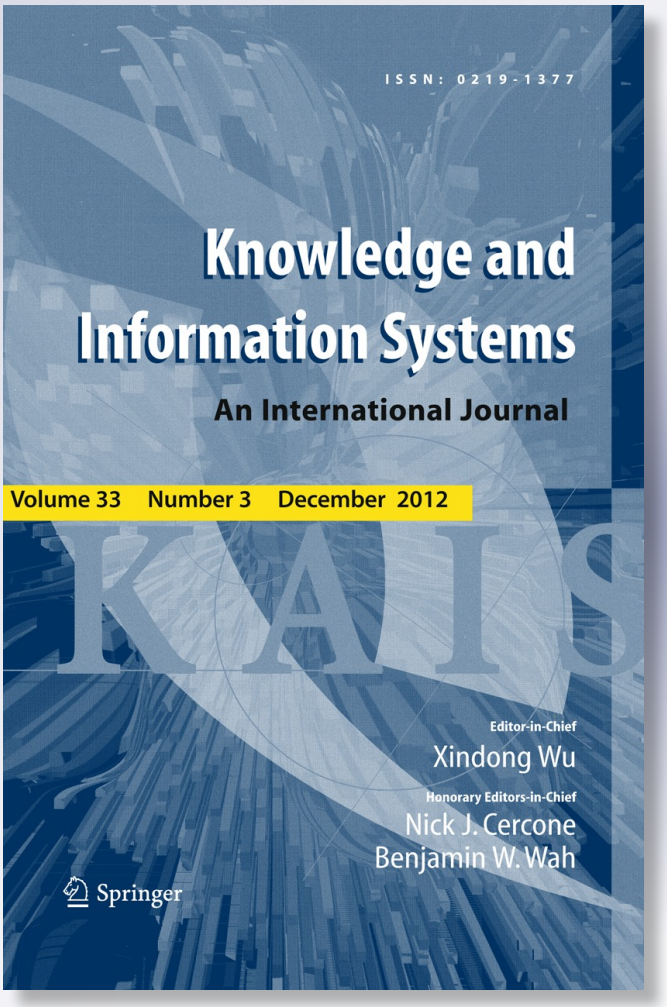
Graph mining for discovering infrastructure patterns in configuration management databases

Pranay Anchuri, Mohammed J. Zaki, Omer Barkol, Ruth Bergman, Yifat Felder, Shahar Golan & Arik Sityon

Knowledge and Information Systems
An International Journal

ISSN 0219-1377
Volume 33
Number 3

Knowl Inf Syst (2012) 33:491-522
DOI 10.1007/s10115-012-0528-3



Springer

Knowl Inf Syst (2012) 33:491–522
DOI 10.1007/s10115-012-0528-3

REGULAR PAPER

Graph mining for discovering infrastructure patterns in configuration management databases

Pranay Anchuri · Mohammed J. Zaki · Omer Barkol · Ruth Bergman · Yifat Felder · Shahar Golan · Arik Sityon

Received: 9 March 2012 / Revised: 16 April 2012 / Accepted: 14 July 2012 /
Published online: 10 August 2012

Abstract A configuration management database (CMDB) can be considered to be a large graph representing the IT infrastructure entities and their interrelationships. Mining such graphs is challenging because they are large, complex, and multi-attributed and have many repeated labels. These characteristics pose challenges for graph mining algorithms, due to the increased cost of subgraph isomorphism (for support counting) and graph isomorphism (for eliminating duplicate patterns). The notion of pattern frequency or support is also more challenging in a single graph, since it has to be defined in terms of the number of its (potentially, exponentially many) embeddings. We present *CMDB-Miner*, a novel two-step method for mining infrastructure patterns from CMDB graphs. It first samples the set of maximal frequent patterns and then clusters them to extract the representative infrastructure patterns.

This work was supported by the HP Labs Innovation Research Program Award, and in part by NSF Grant EMT-0829835.

P. Anchuri · M. J. Zaki (✉)
Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180, USA
e-mail: zaki@cs.rpi.edu

P. Anchuri
e-mail: anchupa@cs.rpi.edu

O. Barkol · R. Bergman · Y. Felder · S. Golan · A. Sityon
HP Labs, 32000 Technion City, Haifa, Israel
e-mail: omer.barkol@hp.com

R. Bergman
e-mail: ruth.bergman@hp.com

Y. Felder
e-mail: yifat.felder@hp.com

S. Golan
e-mail: shahar.golan@hp.com

A. Sityon
e-mail: arik.sityon@hp.com

We demonstrate the effectiveness of CMDB-Miner on real-world CMDB graphs, as well as synthetic graphs.

Keywords Single graph mining · Frequent subgraphs · Sparse graph mining · Configuration management databases

1 Introduction

A configuration management database (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 considered to be a *single* large multi-attributed graph, where the nodes 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). Figure 1 shows a snippet from a real-world CMDB graph, displaying only type labels. A CMDB provides a wealth of information about the largely undocumented IT practices of a large organization, and thus mining the CDMB graph for frequent subgraph patterns can reveal the de facto infrastructure patterns. Once mined, these patterns can be used to either set the default IT policies or refine them if found unsatisfactory. Thus, the discovery of infrastructure patterns is an important real-world application of subgraph mining in IT domain.

Mining a CMDB graph comes with several challenges. The CMDB graph is a massive, multi-attributed, and complex graph. There are various types and subtypes of CIs, which may be hierarchically related. CIs further have various associated attributes and metadata elements. There are a lot of repetitive labels, namely the CIs and their various attributes.

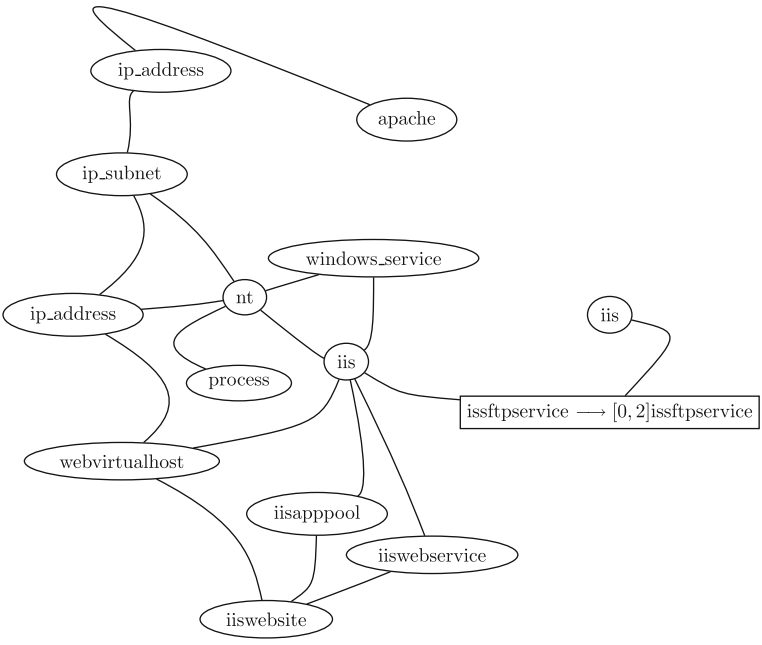


Fig. 1 Snippet from a CMDB graph

For example, there can be hundreds and thousands of running processes of the same type, running on (and thus connected to) a single server. The vast majority of frequent graph mining algorithms assume that the database consists of many different graphs, so that the support or frequency can be computed by counting how many graphs in the database *contain* a given pattern. The containment is defined in terms of subgraph isomorphism, that is, the node mapping, also called an *embedding*, corresponding to isomorphism between a pattern and some subgraph of the database graph. As long as there exists an embedding of the pattern in a database graph, the support can be incremented by one. On the other hand, the support of a pattern in a single graph usually involves finding all possible pattern embeddings (or node/edge-disjoint embeddings), which can potentially be exponential in the size and order of the graph. Furthermore, the subgraph isomorphism problem is rendered more expensive due to the repetitive CIs. Simply mining the frequent subgraph patterns from the CMDB graph is not enough to discover the infrastructure patterns. As is well known in frequent pattern mining, there can be a huge number of mined patterns, with many of them being small variations of one another, what is required is to summarize the patterns and to select only the most representative ones as the infrastructure patterns.

In this paper, we propose an effective approach to mine representative patterns from a large multi-attributed graph, with special focus on discovering infrastructure patterns from CMDB graphs. Our approach consists of two main steps: i) mining a sample of the maximal frequent subgraphs from a single large database graph and ii) clustering the mined patterns via spectral graph clustering, and extracting the representative infrastructure patterns. There are several novel contributions in this paper:

- We propose a new approach for mining/sampling maximal subgraph patterns from a single database graph. In particular, we propose a new network flow-based definition of graph support, which is an upper bound on the number of edge-disjoint embeddings and which allows us to prune patterns the moment they become infrequent. We also propose an optimization to get a tighter upper bound on the number of edge-disjoint embeddings. We further propose a fast filter-based approach for eliminating isomorphic (i.e., duplicate) patterns.
- We propose a new diffusion-based graph similarity method to compute the pairwise similarities between two labeled graphs. The method takes into account both the structure and labels of the graphs. We demonstrate that this method is better compared to existing graph matching algorithms. Given the pairwise similarity matrix, we use spectral graph clustering to extract groups of related patterns. We then select the representative patterns via a coverage-based approach.

We evaluate our approach on several real-world CMDB graphs with millions of nodes and edges, as well as on synthetic graphs, to demonstrate that our method, called *CMDB-Miner*, can find meaningful IT infrastructure patterns. Even though our focus is on CMDB graphs, our approach is generic and can be applied in many other real-world applications with similar characteristics, namely single large graph database, multiple attributes on the nodes and many repeated labels.

2 Background

2.1 Preliminary concepts

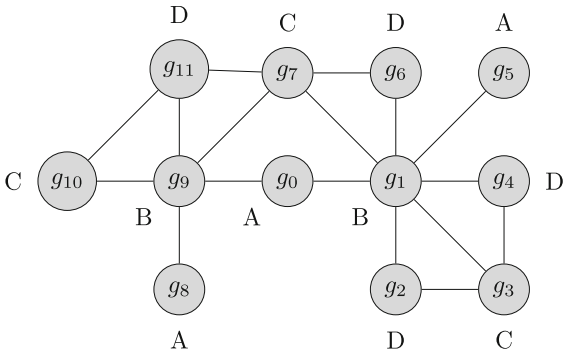
Let Σ denote a given set of labels. A *labeled graph* is a triple $G = (V, E, L)$, where V is the set of vertices or nodes, $E \subseteq V \times V$ is the set of (unordered) edges, and L is the labeling

function for both nodes and edges, so that $L(v)$ is the label of a node v, and $L(e) = L(a, b)$ is the label of an edge $e = (a, b)$. The origin and destination of an edge $e(a, b)$ are a and b, respectively. The *order* of the graph is the number of nodes $|V|$, and the *size* of the graph is the number of edges $|E|$.

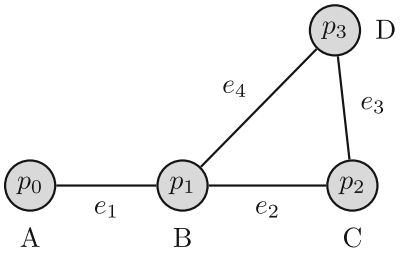
We say that $G' = (V', E', L')$ is a *subgraph* of $G = (V, E, L)$, denoted $G' \subseteq G$, if there exists a 1–1 mapping $\pi : V' \to V$, such that $(v_i, v_j) \in E'$ implies $(\pi(v_i), \pi(v_j)) \in E$. Further, π must preserve vertex and edge labels, that is, $L'(v_i) = L(\pi(v_i))$ for all $v_i \in V'$, and $L'(v_i, v_j) = L(\pi(v_i), \pi(v_j))$ for all edges $(v_i, v_j) \in E'$. The mapping π is called a *subgraph isomorphism* from G' to G. If $G' \subseteq G$, we also say that G *contains* G'. If $G' \subseteq G$ and $G \subseteq G'$, we say that G and G' are *isomorphic*.

