if the isomorphisms share a vertex in common. This is called the MIS support of the pattern. Clearly, it is not feasible to compute the MIS support when the input graphs are large and patterns have large number of isomorphisms. An easy upper bound on the MIS support is the size of the smallest representative set of a vertex in the pattern. Thus, we define the *support* of pattern P in a database graph G as

$$sup(P) = \min_{u \in V_P} \{|R(u)|\}$$

That is, the minimum cardinality over all representative sets of vertices in P. The size of representative sets constructed from the disjoint isomorphisms is equal to the MIS support. Hence, $sup(P)$ is at least as large as MIS support. Other upper bounds for the MIS value have been proposed in gApprox [5] and CMDB-Miner [2] algorithms. The support function used in gApprox can be computed from the representative sets by enumerating the isomorphisms as described in the Sec. 3.3. The support function used by the CMDB-Miner algorithm can also be used by constructing the appropriate flow network on the representative sets. In conclusion, we can mix and match different techniques for candidate generation and support computation to produce different versions of the approximate graph mining algorithm since the isomorphisms are stored as representative sets.

4.3 Complexity

Space Complexity: At any given stage of the mining process, we need to store the candidate representative sets and the precomputed k-hop labels. For a pattern with m vertices, the total amount of memory is $O(m \times |V_G| + k_{max} \times (|\Sigma| * |V_G|))$. The first term corresponds to the representative sets and the second to the precomputed k-hop labels. k_{max} is the maximum value of k for which we compute k-hop labels.

Time Complexity: The cost of matching the k-hop labels requires at most $|h_k(u)|$ augmentations in the flow network F, which is an upper bound on the min cost assuming the cost on each edge is at most one. Each augmentation involves cycle detection which takes $O(|\Sigma|^3)$ time because the number of vertices in F is $O(|\Sigma|)$. Therefore, the time for minimum cost flow is $O(|h_k(u)| \times |\Sigma|^3)$. The time for the bipartite matching in a graph is proportional to the number of vertices and the edges. Since, we try to match the neighbors of a pattern and candidate vertex, the number of vertices is bounded by the maximum degree d_{max} of the pattern and candidate vertices. Therefore, the total time for each dominance check is $O(|h_k(u)| \times |\Sigma|^3 + d_{max}^3)$. The number of dominance checks performed per candidate are $O(k_{max} \times n_g \times |V_G|)$ where n_g is the number of orbit groups in the pattern vertex.

5. EXPERIMENTAL EVALUATION

We ran experiments on several real world datasets to evaluate the performance of our algorithm. All the experiments were run on an 4GB Intel Core i7 machine with a clock speed of 2.67 GHz running Ubuntu Linux 10.04. The code was written in C++ and compiled using g++ version 4.4 with -O3 optimization flag. The default number of random walks is $K = 500$.

Dataset	$\|V\|$	$\|E\|$	$\|\Sigma\|$	Pre processing time
CMDB	10466	15122	84	329.31s
SCOP	39256	154328	20	17.38s
PPI	4950	16515	4950	–

Table 3: Input graph statistics

5.1 Configuration Management DB (CMDB)

A CMDB is used to manage and query the IT infrastructure of an organization. It stores information about the so-called configuration items (CIs) – servers, software, running processes, storage systems, printers, routers, etc. As such it can be modeled as a single large multi-attributed graph, where the vertices represent the various CIs and the edges represent the connections between the CIs (e.g., the processes on a particular server, along with starting and ending times). Mining such graphs is challenging because they are large, complex, multi-attributed, and have many repeated labels. We used a real-world CMDB graph for a large multi-national corporation (name not revealed due to non-disclosure issues) from HP's Universal Configuration Management Database (UCMDB). Table 3 shows the size of the CMDB graph, and also the time for precomputing the k-hop labels.