Let $G = (V, E, L)$ be a single large database graph, and let $P = (V', E', L')$ be a *candidate* pattern, whose support we want to compute. Let π be a subgraph isomorphism from P to G. The sequence $\pi(v_1), \pi(v_2), \ldots, \pi(v_n)$ over all $v_i \in V'$ is called an *embedding* of G' in G. For an edge $e_i = (a_i, b_i) \in E'$, define $\pi(e_i) = \pi(a_i, b_i) = (\pi(a_i), \pi(b_i)) \in E$. The sequence $\pi(E') = \pi(e_1), \pi(e_2), \ldots, \pi(e_m)$ over all edges $e_i \in E'$ is called an *edge mapping* of P in G. For example, given the database graph G in Fig. 2a, and the candidate pattern P in Fig. 2b, the subgraph isomorphism π_3 from P to G specified by the mapping $p_0 \to g_0$, $p_1 \to g_9$, $p_2 \to g_7$, $p_3 \to g_{11}$ corresponds to the embedding 0, 9, 7, 11, and the edge mapping (0, 9), (9, 7), (7, 11), (11, 9). Since π uniquely specifies the embedding and edge mapping, we use these terms interchangeably.

There are several ways to compute the number of occurrences, called the *support*, of P in G. The most straightforward definition is to define the support of P as the number of possible embeddings of P in G, denoted $sup_a(P)$. Figure 2c shows all the possible embeddings of P in G. There are ten embeddings of P in G for this example, thus $sup_a(P) = 10$. Unfortunately, there can be exponentially many embeddings of a pattern in the database graph. For example, if $G = P = K_n$, where K_n is the complete graph on n nodes, with all node and edge labels being the same, then there are $n!$ distinct embeddings of P in G. Unfortunately, due to the label multiplicities in the CMDB graphs, this is a real problem in this application. To avoid the combinatorial blowup, support can also be defined as the maximum number of node or edge-disjoint embeddings of P in G, denoted $sup_n(P)$ and $sup_e(P)$, respectively. Let Π be the set of all possible embeddings of P in G. We say that two embeddings $\pi, \pi' \in \Pi$ are *node disjoint*, if $\pi(v_i) \neq \pi'(v_j)$ for all nodes $v_i, v_j \in V'$. We say that π and π' are *edge disjoint* if $(\pi(v_i), \pi(v_j)) \neq (\pi'(v_a), \pi'(v_b))$ for all edges $(v_i, v_j), (v_a, v_b) \in E'$. Figure 2d, e show examples of a maximum set of edge- and node-disjoint embeddings, respectively. These sets are not unique; for example, the embedding set $\{\pi_0, \pi_3, \pi_9\}$ is also edge disjoint. However, the edge-disjoint support of P is $sup_e(P) = 3$, and the node-disjoint support is $sup_n(P) = 2$. Finding the maximum number of edge (or node) disjoint embeddings is equivalent to finding the maximum independent set (MIS) in an *embeddings graph*, where each embedding is a node, and there exists an edge between two embeddings if they share an edge (or node). Unfortunately, the MIS problem is known to be NP-hard, and thus, both the edge- and node-disjoint embeddings are expensive to compute. One of the novel contributions of this paper is that we approximate the edge-disjoint support via a network flow-based approach. We prefer edge disjointness, since node disjointness is more constrained (every node-disjoint embedding is also an edge-disjoint embedding, but not vice versa).



(a)



(b)

π_0	$\{0, 1, 3, 2\}$
π_1	$\{0, 1, 3, 4\}$
π_2	$\{0, 1, 7, 6\}$
π_3	$\{0, 9, 7, 11\}$
π_4	$\{0, 9, 10, 11\}$

π_5	$\{5, 1, 3, 2\}$
π_6	$\{5, 1, 3, 4\}$
π_7	$\{5, 1, 7, 6\}$
π_8	$\{8, 9, 7, 11\}$
π_9	$\{8, 9, 10, 11\}$

(c)

π_1	$\{0, 1, 3, 4\}$
π_4	$\{0, 9, 10, 11\}$
π_7	$\{5, 1, 7, 6\}$

(d)

π_0	$\{0, 1, 3, 2\}$
π_8	$\{8, 9, 7, 11\}$

(e)

Fig. 2 **a** A database graph. **b** A pattern graph. **c** All embeddings of P. **d, e** Edge- and node-disjoint embeddings of P

2.2 Related work

Many different methods have been developed for frequent subgraph mining [9,18,19,23,33]. Recently, methods that sample and summarize subgraph patterns have gained more traction [8,10,1,26,34]. However, these methods assume that the database contains many different graphs and cannot be directly applied when the database is just a single large graph. This is because they define pattern support to be the number of graphs in the database that contain the pattern. As long as a single embedding is found, the support can be incremented by one, and as such these methods do not have to deal with the problem of enumerating all the embeddings, or computing the maximum number of edge (or node) disjoint embeddings.

Also, pattern support, as defined for a database of many graphs, is *anti-monotonic*, that is, a supergraph cannot have support more than any of its subgraphs. This property allows for fast pruning of candidate patterns during pattern search, since we can prune a pattern (and all of its extensions), when its support falls below a user-specified minimum support threshold, $minsup$. However, the number of embeddings is clearly not anti-monotonic. For example, let $minsup = 3$, and let the database graph comprise a node labeled A, connected to two nodes labeled B, and further, let each of the B nodes be connected to three nodes labeled C. In this database graph, the edge A–B has two embeddings (below $minsup$), but the pattern A–B–C has six embeddings (above $minsup$). The lack of anti-monotonicity is clearly a problem for support computation.

2.2.1 Support in a single graph

Several recent approaches have been proposed to tackle the challenges in mining a single graph. Kuramochi and Karypis [24] proposed a support counting measure that is anti-monotonic. They proposed three different formulations for mining a single graph. The first is based on an exact MIS of the overlap graph, which gives the exact set of edge-disjoint embeddings. The other two approaches are based on approximate MIS, which provide a subset and superset of the edge-disjoint embeddings. However, they require enumerating all the embeddings and then discarding the ones that overlap. Since the total number of possible embeddings is exponential, it makes these methods incapable of finding bigger patterns. Further, instead of the MIS, we propose a network flow-based approach. Fiedler and Borgelt [16] give a formal proof that MIS-based support counting is anti-monotonic. We also prove that our flow-based upper bound on the number of edge-disjoint embeddings leads to an anti-monotonic pruning criteria. Bringmann and Nijssen [5] proposed an *image-based* support of a pattern, defined as the minimum number of mappings from a vertex in the pattern to a vertex in the graph. Our flow-based approach yields a tighter upper bound compared to the image-based support. Chen et al. [11] proposed a *gApprox* to mine frequent approximate patterns from a single large network. They defined the pattern space and proposed strategies to explore it. The support of the pattern is same as the image-based support and requires all the approximate embeddings of the pattern. Li et al. [25] propose a method to compute edge-disjoint support to find frequent dense subgraphs in a single graph. This method is not suitable for CMDB graphs, since infrastructure patterns are not very dense. Besemann and Denton [3] tackle graphs in which nodes have multiple attributes. The edge-disjoint support is computed by constructing a bipartite graph with the original node set (V) and a node for every attribute (U). Edge disjointness is imposed on V, allowing for overlap in the bipartite edges that connect a vertex in V to one of its attributes in U. Recent theoretical work has focused on proving necessary and sufficient conditions for anti-monotonicity of edge overlap based graph support measures [32], and in other generalizations such as homomorphisms and isomorphisms, for labeled and unlabeled, directed and undirected graphs [7].

3 CMDB-Miner: mining CMDB graphs

CMDB-Miner has three main steps. Given the particular characteristics of CMDB graphs, first we preprocess them to extract the relevant attributes for each configuration item and summarize the graph. Second, we perform random walks in the pattern space to extract a sample of the maximal frequent patterns. In the third step, we cluster the maximal patterns (since many of them may be similar) and extract a set of representative patterns from each

Table 1 IP address-related attributes and their values

DNS Server	NetMask	IP address
192.168.1.1	255.255.255.0	192.168.1.2
192.168.1.1	255.255.0.0	192.168.1.3
192.168.1.1	255.255.0.0	192.168.1.4
192.168.1.1	255.255.255.0	192.168.1.5
192.168.1.1	255.255.0.0	192.168.1.6

Table 2 Attribute entropies

Attribute	Entropy
DNS Server	$\log_2(1) = 0.0$
NetMask	$\frac{2}{5}\log_2\left(\frac{2}{5}\right) + \frac{3}{5}\log_2\left(\frac{3}{5}\right) = 0.971$
IP address	$\log_2\left(\frac{1}{5}\right) = 2.322$

cluster. The latter constitute the infrastructure patterns presented to the IT practitioners to help manage and set the IT configuration policies throughout the organization. The details of each step are given below.

3.1 Graph preprocessing

CMDB graphs have many different types of composite items, and each CI may have many possible attributes (with various values). Furthermore, there are many degree of one nodes, called *leaf nodes* in the CMDB graph. Before mining these graphs, we preprocess them in two ways to aid in interpretation and mining. First, we prune attributes based on their entropy, and second, we summarize the multiplicities among the leaf nodes.

3.1.1 Entropy-based attribute pruning

Based on the distribution of values for each attribute, we observed that across the various instances of the same CI type, some of the attributes either have a single value, or they have all distinct values. Let $p_v = \frac{m_v}{m}$ be the probability of observing value v for an attribute a of a given CI type, where m is the total number of occurrences of attribute a, and m_v is the number of times a has value v. The entropy of a is defined as $E(a) = -\sum_v p_v \log_2 p_v$. We prune the uninformative attributes, namely those that have very low or very high entropy, by discarding the tails of the entropy distribution (e.g., discarding attributes within the bottom 5 % and top 5 % of entropy).

This results in a significant reduction in the number of attributes. Table 1 shows instances of an IP address CI, with three attributes. The DNS Server has the same values, whereas the IP address has all distinct values across the instances. The entropy of these attributes is shown in Table 2. DNS Server, and IP ddress will both be pruned, since the former has very low and the latter very high entropy.

3.1.2 Summarizing leaf nodes

A peculiarity of CMDB graphs is that a vast majority of nodes are *leaf nodes*, defined as those with degree one. Further, an *internal node*, defined as a node with degree more than one, can be connected to many of the same types of leaf nodes, a characteristic we call

node multiplicity. Some of the CI types like *process*, *ip_address*, etc., have a wide range of multiplicities. CMDB-Miner employs leaf level summarization, which reduces the size of the CMDB graphs significantly, and aids interpretation of the mined infrastructure patterns. For each leaf node u with CI type t, we define its same label siblings as: $Sib(u,t) = \{x | L(x) = t, x \text{is a leaf node}, x \text{and} u \text{have common neighbor}\}$. For every CI type t, we define its multiplicities as: $Mult(t) = \{\ m\ |\ \exists\ u,\ u \text{ is a leaf}, L(u) = t,\ |Sib(u,t)| = m\}$. In other words, the multiplicities of CI type t, is the multiset comprising the number of its occurrences at the leaf level with common internal neighbors. We discretize the multiplicities $Mult(t)$ using equi-width binning. For each internal node connected to leaf nodes, we then attach a new label of the form $x \longrightarrow [l,u]y$, which is interpreted as internal node x having between l and u occurrences of CI type y as a leaf. Figure 1 shows an example of such a label, namely *issftpservice* $\longrightarrow$ [0, 2] *issftpservice*, meaning that *issftpservice* is connected to up to two other leaf nodes with CI type *issftpservice*.

3.2 Sampling maximal patterns

The goal of this step is to extract a sample of maximal frequent subgraphs from a large, sparse CMDB graph. We do this via random walks in the pattern space, starting from the empty graph and extending the current candidate pattern by a random edge. After each extension, we ensure that the pattern is frequent, according to our new network flow-based approach (described below). Thus, random pattern extension and support computation form the two substeps for each candidate.

3.2.1 Random walks in pattern space

CMDB-Miner takes as input a parameter k, specifying the number of walks to perform. Each random walk begins with the empty graph and extends the patterns via random edge extensions. If a random extension yields a frequent pattern, based on the flow-based support described below, it is accepted. Otherwise, the extension is rejected, and we try another random extension. If none of the possible extensions yield a frequent pattern, we are guaranteed that the current pattern is maximal, and we add it to the set of maximal patterns M. It is important to note that, unlike other graph mining approaches that check for isomorphism during pattern growth (to eliminate duplicates), CMDB-Miner does not check for isomorphism until all the k walks finish. This way we pay the price for isomorphism only for patterns that are maximal, and not at each extension. This strategy confers significant efficiency.