Cost Matrix: The set of labels in a CMDB form a hierarchy which can be obtained from HP's UCMDB. In the absence of domain knowledge, one way to obtain a cost matrix is by assigning low costs for pairs of labels that share many ancestors in the hierarchy and high costs otherwise. The algorithm is general in that it doesn't depend on how the label matching costs are assigned, the range of these values or whether the cost matrix is symmetric. Consider any two labels l_1, l_2 and their corresponding paths p_1, p_2 to the root vertex in the hierarchy. We first define the similarity between the labels to be proportional to the number of common labels in $p_1 \cap p_2$, as follows $sim(l_1, l_2) = \frac{|p_1 \cap p_2|}{2}\left(\frac{1}{|p_1|} + \frac{1}{|p_2|}\right)$. The cost of matching the labels is then $\mathcal{C}[l_1][l_2] = 1 - sim(l_1, l_2)$.

minsup	Time	Avg. Time
10	5604.35	11.21
15	7147.11	14.29
20	7931.56	15.86

Table 4: CMDB: Time (sec) for random walks

minsup	Fwd	Fwd Success	Back	Back Success
10	4.35k	0.5k	1.1k	0.12k
15	5.55k	0.91k	1.22k	0.12k
20	6.3k	0.83k	1.32k	0.85k

Table 5: CMDB: Number of extensions and successes

Results: Table 4 shows the time for $K = 500$ random walks for different values of *minsup* and $\alpha = 0.5$. The average time per random walk is also shown. Somewhat counterintuitively, the time increases for higher minimum support values. This can be explained by the fact that the CMDB graph contains many relatively small subgraphs with low support, and few relatively large subgraphs with high support. Thus, when *minsup* is high, more random edge extensions have to be tried to find the frequent ones, whereas when *minsup* is low fewer random edge extensions are required to locate the frequent patterns. This trend is verified in Table 5, which shows the total number of forward (adding a new label) and backward (connecting two existing vertices in the pattern) edge extensions tried by our algorithm, and also the number of extensions tried that result in a success (i.e., a frequent pattern). We can see that higher *minsup* in general requires more extensions for the CMDB graph.

Example Patterns: Figure 4 shows a maximal approximate pattern found in the CMDB graph, representing a typical "de-facto" configuration of the IT infrastructure in this company. It shows the connection between some services running on an NT server, and also the web/ftp services. The node with label *9 × process* indicates that there are nine nodes in the maximal pattern with label *process* all of which are connected to the *nt* node. This is an example where the run time for computing the representative sets is significantly reduced by the optimization proposed in section 3.4. All of the nine nodes belongs to the same orbit and hence their representative sets are identical.

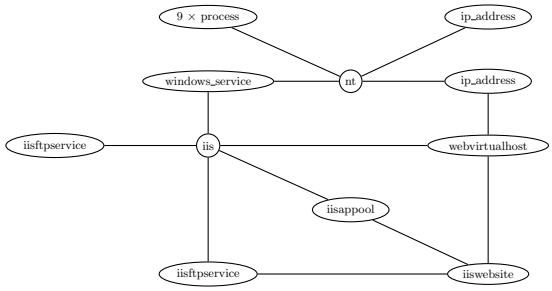


Figure 4: CMDB: Approximate Pattern

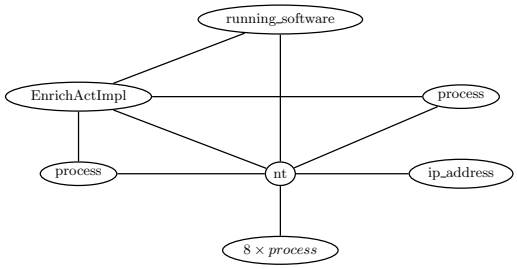


Figure 5: Complete Enumeration Expensive

To show the effectiveness of the pruning based on labels, we compared the time taken to enumerate a single maximal pattern in the CMDB graph. We compared the time with and without label-based pruning. Both the methods terminated the random walk with the maximal pattern shown in Figure 5. However, the total time taken to enumerate the pattern without using any derived label is 18306 secs whereas by using the NL label the total time reduced to only 15.58 secs. The huge difference between the times arises due to the multiplicity (labels with many occurrences) effect in CMDB graphs.

5.2 Protein Structure Dataset (SCOP)

SCOP (`scop.mrc-lmb.cam.ac.uk/scop/`) is a hierarchical classification of proteins based on structure and sequence similarity. The four levels of hierarchy in this classification are: class, fold, superfamily and family. The 3D structure of a protein can be represented as an undirected graph with the vertex labels being the amino acids, with an edge connecting two nodes if the distance between the 3D coordinates of the two amino acids (their α-Carbon atoms) is within a threshold (we use 7 Angstroms). We constructed a database of 100 protein structures belonging to 5 different families with 20 proteins from each family. We chose the proteins from different levels in the SCOP hierarchy, and we also focused on large proteins (those with more than 200 amino acids). The 3D protein structures were downloaded from the protein data bank (`http://www.rcsb.org/pdb`). The database can be considered as a single large graph with 100 connected components. The graph characteristics and k-hop label pre computation times are shown in Table 3. For the SCOP dataset, the support is redefined as the number of proteins containing the pattern, i.e., even if a protein contains multi-

ple isomorphisms we count them only once for the support.