Within a given walk, we assume that the edges (and nodes) in P are numbered in the order in which they are added to generate P, starting from an empty graph. Edge ordering automatically leads to node ordering as well. For example, Fig. 2a shows a database graph G, and Fig. 2b shows a candidate pattern graph P. $e_1 = (p_0, p_1)$ is the first, $e_2 = (p_1, p_2)$ is the second, and $e_3 = (p_2, p_3)$ is the third edge to be added to P. All of these are examples of *forward* edges, that is, an edge that introduces at least one new node to P. The nodes are ordered from p_0 to p_3. Due to node ordering, a forward edge is implicitly directed from a lower to a higher numbered node. The last edge to be added to complete P is $e_4 = (p_3, p_1)$ and is an example of a *backward* edge, defined to be an edge between existing nodes. All of these steps in the random walk are shown in Fig. 3. A backward edge is implicitly directed from a higher to lower numbered node. This direction information is used in our flow-based support detailed next.

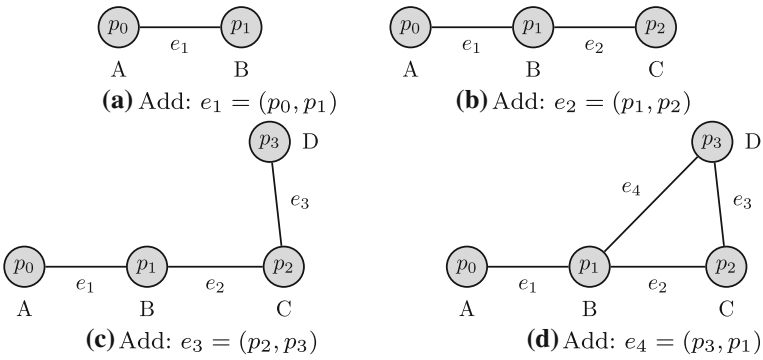


Fig. 3 Steps in the random walk leading to pattern P. **a** Add: $e_1 = (p_0, p_1)$. **b** Add: $e_2 = (p_1, p_2)$. **c** Add: $e_3 = (p_2, p_3)$. **d** Add: $e_4 = (p_3, p_1)$

3.2.2 Network flow-based pattern support

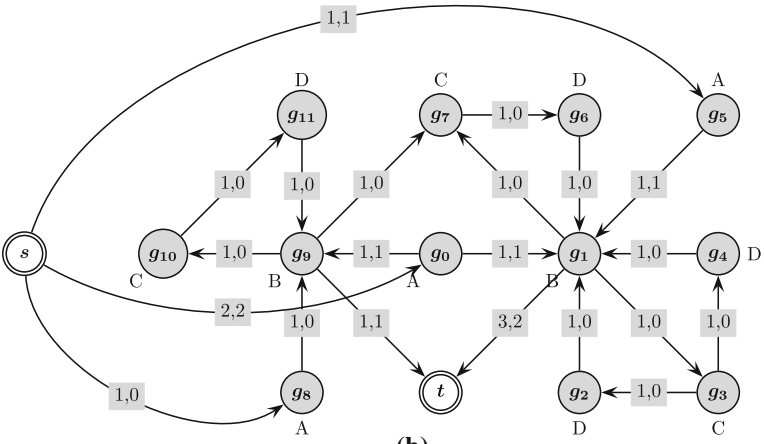
Recall that a *flow network* $G = (V, E)$ is a *directed graph* with two distinguished vertices—source s and sink t. Every ordered edge $(u, v) \in E$ has a capacity $c(u, v) \geq 0$. A flow in this network is a function $f : E \to \mathbb{R}$ that satisfies the following properties: i) capacity constraint: $f(u, v) \leq c(u, v)$, and ii) flow conservation: $\sum_{u \in V} f(u, v) - \sum_{u \in V} f(v, u) = 0$, for all $v \in V \setminus \{s, t\}$. The *value* of a flow is defined as $|f| = \sum_{v \in V} f(s, v)$, and *maximum flow* is a flow with the maximum value. It is known that if all the edge capacities $c(u, v)$ are integers, then there exists a maximum flow with only integer flows on the edges. A *path* from node u to v in a flow network $G = (V, E)$ is a sequence of *distinct* vertices $(v_1, v_2, \ldots, v_k)$ such that $u = v_1$, $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq k - 1$, and $v_k = v$. The *length* of this path is $k - 1$. A path from s to t is also called a s–t path.

We now describe the construction of a flow network in which the maximum flow corresponds to an upper bound on the edge-disjoint support of the pattern. The main idea is that any embedding of a pattern can be viewed as a path from s to t in the flow network, and edge disjointness can be imposed by using unit capacities on the edges.

Consider a pattern $P = (V', E', L')$, and a database graph $G = (V, E, L)$. Let $E' = \{e_1, e_2, \ldots, e_m\}$ be an ordering of the edges in P (e.g., the order in which pattern P was obtained). Recall that each edge is oriented, that is, it is a forward or backward edge. Let Π_i denote the set of all embeddings in G for a single edge $e_i \in E'$. For example, Fig. 4a shows the embeddings for each oriented edge in P. The flow network $F = (V_F, E_F)$ is constructed from the set of embeddings by setting V_F to be the set of distinct nodes over all the edge embeddings Π_i and by adding the directed edge (a_j, b_j), with capacity $c(a_j, b_j) = 1$, for each embedding $a_j, b_j \in \Pi_i$, with $1 \leq j \leq |\Pi_i|$ and $1 \leq i \leq m$. Further, we add an edge (s, u) for each distinct u such that $(u, v) \in \Pi_1$, with capacity $c(s, u) = n_u$, where n_u is the number of time node u appears in Π_1. Finally, we add an edge (v, t) for each distinct v such that $(u, v) \in \Pi_m$, with capacity $c(v, t) = n_v$, where n_v is the number of times v appears in Π_m. Figure 4b shows the flow network obtained from the embeddings of each edge in P. For instance, since $0, 1 \in \Pi_1$, we add the edge $(0, 1)$ in F with capacity 1. Likewise, since $2, 1 \in \Pi_4$, we add the edge $(2, 1)$ in F with capacity 1. The same is done for all embeddings in Π_j, for $1 \leq j \leq 4$. There are three distinct start nodes in Π_1, namely $\{0, 5, 8\}$; thus, we add three edges from the source: $(s, 0)$ with capacity 2, $(s, 5)$ with capacity 1, and $(s, 8)$ with

A–B $e_1(p_0, p_1)$	B–C $e_2(p_1, p_2)$	C–D $e_3(p_2, p_3)$	D–B $e_4(p_3, p_1)$
Π_1	Π_2	Π_3	Π_4
$0, 1$	$1, 3$	$3, 2$	$2, 1$
$0, 9$	$1, 7$	$3, 4$	$4, 1$
$5, 1$	$9, 7$	$7, 6$	$6, 1$
$8, 9$	$9, 10$	$7, 11$	$11, 9$
		$10, 11$	

(a)



(b)

A–B, B–C, C–D, D–B (p_0, p_1), (p_1, p_2), (p_2, p_3), (p_3, p_1)
$(5, 1)$, $(1, 7)$, $(7, 6)$, $(6, 1)$
$(0, 1)$, $(1, 3)$, $(3, 4)$, $(4, 1)$
$(0, 9)$, $(9, 10)$, $(10, 11)$, $(11, 9)$

(c)

Fig. 4 Flow network and maximum flow: **a** Edge embeddings for P. **b** Flow network for P. *Boxes* show capacity and flow on each edge. Maximum flow has value 3. **c** A possible set of three edge-disjoint embeddings corresponding to the maximum flow of 3

capacity 1. Finally, there are two distinct end nodes in Π_4, namely $\{1, 9\}$; thus, we add two edges to the sink: $(1, t)$ with capacity 3, and $(9, t)$ with capacity 1.

Definition 1 The flow-based support of a pattern P, denoted $sup_f(P)$, is defined as the maximum flow in the flow network for P.

There are several efficient implementations of maximum flow. We use Dinic's algorithm [15] that is based on blocking flows. In special cases where all the edges have a unit capacity, it has complexity $O(\min(V^{2/3}, E^{1/2}) \cdot E)$. Figure 4b shows that the maximum

flow value is 3, thus $sup_f(P) = 3$. Figure 4c shows the three disjoint edge mappings for P, corresponding to the three disjoint embeddings in Fig. 2d.

We now prove that the flow-based support of P is an upper bound on the edge-disjoint support. Let $G = (V, E, L)$, and $P = (V', E', L')$ with $|V'| = n$ and $|E'| = m$. We make the following observations:

- LEMMA 1: If π is an embedding of P in G, then there exists a corresponding s–t path in the flow network F. This follows immediately from the facts that: i) P is connected; ii) for each edge $e_i = (a_i, b_i) \in E'$, there is an edge in the flow network corresponding to the edge mapping for e_i, namely $\pi(e_i) = (\pi(a_i), \pi(b_i)) \in \Pi_i$; iii) there exists an edge from s to each start node in Π_1 (for edge $e_1 \in P$), and from each end node in Π_m (for $e_m \in P$) to t. Note that the path length can be less than $m + 2$. It is $m + 2$ when all edges in P lie on some path from s to t.
- LEMMA 2: If π_1, and π_2 are two edge disjoint embeddings of P in G, then the s–t corresponding paths are disjoint, ignoring the out-edges of s and in-edges of t (which may be shared).
- LEMMA 3: If $\Pi = \{\pi_1, \pi_2, \ldots, \pi_k\}$ is a set of edge-disjoint embeddings of P in G, then the maximum flow is at least k. Let n_u embeddings have the same start vertex u, and let n_v embeddings have the same end vertex v. From Lemma 2, we know that ignoring s and t, there are k disjoint paths in the flow network k, corresponding to each of the k embeddings in Π. If $f(e) = 1$ for all edges e on these paths, and if $f(s, u) = n_u$ and $f(v, t) = n_v$, then we can see that the resulting flow has value at least k.

Theorem 1 *The maximum flow in the flow network F for pattern P is an upper bound on the number of edge-disjoint embeddings of P in G.*

Proof In Lemma 3, if Π is the set of all possible edge-disjoint embeddings of P, with $|\Pi| = k$, then the maximum flow in F is at least k, and $sup_f(P) \geq k = sup_e(P)$. Let Q be an extension of pattern P, that is, $P \subseteq Q$. Since every edge disjoint embedding of Q is also an edge-disjoint embedding of P, it immediately implies that $sup_e(Q) \leq sup_e(P) \leq sup_f(P)$. □

The fact that $sup_f(P)$ is an upper bound on the edge-disjoint support allows us to prune any extension (an immediate supergraph) of pattern P if $sup_f(P) < minsup$. This follow immediately from the theorem above, since $sup_f(P) < minsup \implies sup_e(Q) < minsup$, and thus, we can guarantee that no extension of P can be frequent according to edge-disjoint support. We will call the flow network constructed using the embeddings of all the edges in the network as the complete flow (CF) network.

It is worth noting that the edge-disjoint support of P is equal to the maximum number of edge-disjoint s–t paths of length $m + 2$ in the flow network. However, [20] proved that finding the maximum disjoint paths with constraints on the length is NP-Complete. For this reason, our formulation does not place any restrictions on the length of the paths, and thus, we obtain an upper bound on the edge-disjoint support. It is important to note that Dinic's algorithm finds the shortest s–t paths that are saturated. Thus, the flow-based support is close to the actual support if the shortest s–t path length in the flow network is close to the number of edges in the candidate pattern. While it may not be a beneficial strategy for general (dense) patterns, our formulation is very effective for CMDB graphs, which are sparse, and thus, the mined patterns are also sparse. For such patterns, the flow-based support is generally close to the edge-disjoint support.