Cost Matrix: Since there are 20 different amino acids, we need a 20×20 cost matrix. BLOSUM62 [10] is a commonly used substitution matrix for aligning protein sequences. The i, j entry in BLOSUM denotes the log-odd score of substituting the amino acids a_i and a_j, defined as: $\mathcal{B}[i][j] = \frac{1}{\lambda} \log \frac{p_{ij}}{f_i \cdot f_j}$, where p_{ij} denotes the probability that a_i can be substituted by a_j; f_i, f_j denote the prior probabilities for observing the amino acids; and λ is a constant. We compute f_i and f_j from the database, and then reconstruct $p_{ij} = f_i f_j e^{\lambda \mathcal{B}[i][j]}$. Next, we define the pair-wise amino acid cost matrix as $\mathcal{C}[i][j] = 1 - \frac{p_{ij}}{p_{ii}}$, which ensures that the diagonal entries are $\mathcal{C}[i][i] = 0$.

	NL pruning		no pruning	
α	Time	Avg. Time	Time	Avg. Time
0.01	57.27	0.11	115.06	0.23
0.7	228.65	0.45	394.93	0.76
1.0	2689.54	5.37	5966.53	11.93
1.5	6376.93	12.75	12572.96	25.14

Table 6: SCOP: Effect of α (Time in sec)

Algorithm	Time	Avg. Time
NL Pruning	2689.54	5.37
no Pruning	5966.53	11.93
gApprox	15653.2	31.30

Table 7: SCOP: Time (sec) Comparison

minsup	k-hop label	NL label	Verification	Total
10	377.10	378.99	629.804	1796.61
20	476.10	489.59	1188.25	2689.54
25	462.52	513.25	1055.28	2572.33
40	461.30	625.52	1148.42	2818.81

Table 8: SCOP: Time (sec) for Different Steps vs. *minsup*

minsup	Fwd	Fwd Success	Back	Back Success
10	4.89k	0.57k	0.67k	0.44k
20	6.24k	0.27k	0.44k	0.17k
25	6.48k	0.25k	0.40k	0.13k
40	5.6k	0.23k	0.32k	0.1k

Table 9: SCOP: Number of extensions and successes

minsup	neighbor checks	k-hop checks	Verify checks
10	809k	65.5k	1.78k
20	925k	92.9k	1.72k
25	792k	93.32k	1.24k
40	721k	139.78k	9.63k

Table 10: SCOP: Number of matching neighbors, k-hop distance and verification computations.

Results: Table 6 shows the time taken for enumerating approximate maximal patterns for different values of α (with fixed $minsup = 20$). The table shows the time for $K = 500$ random walks and the average time per walk, with and without the label pruning. It can be seen that by using the label-based pruning the time for random walks reduces significantly (by over 100%). As expected, the time increases as the values of α increases, since the number of isomorphisms clearly increases for a more relaxed (larger) cost threshold. When $\alpha = 0.01$, the patterns are exact as $\mathcal{C}[i][j] > \alpha\,, \forall i \neq j$.

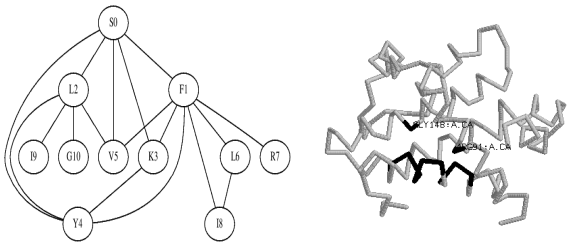
Table 7 compares the time taken to mine $K = 500$ maximal patterns from the SCOP dataset using the NL label algorithm and two other algorithms, with $minsup = 20$ and $\alpha = 1.0$. The *no pruning* algorithm computes the representative sets from the candidate representative sets directly using the verification procedure described in section 3.3. The *gApprox* algorithm is based on [5] and stores all isomorphisms during the course of enumerating a maximal pattern. It can be see that the run time for the NL based algorithm is significantly less as it prunes invalid candidates without performing an expensive verification procedure or storing a large number of isomorphisms.