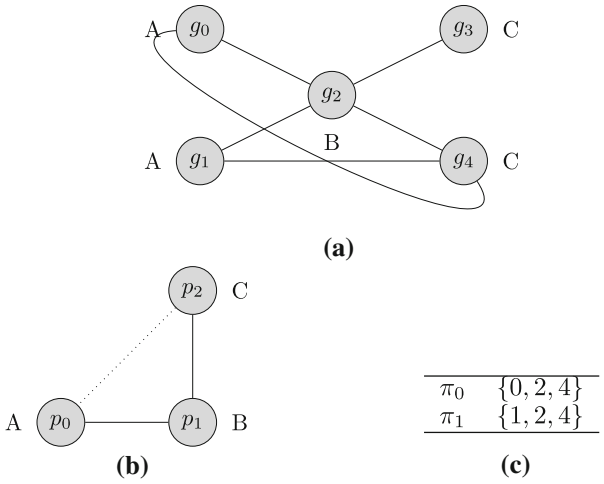


Fig. 5 **a** A database graph. **b** A candidate pattern obtained by adding a back edge between p_2 and p_0. **c** All embeddings of P. The embeddings π_0 and π_1 of P in G are not edge disjoint

4 Optimization

The upper bound on the edge-disjoint support calculated using the flow network formulation of a candidate pattern may be loose if the length of the shortest path between s and t is smaller compared to the number of edges in the candidate pattern. This situation arises when the candidate pattern is generated as a result of adding a back edge to a vertex close to p_0 or by branching from a vertex close to p_0. In the extreme case when the back edge is connected to p_0 or the branch extends from p_o, any flow that originates at s can reach t without any restriction, that is, the capacity constraint on the internal edges may have no effect.

Consider the sample database shown in Fig. 5a, and suppose that $e1(p_0, p_1)$ and $e2(p_1, p_2)$ are the two edges that are added to an empty pattern leading to a frequent pattern P'. In the next step of the random walk, a back edge is added from p_2 to p_0 leading to the candidate pattern P shown in Fig. 5b. Figure 5c shows all the embeddings for the candidate pattern P. It can be seen that the maximum number of edge-disjoint embeddings for P is 1 as the edge mapping of e_2 is (2, 4), and it is in both π_0 and π_1. Figure 6 shows the edge mappings and the resulting flow network for P. The maximum s–t flow of 2 in the network is obtained by pushing a unit flow on the paths s–g_0–t and s–g_1–t. Notice that there are no internal edges on these paths. In this example, if the flow network is constructed using only edge mappings of $e'_3(p_0, p_2)$ and $e'_2(p_2, p_1)$, then the maximum flow is 1. Figure 7 shows this alternative. This construction separates the s' and t' nodes by at least two edges, and hence, the capacity constraint on the edges restricts the maximum flow to 1.

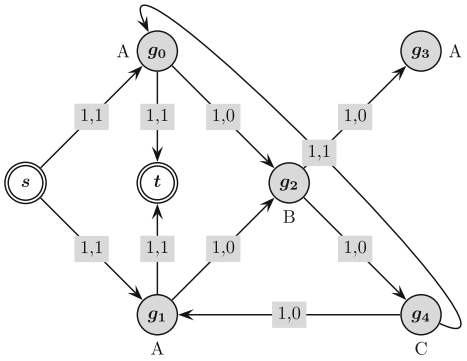
4.1 Longest path network

The key take away from the above example is that flow can be restricted by separating s and t in the network. Therefore, to obtain a better upper bound on the edge-disjoint support, we compute the maximum flow in a network where s and t are most separated, but the support upper bound is still guaranteed. We refer to this optimization as the Longest path (LP) network and contrast it with the CF network given above.

Let $P' = (V', E')$ be a frequent pattern and $P = (V, E)$ be the pattern obtained by adding the edge $e_k = (v_i, v_j)$. Note that $v_j \notin V'$ if e_k is a forward edge. To construct the LP network

A–B $e_1(p_0, p_1)$	B–C $e_2(p_1, p_2)$	C–A $e_3(p_2, p_0)$
Π_1	Π_2	Π_3
$0, 2$	$2, 3$	$4, 0$
$1, 2$	$2, 4$	$4, 1$

(a)



(b)

Fig. 6 Flow network and maximum flow: **a** Edge embeddings for P. **b** Flow network for P. *Boxes* show capacity and flow on each edge. Maximum flow has value 2

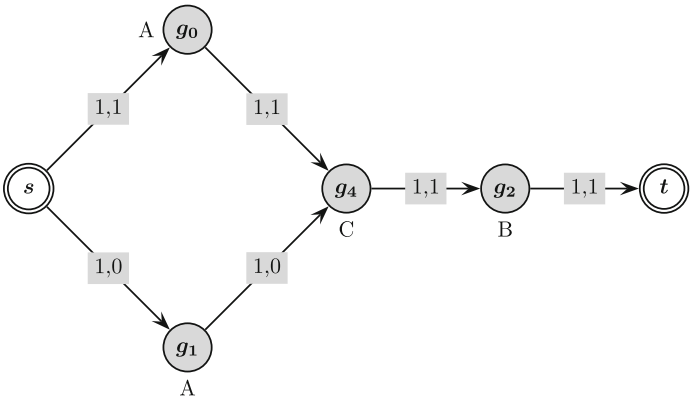
of P, we first find a vertex $v_s \in V'$ which is farthest from v_j, with S denoting the LP, which starts at v_j and ends at v_s. We place an additional constraint that the first edge on S is (v_j, v_i). The LP network then is the same as the flow network of a pattern $Q = (V'', E'') \subseteq P$ where E'' is the set of all edges present on the path S. The source node s is connected to the edge mappings of the first edge on S, and the mappings of the last edge on S are connected to the sink t in the LP network. As before, the maximum flow in the LP network is an upper bound on the edge-disjoint support. There is a twofold advantage to using the maximum flow in the LP network: 1) Q does not have a cycle and is branch free, which implies that the s–t flow is more constrained. 2) The size of the LP network is smaller compared to the size of flow network for P. This reduces the time required to compute the maximum flow. We will now prove the upper bound property of the LP network. The argument for the proof is similar to Theorem 1. We continue to use the notation introduced above and make the following observations about the LP network.

– LEMMA 4: If π is an embedding of P in G, then there exists a corresponding s–t path in the LP network. This follows immediately from the following facts: i) Q is connected; ii) for each edge $e_i = (a_i, b_i) \in E''$, there is an edge in the LP network corresponding to the edge mapping for e_i, namely $\pi(e_i) = (\pi(a_i), \pi(b_i)) \in \Pi_i$; iii) there exists an edge from s to each start node in the edge mappings of (v_j, v_i), and to t from each end node in the mappings of the last edge on the path S.

Given the existence of s–t path, the rest of the proof is similar to the proof of Theorem 1.

A–C	C–B
$e'_1(p_0, p_2)$	$e'_2(p_2, p_1)$
Π_1	Π_2
$0, 4$	$3, 2$
$1, 4$	$4, 2$

(a)



(b)

Fig. 7 Flow network and maximum flow: **a** Edge embeddings for the edges on the longest path. **b** Flow network for edges on the path. *Boxes* show capacity and flow on each edge. Maximum flow has value 1. Note that g_3 has been omitted from the network as it not connected to any embedding in Π_1

4.2 Pruning isomorphic patterns

Given a minimum support threshold *minsup*, and given k, the number of random walks, CMDB-Miner performs k random walks in the pattern space, to yield a set M of exactly k maximal frequent subgraphs, using flow-based support. However, since the walks are random, they may yield isomorphic maximal patterns. Such isomorphic patterns have to be discarded before the infrastructure pattern extraction step. Unfortunately, while graph isomorphism is in NP, it is not known whether it is NP-complete or is in P [14].

Instead of checking for isomorphism between every pair of maximal patterns in M, we use a sequence of polynomial-time filters to create equivalence classes of possibly isomorphic patterns. Thus, the worst-case exponential time algorithm for graph isomorphism has to be applied to only pairs of graphs within the same equivalence class. Initially, M comprises a single equivalence class. We then apply the following filters:

- NODE MULTISET: Given a pattern $P = (V', E', L')$, define $\rho_V(P) = \{L(v_i) : v_i \in V\}$ to be the *multiset* of node labels in P. It is easy to see that two patterns P and P' cannot be isomorphic if $\rho_V(P) \neq \rho_V(P')$. In this case, P and P' are added to different equivalence classes and never have to be checked for isomorphism.
- EDGE MULTISET: Given pattern $P = (V', E', L')$, for each edge $e_i = (a_i, b_i) \in E'$, define a *composite edge label* to be the triple $\mathcal{L}(e_i) = (L'(a_i), L'(b_i), L'(e_i))$, with $a_i < b_i$. Define the filter $\rho_E(P) = \{\mathcal{L}(e_i) : e_i \in E'\}$ to be the *multiset* of composite edge labels for P. Two patterns P and P' cannot be isomorphic if $\rho_E(P) \neq \rho_E(P')$.

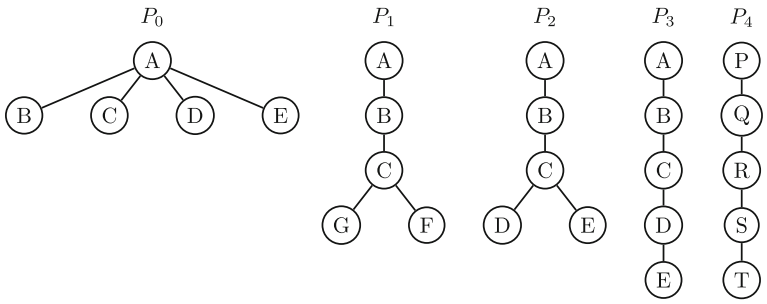


Fig. 8 Sample maximal patterns

– LAPLACIAN SPECTRUM: Let A be the adjacency matrix for pattern P, that is, $A(v_i, v_j) = 1$ if $(v_i, v_j) \in E'$, and $A(v_i, v_j) = 0$, otherwise. Let D be the diagonal degree matrix for P, defined as $D(v_i, v_i) = \sum_{v_j} A(v_i, v_j)$, and $D(v_i, v_j) = 0$ for all $v_i \neq v_j$. Define the *normalized Laplacian matrix* of P as follows: $N = D^{-1/2} \cdot (D - A) \cdot D^{1/2}$. N is a $n \times n$ positive semi-definite matrix, and thus, N has n (not necessarily distinct) real, positive eigenvalues: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$. Define the Laplacian spectrum of P as the multiset $\rho_S(P) = \{\lambda_i : 1 \leq i \leq n\}$. It is known that two isomorphic patterns are iso-spectral, that is, they have the same Laplacian spectrum [14]. Thus, P and P' cannot be isomorphic if $\rho_S(P) \neq \rho_S(P')$

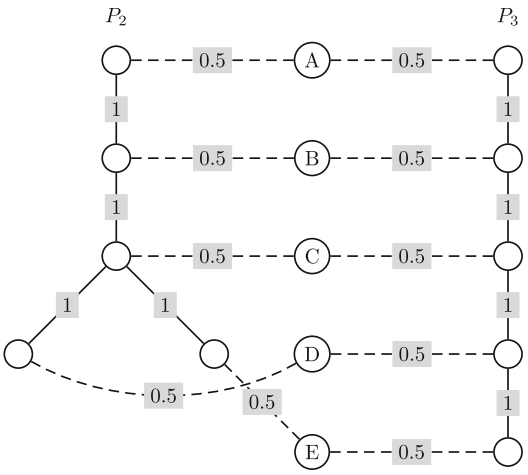
After applying the above filters, the set M is partitioned into smaller equivalence classes of possibly isomorphic graphs. For each pair of graphs in the same class, we perform full isomorphism checking using the VF2 [13] algorithm. The output of this step is the final set M of non-isomorphic maximal frequent patterns in G. Note that at this stage, we can find the actual edge disjoint support of all the maximal patterns by using the maximal independent set approach proposed in [24].

5 Infrastructure pattern extraction

Given a set of non-isomorphic maximal patterns M, CMDB-Miner clusters them into groups of similar patterns and then selects a representative set of infrastructure patterns from each cluster. There are three main steps: i) defining pairwise similarities between patterns, ii) graph clustering, iii) infrastructure pattern extraction.

5.1 Pattern similarity

Before clustering the maximal patterns, we have to define a similarity measure between patterns, which takes into account both the structure and label information. Graph edit distance-based methods [6] are a popular approach to compute the similarity; however, the vast majority of these methods focus mainly on the structure. For example, a purely structure-based method would consider P_3 and P_4 in Fig. 8 to be highly similar. Methods that consider labels include [17,28]. We propose a novel pattern similarity approach based on diffusion kernels [22], which works well for CMDB graphs. As such, the clustering method is independent of the similarity measure, and thus any of the attributed graph similarity measures can also be used. For instance, we compare our method with the similarity flooding method [27] in the experimental section.



(a)

↘	P_0	P_1	P_2	P_3	P_4
P_0	**0.47**	0.10	0.26	0.25	0.0
P_1	0.10	**0.40**	0.06	0.09	0.0
P_2	0.26	0.06	**0.40**	0.33	0.0
P_3	0.25	0.09	0.33	**0.37**	0.0
P_4	0.0	0.0	0.33	0.0	**0.47**

(b)

Fig. 9 Augmented graph and pattern similarity: **a** shows the augmented weighted graph for P_2 and P_3 in Fig. 8. Structural edges are *solid*, whereas attribute edges are shown *dashed*. **b** shows the pairwise similarity matrix between all five patterns in Fig. 8

We define similarity between two patterns $P = (V_P, E_P, L_P)$ and $Q = (V_Q, E_Q, L_Q)$, as

$$Sim(P, Q) = Jaccard(P, Q) \times Diffusion(P, Q) \tag{1}$$

Here, $Jaccard(P, Q) = \frac{|L_P \cap L_Q|}{|L_P \cup L_Q|}$ is the Jaccard coefficient between the label sets for P and Q. The more the labels in common, the higher the Jaccard similarity. $Diffusion(P, Q)$ is the diffusion kernel-based similarity between P and Q that considers both the structure and the label information, as described below.

Following a procedure similar to that in [35], given P and Q, we first create an augmented weighted graph $R = (V_R, E_R, W_R)$. Here, $V_R = V_P \cup V_Q \cup \{l | \exists v \in V_P, L_P(v) = l\} \cup \{l | \exists v \in V_Q, L_Q(v) = l\}$, that is, R contains both structural nodes (those in P and Q) and attribute nodes (labels for nodes in P and Q). $E_R = E_P \cup E_Q \cup \{(v, l) : v \in V_P, L_P(v) = l\} \cup \{(v, l) : v \in V_Q, L_Q(v) = l\}$. In other words, E_R contains both the structural edges (the original edges between vertices in both P and Q), as well as the attribute edges (between a node in P and Q, and its label). Finally, $W_R : E_R \rightarrow \mathbb{R}$ is a function that assigns a weight to each edge. The weights on structural edges are set to 1, that is, $W(u, v) = 1.0$ for all $(u, v) \in E_P \cup E_Q$. The weights on attribute edges are set as follows: $W(v, l) = \frac{1}{n_l}$, where n_l is the number of neighbors of node l in R. In the augmented graph, two structural vertices that have the same label l are both neighbors of the attribute node l. To avoid inflating the

similarity purely due to labels (which has already been accounted for by $Jaccard(P, Q)$), we assign the fractional weight on attribute edges. Figure 9a shows the augmented weighted graph for P_2 and P_3 from Fig. 8.

To compute $Diffusion(P, Q)$ for each pair of patterns, we use the diffusion kernel approach [22] over their augmented graph. A diffusion kernel mimics the physical process of diffusion where heat, gases, etc., originating from a point diffuse with time. On graphs, it is the local similarity that diffuses via continuous time random walks (i.e., with an infinite number of infinitesimally small steps). Given the augmented graph $R = (V_R, E_R, W_R)$, the matrix W_R is taken to be the weighted adjacency matrix of R. Further, define the diagonal degree matrix as $D(v_i, v_i) = \sum_{v_j} W_R(v_i, v_j)$, and $D(v_i, v_j) = 0$ when $i \neq j$. The *Laplacian matrix* of R is then defined as: $N = D - W_R$. Finally, the diffusion kernel matrix is defined as $K = e^{\beta L} = \sum_{k=0}^{\infty} \frac{\beta^k}{k!} L^k$, where β is a real-valued diffusion parameter, and $e^{\beta L}$ is the matrix exponential (with $L^0 = I$ and $0! = 1$). Since L is positive semi-definite, it has $|V_R| = n$ real and positive eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$. Let $\boldsymbol{u}_i$ be the eigenvector corresponding to eigenvalue λ_i. Then, the diffusion kernel can easily be computed as the spectral sum [22]: $K = \sum_{i=1}^{n} \boldsymbol{u}_i e^{\beta \lambda_i} \boldsymbol{u}_i^T$. The eigenvalues and eigenvectors of K can be computed in $O(n^3)$ time, where $n = |V_R|$.

The kernel matrix entry $K(v_i, v_j)$ gives the diffusion-based similarity between any two vertices in the augmented graph R for patterns P and Q. In particular, we are interested in those entries $K(u, v)$ where $u \in V_P$ and $v \in V_Q$. We define the diffusion similarity between P and Q as follows: If $L_P \cap L_Q = 0$, then we set $Diffusion(P, Q) = 0$, otherwise

$$Diffusion(P, Q) = \min_{l \in L_P \cap L_Q} \left\{ \max_{\substack{u \in V_P, v \in V_Q \\ (u,l),(v,l) \in E_R}} \{K(u, v)\} \right\}$$

In other words, the diffusion similarity between P and Q is defined as the least label similarity over all labels l, such that the label similarity is the maximum kernel similarity over pairs of nodes u, v that share a given label l. Figure 9b shows the pairwise similarities between all the patterns in Fig. 8, based on Eq. (1), that combines both the $Jaccard()$ and $Diffusion()$ values.

5.2 Clustering

We employ graph clustering to cluster the set of maximal patterns M. In particular, given the similarity matrix $S(i, j) = Sim(P_i, P_j)$ between any two patterns $\in M$, we can think of S as the weighted adjacency matrix of a *similarity graph*, where each maximal pattern is a node, and any two maximal patterns are linked with weight $S(i, j)$. Clustering of the patterns is then equivalent to clustering the nodes in the similarity graph. While many algorithms have been proposed for graph clustering [29], we use the Markov clustering (MCL) [31] approach as opposed to spectral methods [30], since MCL does not require the number of clusters as input.

Let D be the diagonal degree matrix corresponding to the weighted similarity matrix S. Let $N = D^{-1}S$ be the normalized adjacency matrix for the similarity graph. The matrix N is a row-stochastic or Markov matrix that specifies the probability of jumping from node P_i to any other node P_j. N is thus that transition matrix for a Markov random walk on the similarity graph. As such, the kth power of N, namely N^k, specifies the probability of transitioning from P_i to P_j in a walk of k steps. MCL [31] takes successive powers of N to *expand* the

influence of a node. However, it damps the extent of a nodes' influence, by an *inflation* step, whose goal is to enhance higher and diminish lower transition probabilities. Given transition matrix N, define the inflation operator Υ given as follows: $\Upsilon(N, r) = \left\{ \frac{N(i,j)^r}{\sum_{a=1}^{n} N(i,a)^r} \right\}_{i,j=1}^{n}$. In essence, Υ takes each element of N to the rth power and then re-normalizes the rows to make the matrix row-stochastic.

Given the initial N matrix, and an inflation parameter r, MCL is an iterative matrix algorithm consisting of two main steps: i) expansion: $N = N^2$, followed by ii) inflation: $N = \Upsilon(N, r)$. The method converges to a doubly idempotent matrix, and the strongly connected components in the corresponding induced graph yield the final node clusters [31]. The only parameter in MCL is the inflation value r that controls the granularity. Higher values lead to more, smaller clusters, whereas smaller values lead to fewer, larger clusters. MCL runs in $O(tn^3)$ time, where $|M| = n$, and t is the number of iterations until convergence.

5.3 Infrastructure pattern extraction

Given a set of clusters C_i, $1 \leq i \leq k$ obtained via the MCL approach, the final step in CMDB-Miner is to extract the so-called infrastructure patterns, that is, representative members from each cluster. Given a similarity threshold θ, from each cluster C_i we aim to extract as subset of the patterns $R_i \subseteq C_i$, such that for each $P_j \in C_i$, there exists a pattern $P \in R_i$ with $Sim(P_j, P) \geq \theta$. The task is to find a minimal set of representative patterns for each cluster. However, this problem is equivalent to smallest set cover, an NP-Complete problem, which nevertheless has a greedy $\Theta(\log n)$ approximation algorithm [12]. The greedy heuristic iteratively chooses the pattern that covers or represents the largest number of remaining elements in a cluster, until all the cluster members are covered.

6 Experimental evaluation

In this section, we evaluate CMDB-Miner on real-world CMDB graphs for two multi-national corporations, company A and B (names not revealed due to non-disclosure issues), from HP's Universal Configuration Management Database (UCMDB). We also conduct experiments to validate some of the design choices in the implementation of CMDB-Miner. All experiments were performed on a machine with 2.67GHz Intel i7 processor with 4GB of memory running Ubuntu Linux version 10.04.

6.1 Preprocessing

The raw CMDB graph of company A contains 443,192 vertices and 480,143 edges. Company B also contains a similar number of nodes and edges and is shown in Table 3. We discard

Table 3 CMDB Graphs A,B: before and after preprocessing

Property	A		B	
	Before	After	Before	After
$\|V\|$	443,192	11,363	455,012	57,525
$\|E\|$	480,143	20,978	523,415	149,229
Avg. Deg.	2.16	3.68	2.3	5.16

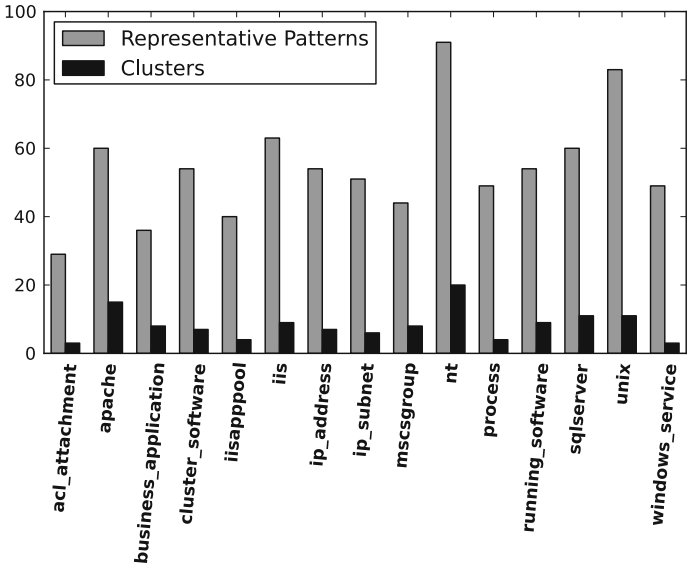


Fig. 10 Attribute pruning

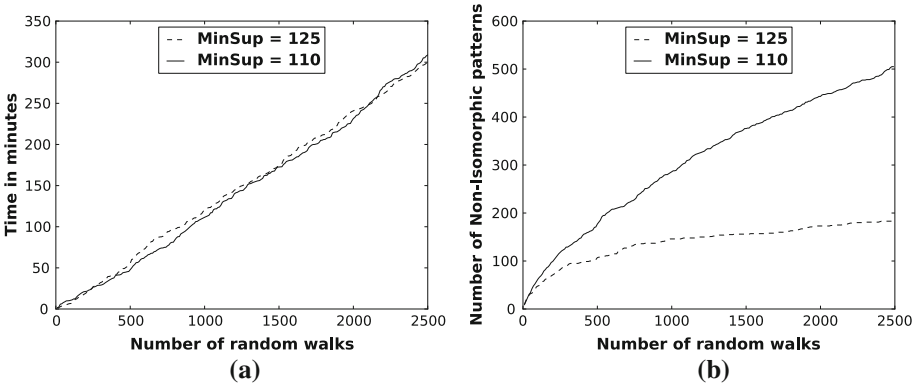


Fig. 11 Maximal and non-isomorphic patterns for company A: **a** sampling time and **b** number of distinct maximal patterns, versus number of random walks

uninformative attributes for each composite item, by discarding both high and low entropy attributes. This results in a significant reduction in the number of attributes, as shown in Fig. 10 for some of the common CI types in the CMDB graph of company A (similar results are obtained for B too). More than 75 % of the attributes are pruned in this stage. Further, collapsing leaf nodes reduces the total number of vertices to 11,363. Table 3 shows the graph order and size, as well as average degree, both before and after preprocessing. As pointed out earlier, these two preprocessing steps also aid in better interpretation of the final infrastructure patterns.