Table 8 shows the time taken for $K = 500$ random walks for various values of *minsup*, with $\alpha = 1.0$. The table shows the time spent in the k-hop matching, NL matching, and pattern verification steps, and the total time. We see a similar trend compared to the CMDB graph in terms of the run time, i.e., as *minsup* increases the time also increases. From Table 9 we can see that the number of forward extensions tried increases with higher *minsup*, though the number of backward extensions tried decreases slightly. However, the increase in time with higher *minsup* is also a result of increased cost of NL label pruning and the verification steps, both in terms of time (as seen in Table 8) and in terms of the number of such checks (as seen in Table 10).

α	NL label	k-hop label	Verification	Total
0.5	119.84	35.53	13.71	169.08
		250.57	36.33	286.9
0.75	293.40	127.96	224.55	645.91
		368.84	653.14	1021.98

Table 11: SCOP: Effectiveness of labels (Time in sec)

Table 11 compares the effectiveness of NL and k-hop labels for different values of the threshold α. For each value of α, the top row shows the time with NL label, whereas the bottom row shows the time using only the k-hop label. The NL label clearly reduces the time taken. In fact, it reduces the time for both the k-hop matching and the pattern verification steps, since NL is very effective in pruning the representative set. This effect is best seen for $\alpha = 0.75$, where the total time for verification reduces even though matching the neighbors takes more time compared to k-hop matching. This shows the effectiveness of the NL label versus k-hop label in isolation.



(a) Pattern (b) Structural Motif

Figure 6: SCOP: Approximate Pattern and its Structure

Example Patterns: Figure 6 show an example approximate protein graph pattern and its corresponding 3D structure extracted from the SCOP dataset. For example, the graph in 6a has support 19, and the structure of one its occurrences, in protein PDB:1R2E, is shown in 6b. The common motif comprises the black colored amino acids some of whom are far apart in sequence but are spatially close in 3D. It is important to note that the cost of this isomorphism is $C(\phi) = 0.4541$, indicating that exact isomorphism cannot find the motif.

5.3 Protein-Protein Interaction Network (PPI)

We ran experiments on a yeast (Saccharomyces cerevisiae) PPI network. The list of interacting proteins for yeast was downloaded from the DIP database (`http://dip.doe-mbi.ucla.edu`). As seen in Table 3, the PPI network has 4950 proteins and 16,515 interactions. Unlike the other datasets, each node in the PPI network essentially has a unique label, which is the protein name. One of the differences for the PPI graph is that we do not utilize the k-hop labels. The complexity of matching the k-hop labels depends on the number of literals in the k-hop label. As each label (protein) is unique in the PPI graph, the number of literals in the k-hop label of a vertex v in a PPI network is equal to the number of vertices reachable in k-hops. This increases the run time for the k-hop label matching. Therefore, for mining PPI networks we use only the neighbor mapping component of NL labels (thus, the pre processing time for PPI is not applicable in Table 3).

Cost Matrix: To construct the cost matrix for the protein network we consider the similarity between the protein sequences for any two adjacent nodes. Sequence similarity is obtained via the BLAST alignment score (`http://blast.ncbi.nlm.nih.gov/`), that returns the expected value (E-value) of the match. A low E-value implies high similarity, thus we create a binary cost matrix between the proteins by setting $\mathcal{C}[p_i][p_j] = 0$ iff the proteins p_i and p_j have high similarity, i.e., iff $E\text{-}value(p_i, p_j) \leq \epsilon$. We empirically set $\epsilon = 0.003$. Once the binary cost matrix is constructed the algorithm is run with $\alpha = 0$ since the label mismatch is handled by the cost matrix.

minsup	Time	Avg. Time
3	834.55	1.67
5	377.56	0.76
10	254.24	0.51

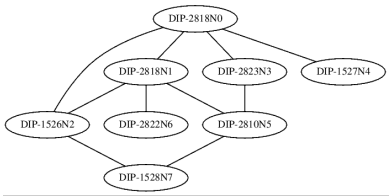
Table 12: Time (sec) for random walks in PPI Dataset

minsup	Fwd	Fwd Success	Back	Back Success
3	54.56k	0.34k	0.85k	0.025k
5	22.83k	0.11k	0.16k	0.022k
10	10.2k	0.73k	0.75k	0.09k

Table 13: PPI: Number of extensions and successes

Results: Table 12 shows the time for $K = 500$ random walks in the yeast PPI network for different values of *minsup*. It can be seen that the time for random walks decreases as the support value increases, which is opposite to the trend seen for CMDB and SCOP graphs. We verify in Table 13 that for PPI the number of forward extensions is drastically lower for higher *minsup* values.