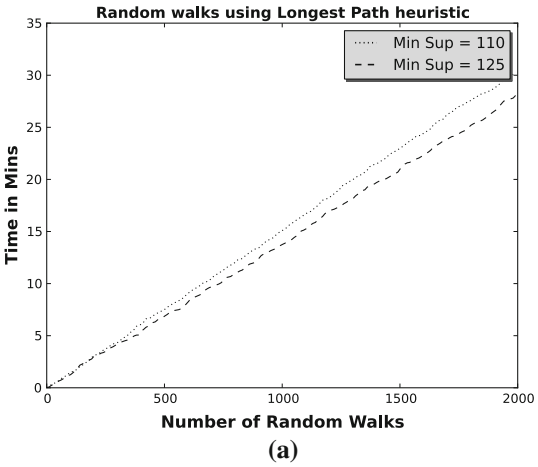


Fig. 12 Sampling time for company a using longest path Optimization. **a** Time

(a)

Table 4 Biggest pattern extracted

Database	# Vertices	# Edges
A	24	41
B	54	55

Table 5 Isomorphism checking filters (Time in Seconds)

Walks	Non-isomorphic	Filtering	VF2
100	89	0.17	2.05
500	361	5.26	82.07
700	464	3.14	47.57
1,000	671	10.96	148.66

6.2 Sampling maximal patterns

Figure 11a shows the time for sampling maximal patterns versus number of random walks, for two different (absolute) values of minimum support for company A using the full flow algorithm. We can see that as expected time is linear in the number of walks. Figure 11b shows the number of distinct or non-isomorphic maximal patterns versus the number of random walks. We can see that for $minsup = 125$, the fraction of distinct maximal patterns decreases with the number of walks, indicating convergence to the "true" set of maximal patterns. The convergence has not yet been reached for $minsup = 110$ within 2,500 walks. These curves suggest an automated method to stop sampling, namely when the ratio of the number of distinct patterns to the number of walks falls below some threshold. Figure 12a shows the time for random walks using the LP network. It can be seen that the LP formulation greatly reduces the running time for sampling maximal patterns. Similar results were also obtained for company B. We show the size of the maximal pattern extracted in Table 4.

Table 5 shows the time to detect the number of distinct maximal patterns. We compare the time taken by our filter-based approach versus the cost of running the VF2 algorithm on each pair of patterns in M. It is clear that the sequence of filters is very effective in reducing the running time by over an order of magnitude.

6.2.1 Baseline algorithm (BA)

To compare the effectiveness of our random sampling and network flow formulations, we compared them with a baseline approach we implemented. Most of the existing single graph mining algorithms explore the entire pattern space and require an enumeration of all the embeddings to compute the support of a candidate pattern. Instead of comparing with a specific algorithm, we formulated a baseline approach that retains a common attribute present in all the existing algorithms, namely embedding enumeration. The support of a pattern is computed using the maximum flow in the LP network constructed using all the embeddings, and the candidates are generated using random sampling as before.

Unfortunately, on the real data sets from Company A and B, the baseline algorithm could not be run (it crashed after some time) because of the exponential number of embeddings of the candidates, especially given the multiplicity and other characteristics of the CMDB graphs mentioned earlier. CMDB-Miner is essentially the only viable approach for these types of graphs. Nevertheless, to get a better understanding of the baseline methods and CMDB-Miner, we also compared them on synthetic graphs, as described below.

6.2.2 Synthetic graph data

Synthetic graph data sets were generated using the graph generator provided by Kuramochi et al. [23]. A simple graph can be generated by using the following parameters: number of transactions (N), number of Edge Labels (L), number of vertex labels (V), and other parameters that control the size and number of frequent patterns in the graph. Figure 13a shows the time versus number of random walks in the pattern space for four different minimum support values in a graph containing 3,000 edges, 1,817 nodes, with 200 node labels, and constructed using frequent patterns of size 10. All the edges in the graph have the same label. The figure shows that the time is linear in the number of walks for the different support values. However, note that that due to the random pattern extensions, there is not always a clear relationship between the time for a lower and higher minimum support. In general, it is the case that a lower support values take more time. However, we can see that extracting 2,000 patterns using support of 2 takes slightly less time than that for support 4 for these sparse synthetic graphs. This might be an artifact of the higher probability of finding frequent extensions with lower support values.

The plot in Fig. 13b shows the number of non-isomorphic maximal patterns for support value of 10, as a function of the number of random walks. Figure 13c compares the run times for a given number of random walks using the CF network and the LP optimization. The minimum support used for this experiment is 4 and the graph is generated using the parameters: 7,000 edges, 3,748 nodes, with 60 distinct vertex labels, 1,400 frequent graphs of average size 10. By using the LP optimization, the run time reduces significantly. This is due to the fact that flow in the LP network is a tighter upper bound and in general contains fewer nodes and edges. Figure 13d shows the time for enumerating 1,000 maximal patterns versus the number of nodes/vertices in the graph data set. Here, we also compare with the baseline method. When the data set size is small, the LP approach is marginally better compared to the baseline or the CF approach. However, as the data set size increases, the run time for LP becomes significantly less compared to other two algorithms. The figure also shows that the run time for BA increases steeply as the data set size increases where the increase for LP is much slower. This also shows the scalability of the LP algorithm. Finally, Fig. 13e shows that the run time in general decreases as the minimum support is increased. Here again, the LP network yields the best results.

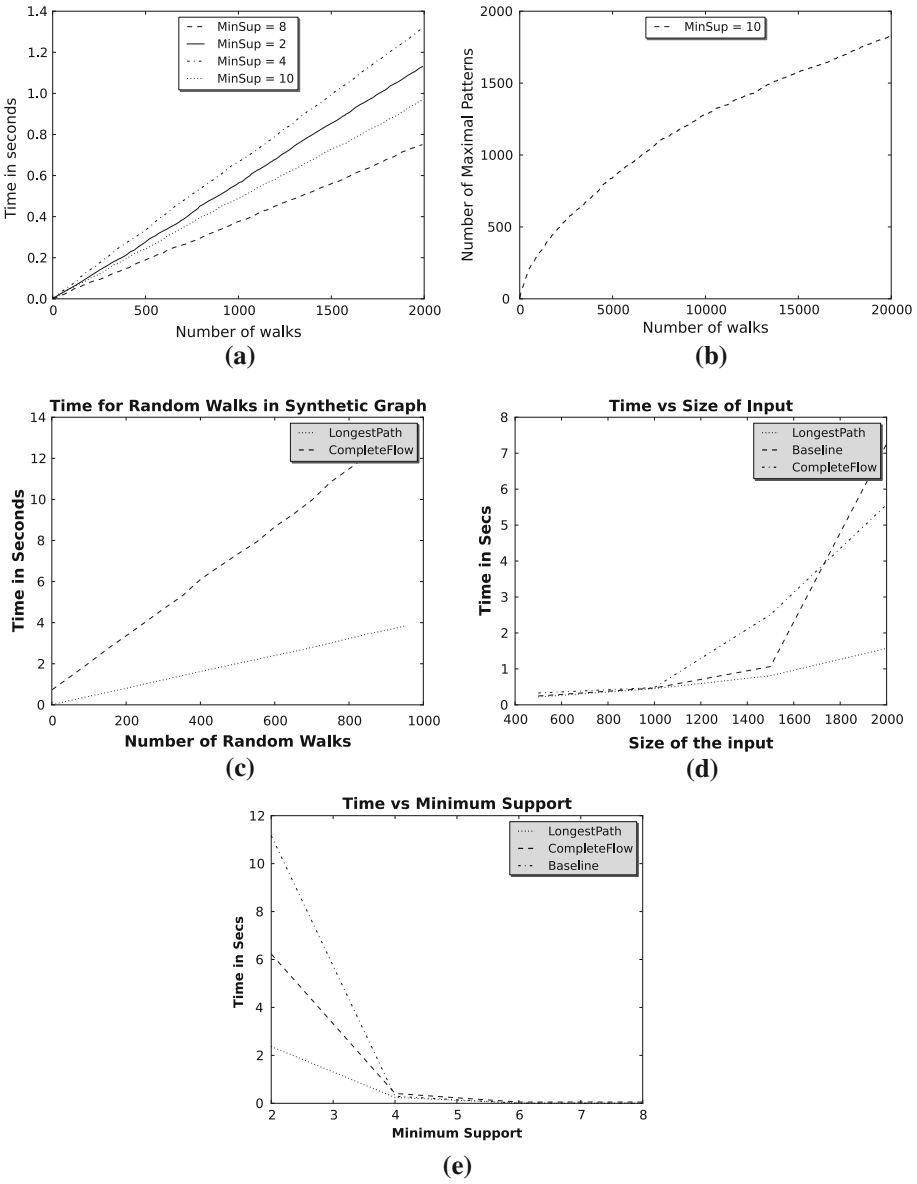


Fig. 13 **a** Shows times for random walks with different minimum support. **b** Shows the number of non-isomorphic maximal patterns. **c** Compares the run time using longest path and complete flow networks. **d** Shows the effect of database size (number of nodes) on the run time. **e** Shows the effect of minium support on run time. All the times mentioned are in secs

6.3 Pattern similarity

We now show the effectiveness of the our new similarity metric $Sim(P, Q)$ defined in (1), by comparing it with the *SimFlood* algorithm [27], which is a widely used technique for graph matching. Given two patterns, P and Q, SimFlood first computes a match score

Table 6 Similarity flooding variations: σ^i is the similarity between a pair of nodes in the ith iteration, and φ is the weighted sum of the similarities of all the predecessors of a given node pair

Identifier	Fixpoint formula
Basic	$\sigma^{i+1} = normalize(\sigma^i + \varphi(\sigma^i))$
A	$\sigma^{i+1} = normalize(\sigma^0 + \varphi(\sigma^i))$
B	$\sigma^{i+1} = normalize(\varphi(\sigma^i + \sigma^i))$
C	$\sigma^{i+1} = normalize(\sigma^i + \sigma^i + \varphi(\sigma^i + \sigma^i))$

between nodes in P and Q based on the string matching of the labels. The match score of a pair of labels depends on the prefixes and suffixes of the labels. This matching criterion is not meaningful in the case of maximal patterns extracted from CMDB graphs because all the labels are defined in the same context, and every pair of labels is either the same or different. Hence, we used a $0/1$ score for the initial match. In the second step, SimFlood constructs an induced propagation graph in which each node is a pair of matchable nodes obtained from P and Q. The number of nodes in the induced propagation graph can be as many as $|E_P| \times |E_Q|$ depending on the edge labels. In contrast, the order of the Augmented Weighted Graph constructed by our algorithm is equal to the sum of the number of nodes and labels present in both the patterns. Though the payoff by using Augmented Graph is marginal when the task is just to compare a single pair of patterns, there is a significant improvement in the overall runtime when similarity between every pair of patterns is required. In SimFlood, the similarity between nodes is computed by propagating the initial match scores to the neighbors in a manner similar to the pagerank algorithm [4]. SimFlood uses four different formulas (shown in Table 6) for computing the match score in any iteration using the score in previous iteration and the initial match score. Our algorithm on the other hand propagates the similarity using the idea of diffusion. To make a fair comparison between the algorithms, similarity score between P and Q is computed similar to $Diffusion(P, Q)$, except that the similarity between nodes v_i and v_j is the fixed point value of the node pair (k_i, k_j) in the induced propagation graph instead of the kernel matrix entry $K(v_i, v_j)$.

6.3.1 Comparing similarity metric

We compare our $Sim(P, Q)$ based kernel, with the $SimFlood(P, Q)$ kernel, by comparing the quality of the clusters the MCL algorithm generates using these kernels. The quality of a clustering of maximal patterns is measured using coverage and conductance [21], which are frequently used in graph clustering literature. The value of the minimum cut of a cluster is a measure of its quality. A low value means that the cluster contains large number of dissimilar pairs of nodes, which in turn implies that the cluster is of bad quality. Given a clustering $C = \{C_1, C_2, \ldots, C_k\}$ of the graphs into k clusters, the conductance $Con(C_i)$ of a cluster C_i is a generalization of the minimum cut that considers the size of the cut and also the similarity between a node and its neighbor. Given a set of clusters, the conductance of the clustering is defined as the minimum of the conductance of all clusters, and the intercluster conductance $Con(C)$ [2] can be defined as the compliment of the maximum conductance over all the clusters.

$$\begin{aligned} Con(C_x) &= \frac{\sum_{i \in C_x, j \notin C_x} a_{ij}}{\min\left(a(C_x), a(V \setminus C_x)\right)} \\ Con(C) &= 1 - \max_{i \in 1,\ldots,k} \phi(C_x) \end{aligned} \tag{2}$$

Table 7 $I = 2$: Time for computing the pair similarity between maximal patterns using diffusion kernel approach and variations of the similarity flooding

Method	Time (s)	Cov^1	Con^2	Clusters	#Repr
Basic	148.82	0.97	0.29	2	13
A	240.09	0.99	0.28	2	25
B	45.47	0.97	0.33	2	20
C	36.84	0.99	0.28	2	25
Diffusion	**9.68**	0.98	**0.33**	2	24

Cov^1 Coverage of the clustering
Con^2 External conductance of the clustering
Bold values indicate best results in time or quality

Table 8 $I = 4$: Time for computing the pair similarity between maximal patterns using diffusion kernel approach and variations of the similarity flooding

Method	Time (s)	Cov^1	Con^2	Clusters	#Repr
Basic	148.82	0.89	0.17	4	16
A	240.09	0.99	0.28	2	25
B	45.47	0.94	0.31	3	19
C	36.84	0.99	0.33	2	26
Diffusion	**9.68**	0.86	0.21	5	26

Cov^1 Coverage of the clustering
Con^2 External conductance of the clustering
Bold value indicates best result in time

where a_{ij} is the similarity between the vertices i and j in the graph, and $a(C_x)$ is the sum of similarities between all pairs of vertices with at least one end in cluster C_x.

The other quality measure we use is the coverage (Cov), defined as the ratio of sum of the similarities between vertices in the same cluster and sum of similarities between all pairs of vertices in the graph. A clustering with a high value of coverage is preferred over a clustering with low coverage.

Tables 7 and 8 compare our similarity metric with SimFlood in terms of time, the number of resulting clusters, cluster coverage, conductance, and the number of representative patterns extracted for different values of inflation parameter used in the MCL algorithm. In the tables, Basic, A, B, and C refer to the different update formulas of SimFlood, whereas Diffusion refers to our similarity method.

It is evident from the table that computing the similarity using our approach is significantly faster compared to the SimFlood method. We achieve this efficiency without sacrificing the quality of the resulting clusters. With inflation $I = 2$ (Table 7), the coverage of the clustering obtained using our similarity metric is very close to the maximum possible value 1. This means that the similarity between the patterns in the same cluster is very high compared to the similarity between patterns in different clusters. With inflation $I = 4$ (Table 8), our coverage is less than the coverage obtained using SimFlood. However, note that our method finds more clusters, and coverage is biased toward fewer clusters, which is the main reason for lesser coverage value. In terms of the conductance of the clustering, our similarity metric gives the best value when the value of inflation is $I = 2$, when

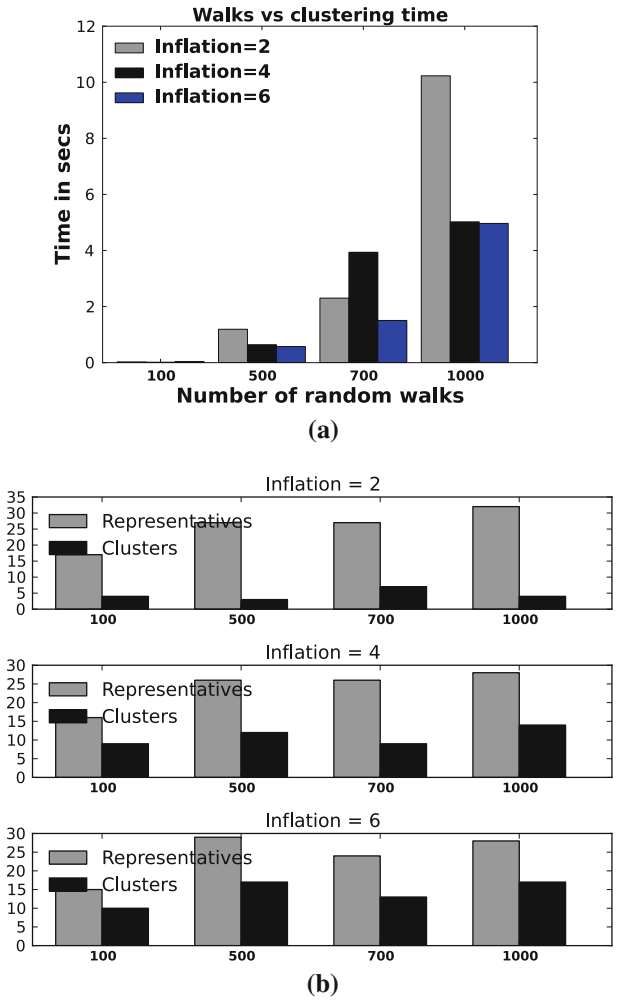


Fig. 14 **a** Shows time to extract clusters, and **b** shows the number of clusters and representative infrastructure patterns, for different values of inflation parameter

all methods find the same number of clusters. Ideally, one should compare the coverage and the conductance values when the number of resulting clusters is same using both the methods. We tried MCL with a high values for inflation so that the number of clusters returned using SimFlood is 5. Even with an inflation of 20, the number of clusters generated using method C remained at 2, whereas our algorithm generated 9 clusters. This shows that $Diffusion$ is better compared to SimFlood at discriminating the maximal patterns. The tables also show the number of representative patterns extracted with a similarity threshold value of $\theta = 0.8$. It can be seen that the number of representatives extracted using our similarity metric is close to the number of representatives extracted using SimFlood with maximum coverage, and when $I = 4$, it achieves that with a higher number of clusters.

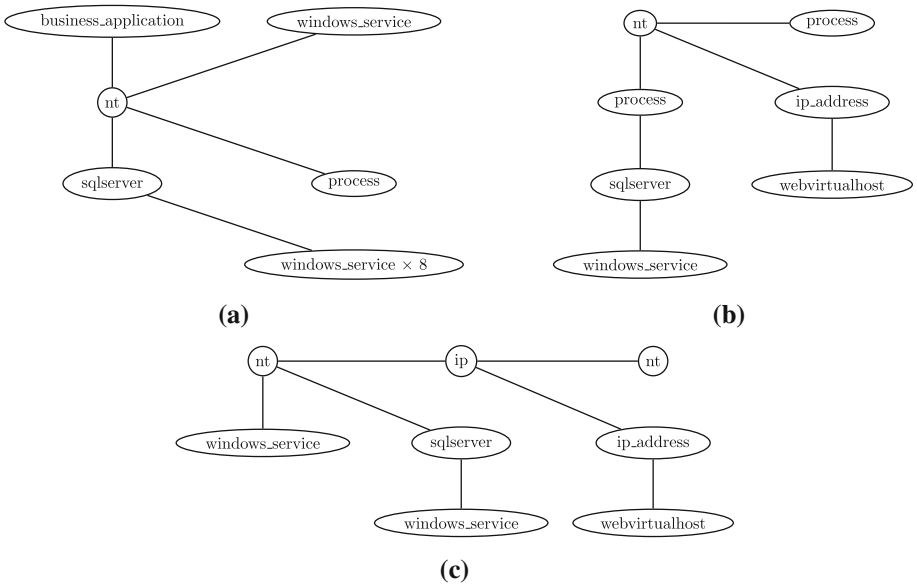


Fig. 15 Mined maximal patterns

6.4 Infrastructure pattern extraction

Having examined the performance of our sampling and similarity methods, we now turn attention to examples of mined infrastructure patterns from the real CMDB graphs. For company A, Fig. 14 shows the clustering time, and the number of clusters and representative patterns versus the given number of walks, for different values of the inflation parameter $r = 2, 4, 6$. We used $\theta = 0.9$ (threshold for a pattern to represent another pattern), and $\beta = 2$ (the diffusion kernel parameter). Clustering time is negligible compared to the time to sample the set of maximal patterns. The number of clusters increases with increase in the inflation parameter, as expected. Also, in most cases, the number of representative patterns remains the same. This is due to the characteristics of CMDB graphs, where each CI type is connected to only a limited number of other CI types. Thus, most of the maximal patterns contain either very similar or very different node labels. As the similarity measure is based on the attributes, these patterns tend to remain in the same cluster or different clusters, respectively. The small number of representative patterns shows the effectiveness of CMDB-Miner in summarizing large and sparse CMDB graphs into a small set of infrastructure patterns.

6.4.1 Example infrastructure patterns

Figure 15 shows three mined maximal patterns, which are a partial view of a general construct that is known in CMDB, and is defined as a standard Topological Query: *Node = nt, sqlserver = running_software*. Figure 16 shows another infrastructure pattern mined by CMDB-Miner. This pattern is a representative for several other patterns in a cluster. In order to choose a representative for each cluster, we currently choose a member of the cluster that maximizes the overall similarity to other members of the cluster. Alternatively, we can consider trimmed similarity (to say only the closest 80 % of the members), or we could aim for a description

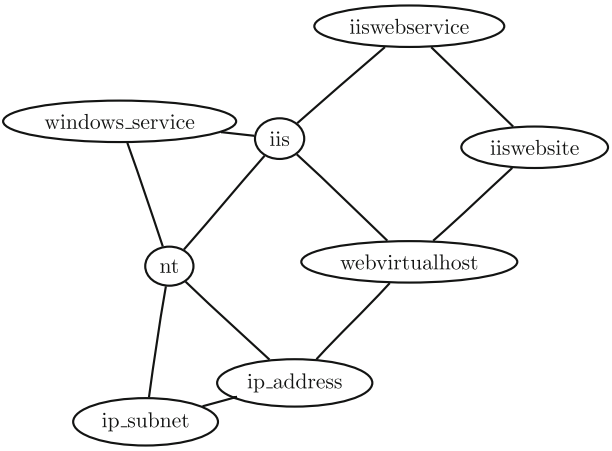


Fig. 16 Infrastructure pattern

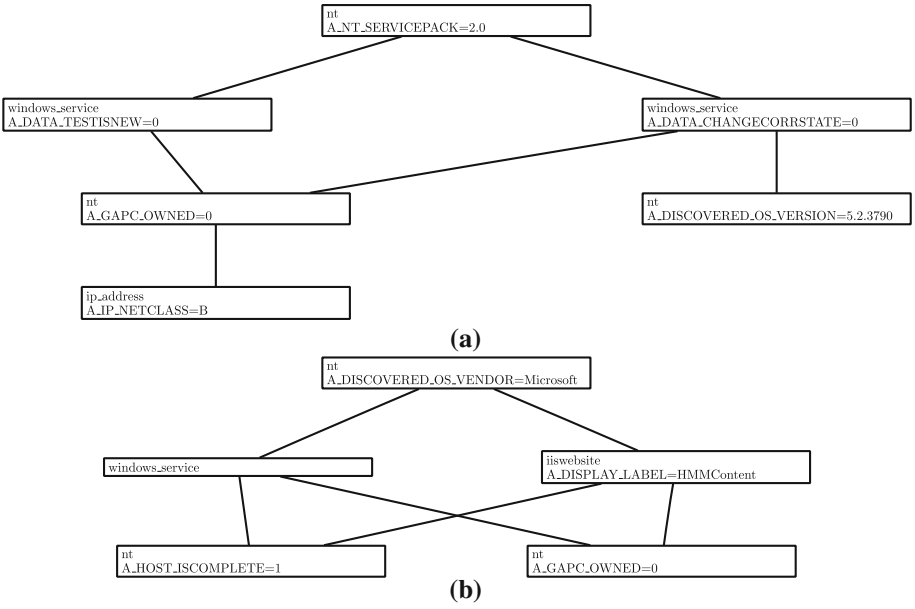


Fig. 17 Maximal pattern with node attributes

of a family of graphs that describes a large majority of the cluster. Exploring these options is part of future work.