Example Patterns: Figure 7a shows a mined maximal frequent approximate pattern (using $minsup = 5$). The proteins are labeled with their DIP identifiers (e.g., DIP-2818N); the last number in the label is just a sequential node id. It is worth emphasizing that exact subgraph isomorphism would not yield any patterns in this dataset, since each label is unique. However, since we allow a protein to be replaced by a similar protein via the cost matrix $\mathcal{C}$, we obtain interesting approximate patterns. To judge the quality of the mined patterns we use the gene ontology (GO; `www.geneontology.org`), which comprises three structured, controlled vocabularies (ontologies) that describe gene products in terms of their associated biological processes (BP), molecular functions (MF), and cellular components (CC). For each of the mined approximate patterns we obtain the



(a) Pattern

GO Terms	Description
BP:0051603	proteolysis involved in cellular protein catabolic process
MF:0004298	threonine-type endopeptidase activity
CC:0034515	proteasome storage granule

(b) Common GO Terms

Figure 7: Approximate PPI Pattern and GO Enrichment

set of all the GO terms common to all proteins in the pattern. This serves as an external validation of the mined results, since common terms imply meaningful biological relationships among the proteins. Figure 7b shows the common GO terms for the pattern in Figure 7a. This subgraph comprises proteins involved in proteolysis as the biological process, i.e., they act as enzymes that lead to the breakdown of other proteins into amino acids. Their molecular function is endopeptidase activity, i.e., breakdown of peptide bonds of non-terminal amino acids, in particular the amino acid Threonine. These proteins are located in the proteasome storage granule, and most likely comprise a protein complex (proteasome) – a molecular machine – that digests proteins into amino acids.

6. RELATED WORK

In the past, many algorithms have been proposed to mine subgraphs from a given database of graphs. These algorithms can be mainly divided in to two classes depending on how the candidate patterns are generated. Algorithms like those in [11, 12, 14] are Apriori based methods, i.e., a candidate pattern of size $k + 1$ is generated by combining two frequent graphs of size k that have a common $k - 1$ sized subgraph. Algorithms like those in [18], on the other hand, belong to the class of pattern growth algorithms in which a candidate pattern is generated by extending a frequent pattern with another edge. Sampling approaches like those proposed in [3, 9, 19] mine a representative set of maximal patterns from a database of graphs or a single graph; they are especially effective in large real-world graphs where complete graph mining is practically infeasible.

Mining subgraphs from a single graph is a related problem which is surprisingly difficult compared to mining from a database of graphs. In [15], they defined the support of a pattern in a single graph as the maximum number of edge disjoint isomorphisms, which is itself an NP-Hard problem. In [7], they proposed a definition of support based on overlapping ancestor isomorphisms. In [3], we proposed CMDBMiner to mine frequent patterns from a single large graph. Support of a pattern is defined as the maximum flow in an appropriately constructed flow network with capacities. This method estimates the support of a pattern without enumerating its isomorphisms.

There has been little work in approximate subgraph mining. In [5], they proposed *gApprox* to mine approximate frequent subgraphs. The degree of approximation between a pattern and its isomorphism includes label mismatches and missing edges. The search space is explored in a depth

first order and the support of a pattern is computed by enumerating all its isomorphisms. This approach is not feasible for large graphs with label multiplicities as there are potentially an exponential number of isomorphisms [3]. In [13], they proposed APGM to approximate frequent subgraphs from a database of graphs. The method is similar to the *gApprox* method, with the main difference being that the entire 1-hop neighborhood of the current embeddings is explored to enumerate all extensions of the frequent pattern and their corresponding embeddings, whereas *gApprox* enumerates the embeddings for a single extension in each step. However, they store the complete set of approximate embeddings of the current frequent pattern, which can be a problem. In [1], the authors proposed strategies to speed up the existing approximate mining algorithms, by limiting the number of candidates and also the number of duplicate checks performed. They assume that the underlying algorithm takes care of the label and/or edge mismatches. In [19], they proposed a randomized algorithm to mine approximate patterns from a database graphs. This method only handles edge mismatches and not label costs.