6.4.2 Mining graphs with multiple attributes

Since CMDB graphs have a rich set of attributes for each composite item, we also mined patterns at a much finer level of granularity in terms of the CI nodes and their attributes. To mine such multi-attributed CI items, we created a new graph, where each node $v \in V$ with a label l and an attribute list $< A_1 : Val_1, \ldots, A_k : Val_k >$ is replaced with k

nodes with labels $l@A_1 - Val_1, \ldots, l@A_k - Val_k$, respectively. Also, each of the k labeled nodes from v are lined to each of the k' labeled nodes from another vertex v', if the edge (v, v') existed in the original coarse graph. Thus, if each node in the graph contains $O(d)$ attributes, then the size of the resulting graph increases by $O(d^2)$. This finer grained graph can then be mined using CMDB-Miner as before. Figure 17 shows some example patterns extracted from company A with richer node labels. For example, we can see that the CI item corresponding to nt has been refined into several of its specific attribute values, for example, $nt.A\_NT\_SERVICEPACK = 2.0$, $nt.A\_GAPC\_OWNED = 0$, and so on, for other CI items as well. Such fine-grained patterns are more precise than the coarse CI item-based patterns and thus may lead to a better characterization of the de facto infrastructure patterns and policies in an IT organization.

7 Conclusions

We have demonstrated that CMDB-Miner is an effective algorithm for mining real-world CMDB graphs. It makes use of the characteristics of such graphs (e.g., label multiplicities, sparsity) to speed up the mining process. It performs random walks without having to check for isomorphism, which is only performed on the final set of maximal patterns via a filter-based approach. Further, we proposed a new flow-based upper bound on the edge-disjoint support that allows for effective pattern pruning and which avoids the exponential blowup in the number of possible embeddings that plague many previous methods. To extract the infrastructure patterns, we proposed a new diffusion kernel-based similarity that takes into account both the structure and label information. We show that CMDB-Miner is able to extract meaningful infrastructure patterns. In terms of future work, we plan to parallelize the approach for better scalability. We also want to explore other random sampling techniques by which the probability of exploring a non-isomorphic pattern is high in each random walk. We would also like to mine approximate patterns, with possibly mismatched nodes and edges.

References

1. Al Hasan M, Zaki MJ (2009) Output space sampling for graph patterns. In: Proceedings of the 35th international conference on very large data bases, VLDB endowment, vol 2, no. 1, pp 730–741
2. Almeida H, Guedes D, Meira W Jr, Zaki MJ (2011) Is there a best quality metric for graph clusters? In: 15th European conference on principles and practice of knowledge discovery in databases
3. Besemann C, Denton A (2007) Mining edge-disjoint patterns in graph-relational data. In: Proceedings of the workshop on data mining for biomedical informatics at SDM-07, Citeseer, Minneapolis
4. Brin S, Page L (1998) The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the seventh international conference on, world wide web 7, WWW7, pp 107–117
5. Bringmann B, Nijssen S (2008) What is frequent in a single graph? In: 12th Pacific-Asia conference on knowledge discovery and data mining
6. Bunke H, Shearer K (1998) A graph distance metric based on the maximal common subgraph. Pattern Recognit Lett 19(3–4):255–259
7. Calders T, Ramon J, Van Dyck D (2011) All normalized anti-monotonic overlap graph measures are bounded. Data Min Knowl Discov. doi:10.1007/s10618-011-0217-y (online first)
8. Chaoji V, Al Hasan M, Salem S, Besson J, Zaki MJ (2008) ORIGAMI: a novel and effective approach for mining representative orthogonal graph patterns. Stat Anal Data Min 1(2):67–84
9. Chaoji V, Al Hasan M, Salem S, Zaki MJ (2008) An integrated, generic approach to pattern mining: data mining template library. Data Min Knowl Discov 17(3):457–495
10. Chen C, Lin CX, Yan X, Han J (2008) On effective presentation of graph patterns: a structural representative approach. In: Proceeding of the 17th ACM conference on information and knowledge management, ACM, pp 299–308

11. Chen C, Yan X, Zhu F, Han J (2007) Gapprox: mining frequent approximate patterns from a massive network. In: Proceedings of the 2007 seventh IEEE international conference on data mining, ICDM '07, pp 445–450
12. Chvtal V (1979) A greedy heuristic for the set-covering problem. Math Oper Res 4(3):233–235
13. Cordella LP, Foggia P, Sansone C, Vento M (2004) A (sub) graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell 26(10):1367–1372
14. Cvetkovic DM, Rowlinson P, Simic S, Biggs N (1997) Eigenspaces of graphs. Cambridge University Press, Cambridge
15. Dinitz Y (2006) Dinitzalgorithm: the original version and evens version. Theor Comput Sci :218–240
16. Fiedler M, Borgelt C (2007) Support computation for mining frequent subgraphs in a single graph. In: 5th international workshop on mining and learning with graphs
17. Hidovic D, Pelillo M (2004) Metrics for attributed graphs based on the maximal similarity common subgraph. Int J Pattern Recog Arti Intell 18(3):299–313
18. Huan J, Wang W, Prins J (2003) Efficient mining of frequent subgraphs in the presence of isomorphism. In: ICDM Proceedings, IEEE
19. Inokuchi A, Washio T, Motoda H (2003) Complete mining of frequent patterns from graphs: mining graph data. Mach Learn 50(3):321–354
20. Itai A, Perl Y, Shiloach Y (1982) The complexity of finding maximum disjoint paths with length constraints. Networks 12:277–286
21. Kannan R, Vempala S, Veta A (2000) On clusterings-good, bad and spectral. In: Proceedings of the 41st annual symposium on foundations of computer science, FOCS '00, p 367
22. Kondor R, Vert J-P (2004) Diffusion kernels. In: Scholkopf B, Tsuda K, Vert J-P (eds) Kernel methods in computational biology. The MIT Press, Cambridge
23. Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: 1st IEEE international conference on data mining
24. Kuramochi M, Karypis G (2005) Finding frequent patterns in a large sparse graph. Data Min Knowl Disc 11(3):243–271
25. Li S, Zhang S, Yang J (2010) Dessin: mining dense subgraph patterns in a single graph. Sci Stat Database Manag 178–195
26. Li J, Liu Y, Gao H (2011) Summarizing graph patterns. IEEE Trans Knowl Data Eng (99):1. doi:10.1109/TKDE.2010.48 (online early access)
27. Melnik S, Garcia-Molina H, Rahm E (2002) Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: Proceedings of the 18th international conference on data engineering, ICDE '02, p 117
28. Neuhaus M, Riesen K, Bunke H (2006) Fast suboptimal algorithms for the computation of graph edit distance. Struct Syntactic Stat Pattern Recogn 163–172
29. Schaeffer SE (2007) Graph clustering. Comput Sci Rev 1(1):27–64
30. Shi J, Malik J (2000) Normalized cuts and image segmentation. IEEE Trans Pattern Anal Mach Intell 22(8):888–905
31. Van Dongen S (2004) Graph clustering via a discrete uncoupling process. SIAM J Matrix Anal Appl 30(1):121–141
32. Vanetik N, Shimony SE, Gudes E (2006) Support measures for graph data. Data Min Knowl Discov 13(2):243–260
33. Yan X, Han J (2002) Gspan: graph-based substructure pattern mining. In: IEEE international conference on data mining
34. Zhang S, Yang J, Li S (2009) Ring: an integrated method for frequent representative subgraph mining. In: 2009 ninth IEEE international conference on data mining, IEEE, pp 1082–1087
35. Zhou Y, Cheng H, Yu JX (2009) Graph clustering based on structural/attribute similarities. Proc VLDB Endow 2(1):718–729

Author Biographies

Pranay Anchuri is a Ph.D. candidate in the Department of Computer Science at Rennselaer Polytechnic Institute, Troy, USA where he is advised by Mohammed J. Zaki. Earlier, he received his B.Tech in Computer Science with an Honors in Data Mining from IIIT Hyderabad, India. His research interests are in large scale graph mining, approximate pattern mining, and social network analysis.

Mohammed J. Zaki is a Professor of Computer Science at RPI. He received his Ph.D. degree in computer science from the University of Rochester in 1998. His research interests focus on developing novel data mining techniques, especially in bioinformatics. He has published over 200 papers and book chapters on data mining and bioinformatics. He is the founding co-chair for the BIOKDD series of workshops. He is currently Area Editor for Statistical Analysis and Data Mining, and an Associate Editor for Data Mining and Knowledge Discovery, ACM Transactions on Knowledge Discovery from Data, Knowledge and Information Systems, ACM Transactions on Intelligent Systems and Technology, Social Networks and Mining, and International Journal of Knowledge Discovery in Bioinformatics. He was/is the program co-chair for SDM'08, SIGKDD'09, PAKDD'10, BIBM'11, CIKM'12, and ICDM'12. He received the National Science Foundation CAREER Award in 2001 and the Department of Energy Early Career Principal Investigator Award in 2002. He received the HP Innovation Research Award, 2010–2012, and the Google Faculty Research Award in 2011. He is a senior member of the IEEE and was named an ACM Distinguished Scientist in 2010.

Omer Barkol is a research manager in HP Labs in Israel. The main research agenda of Omer and his team deals with analytics and collaboration in large organizations, with a focus on information management. As of 2008, Omer works in HP Labs, working both in the area of imaging and printing, and in the research area of IT information management. Omer has led the research project of print inspection which included a major tech transfer. He was also involved in research and development of image-based automation and in automation for configuration management. Today, Omer's main research focus is on complex-data mining (i.e., graphs), working with various teams in HP software. Prior to joining HP, Omer has led a software team in Charlotte's Web Networks, dealing with routing protocols. Omer has a Ph.D. and M.Sc in Computer Science from Israel Institute of Technology (Technion). His research areas were in theoretical computer science, coding theory, and cryptography. Omer received his B.Sc in Mathematics and Computer Science from the Technion.

Ruth Bergman is the director of HP Labs in Israel. The research agenda for the lab is analytics, collaboration, and automation in large organizations, with a focus on IT management and serviceability of printing systems. Ruth joined HP in October, 2001. She was the research manager and principal research scientist for the IT Informatics big bet since 2008. She has created business opportunities via a novel image-based technology for software testing applications, IT automation technology for configuration management, and a Knowledge Management system to enhance collaboration in IT management. She led projects to support HP's printing and imaging business including repairing defective images, including dust and scratch removal, which is included in all of HPs scanner and All-In-One products, and perceptual segmentation which extracts perceptual tags from images such as sky, skin, and foliage and is a key ingredient for enhancing images in the HP Indigo Photo Enhancement Server (HIPIE). Prior to joining HP, Ruth was a researcher at the NASA Jet Propulsion Laboratory in Pasadena, California, and at the Lincoln Laboratory at the Massachusetts Institute of Technology (MIT). Ruth has a Ph.D. in Electrical Engineering and Computer Science from MIT and a Master's and Bachelor's degrees in Computer Science from the University of California, San Diego. She has authored six book chapters, 10 conference papers, and holds 7 patents and 17 patent applications.

Yifat Felder is a software engineer in HP software Israel. Her team works on algorithms for extracting topological data from graph representation of the IT world. As of 2009, Yifat works in HP. Yifat has worked on several aspects, including incorporation of external topological data source into the existing topology, and developing a performance-related state machine enhancement for quicker data analysis. Her latest project is developing a search engine based on natural language. Yifat has a M.Sc. and B.Sc. degrees in Computer Science from Tel-Aviv University. Her main focus in research was in the bioinformatics field.

Shahar Golan is a researcher in HP labs Israel since May 2010. He is part of the Information Analytics lab, which deals with analytics and collaboration in large organizations, with a focus on information management. Shahar received his Ph.D. (summa cum laude) at Ben Gurion University in 2010. His research was done under the supervision of Prof. Daniel Berend on the topic "Algebraic Methods for Solving Constraint Satisfaction Problems." In the past, Shahar worked in HyperRoll (later purchased by Oracle) as a developer and later as the research team leader. HyperRoll develops an aggregation engine for relational database. Shahar's research interests are constraint satisfaction problems, algorithms on graphs and data mining.

Arik Sityon is a team manager in HP Software in Israel. The main agenda of Arik and his team deals with IT information management, with a focus on topological algorithms to analyze the IT information. As of 2006, Arik works in HP Software, working in the area of IT information management. Arik has led the project of supporting multi-tenancy in HP UCMDB product. He was also involved in research and development of a search engine for the IT information which is still his main work focus. Arik has an M.Sc. in Computer Science from Ben Gurion University. His research area was computational geometry. Arik received his B.Sc. in Computer Science from Ben Gurion University.