Graph querying is another problem that is related to subgraph mining. The goal is to find matches of a given query graph in a single graph or database of graphs. In [17], they proposed an indexing method to extract the approximate occurrences of a given graph query in large graph databases. In [8], they proposed a polynomial time algorithm for detecting isomorphism between spectrally distinguishable graphs. An isomorphism, if it exists, is obtained by matching the steady state vectors of Markov chains in both the graphs. The problem with the indexing approaches is that they are efficient in retrieving a single match for the query graph but fail at retrieving all matches, and thus are not suited to mine frequent patterns. Furthermore, they assume that the query is given, and thus they do not perform pattern enumeration as required in graph mining.

7. DISCUSSIONS AND CONCLUSIONS

We presented an effective approach to mine approximate frequent subgraph patterns from a single large graph database in the presence of a label cost matrix.

There are two main parameters in our method: K, the number of random walks, and α the cost threshold. The value of K is directly proportional to the number of maximal approximate patterns we desire, and is relatively easy to set. On the other hand, choosing an appropriate value of α is very important as it affects the quality of patterns mined. Depending on the application domain and the purpose of the graph mining, let t be the number of vertices in the pattern for which we allow label mismatches in the subgraph isomorphism. One reasonable value of α is $t \times IMQ$ where IMQ is the inter-quartile mean, i.e., the mean of the entries between the first quartile (25^{th} percentile) and the third quartile (75^{th} percentile) of the entries in the cost matrix arranged in sorted order. t can be chosen by first enumerating maximal patterns with $\alpha = 0$ and then computing the average size m of the maximal patterns mined from the graph. The value of t then is a fraction of the average size m. Care has to be taken not to choose a very large α as it leads to patterns of poor quality and also increases the run time significantly as can be seen in Table 6.

In terms of future work, we plan to increase the efficiency of our method by exploiting parallelism. Obviously different walks can be carried out in parallel. However, more interesting is the parallelization of the approximate isomorphism generation and label-based pruning steps, including verification. We also want to explore the idea of label based pruning for more general definitions of approximate isomorphism including edge mismatches.

8. REFERENCES

[1] N. Acosta-Mendoza, A. G. Alonso, and J. E. Medina-Pagola. On speeding up frequent approximate subgraph mining. In *LNCS Vol. 7441*, 2012.

[2] P. Anchuri, M. J. Zaki, O. Barkol, R. Bergman, Y. Felder, S. Golan, and A. Sityon. Infrastructure pattern discovery in configuration management databases via large sparse graph mining. In *11th IEEE International Conference on Data Mining*, December 2011.

[3] P. Anchuri, M. J. Zaki, O. Barkol, R. Bergman, Y. Felder, S. Golan, and A. Sityon. Graph mining for discovering infrastructure patterns in configuration management databases. *Knowledge and Information Systems*, 33(3):491–522, Dec. 2012.

[4] V. Chaoji, M. A. Hasan, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: A Novel and Effective Approach for Mining Representative Orthogonal Graph Patterns. *Statistical Analysis and Data Mining*, 1(2):67–84, June 2008.

[5] C. Chen, X. Yan, F. Zhu, and J. Han. gApprox: Mining frequent approximate patterns from a massive network. In *ICDM Conference*, 2007.

[6] J. Edmonds and E. L. Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973.

[7] M. Fiedler and C. Borgelt. Support computation for mining frequent subgraphs in a single graph. In *5th International Workshop on Mining and Learning with Graphs*, 2007.

[8] M. Gori, M. Maggini, and L. Sarti. Exact and approximate graph matching using random walks. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(7):1100–1111, july 2005.

[9] M. A. Hasan and M. J. Zaki. Output space sampling for graph patterns. *Proceedings of the VLDB Endowment*, 2(1):730–741, 2009.

[10] S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919, 2002.

[11] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM Conference*, 2003.

[12] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

[13] Y. Jia, J. Zhang, and J. Huan. An efficient graph-mining method for complicated and noisy data with real-world applications. *Knowl. Inf. Syst.*, 28(2):423–447, 2011.

[14] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM Conference*, 2001.

[15] M. Kuramochi and G. Karypis. Finding Frequent Patterns in a Large Sparse Graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[16] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[17] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE Conference*, 2008.

[18] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM Conference*, 2002.

[19] S. Zhang and J. Yang. Ram: Randomized approximate graph mining. In *SSDBM Conference*, 2008.