The Journal of Systems and Software 140 (2018) 17–31

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Efficient graph pattern matching framework for network-based in-vehicle fault detection

Sun Geol Baek, Dong Hyun Kang, Sungkil Lee, Young Ik Eom*

College of Software, Sungkyunkwan University, Suwon, Korea

ARTICLE INFO

Article history:
Received 14 August 2017
Revised 20 February 2018
Accepted 23 February 2018
Available online 24 February 2018

Keywords:
Vehicle fault detection
Graph pattern matching
Graph similarity
Top-*k* query answering

ABSTRACT

Abnormal messages propagated from faulty operations in an in-vehicle network may severely harm a vehicular system, but they cannot be easily detected when their information is not known in advance. To support an efficient detection of faulty message patterns propagated in the in-vehicle network, this paper presents a novel graph pattern matching framework built upon a message log-driven graph modeling. Our framework models the unknown operation of the in-vehicle network as a *query graph* and the reference database of normal operations as *data graphs*. Given a query graph and data graphs, we determine whether the query graph represents normal or fault operation by using the distance measure on the data graphs. The analysis of the faulty message propagation requires to consider the sequence of events in the distance measure, and thus, using the conventional graph distance measures can generate false negatives due to the lack of consideration of the sequence relationships among the events. We therefore propose a novel distance metric based on the maximum common subgraph (MCS) between two graphs and the sequence numbers of messages, which works robustly even for the abnormal faulty patterns and can avoid false negatives in large databases. Since the problem of MCS computation is NP-hard, we also propose two efficient filtering techniques, one based on the lower bound of the MCS distance for a polynomial-time approximation and the other based on edge pruning. Experiments performed on real and synthetic datasets to assess our framework show that ours significantly outperforms the previously existing methods in terms of both performance and accuracy of query responses.

1. Introduction

Modern vehicles provide diverse operations for safety, performance, and driving convenience. The in-vehicle operations are typically controlled by electronic control units (ECUs), which are connected by an in-vehicle network, such as a controller area network (CAN) (ISO Standard 11898, 1993), so they can interact with each other for their normal operations. Nowadays, an increasingly large number of ECUs (recently, reaching up to 80) have brought higher complexity in their control and analysis than ever before. Moreover, as autonomous driving tests on public roads have become a reality, operation analysis on the ECUs that send and receive messages over the CAN has become more crucial for automotive manufacturers. In practice, during the autonomous driving, ECUs often fail due to unexpected network situations such as external disturbances (Hansson et al., 2002; Suwatthikul et al., 2011). Such faulty operations may severely harm the entire vehicular system since the faulty operations are rapidly propagated to other ECUs in the network. Thereby, the efficient detection of ECU faults in the in-vehicle network has become increasingly important.

For the detection of ECU faults on the network level, most studies are based on message analysis of CAN that is widespread as an in-vehicle network. Fault diagnostics on the CAN bus can be generally divided into two main categories: online diagnosis and offline diagnosis. The majority of online diagnosis studies are directed onto the network scheduling analysis due to the considerable demand for accurate predictions of failure probabilities in the in-vehicle network. Tindell et al. (1995) proposed a method to analyze the worst-case delivery time of CAN messages caused by communication delays when the CAN bus is busy. Their seminal work greatly influenced many subsequent studies (Davis et al., 2007; Hansson et al., 2002; Keskin, 2013; Mubeen et al., 2011; Navet et al., 2000) for efficient scheduling techniques and failure probability models of the in-vehicle ECU network. The offline diagnosis, on the other hand, mainly focuses on analysis of in-vehicle network message logs using message rates (Suwatthikul, 2010) and the traits of messages (Taylor et al., 2014). In the offline diagnosis, the data read from the CAN are monitored based on pre-established reference values, and then are classified into either

* Corresponding author.
E-mail addresses: sg.baek@skku.edu (S.G. Baek), kkangsu@skku.edu (D.H. Kang), sungkil@skku.edu (S. Lee), yieom@skku.edu (Y.I. Eom).

https://doi.org/10.1016/j.jss.2018.02.050
0164-1212/© 2018 Elsevier Inc. All rights reserved.

good or bad conditions. For example, if the values indicating the vehicle states such as message information and frequency of occurrence have moved out of the reference values, such states are classified as bad conditions.

However, online and offline diagnosis methods are commonly conducted using planned operation tests that include a set of pre-defined diagnosis scenarios. Thus, their approaches can detect solely the presence of faults in the planned tests. For example, despite the presence of underlying network problems or malfunctions of ECUs during vehicle operations, no fault operations are detected if the overall network messages meet the criteria of vehicle diagnostics. These conventional vehicle diagnostics using predetermined criteria for the planned tests inevitably result in inaccurate and time-consuming tasks in such a large number of network message logs occurring during autonomous driving. As a result, they still suffer from detecting fault operations in real driving conditions, since unexpected vehicle operating conditions that do not exist in the predefined diagnosis scenarios can occur at any time. This inspired us to explore a new approach for detecting fault operations in the network message logs.

In this paper, we present a novel graph-based vehicle diagnosis framework. In our work, to describe message patterns and their propagation in the in-vehicle network, the vehicle operations in the CAN message logs are modeled as graphs, called *event sequence graphs*, where vertices and edges represent ECUs and message flows among the ECUs, respectively. The yet unknown patterns, such as an arbitrary set of messages in the message logs, are modeled as a *query graph* and normal operations of the vehicle are modeled as *data graphs*. Given a query graph and data graphs, and there are failures in the query graph, the query graph will not be exactly matched with any data graphs. Our framework determines whether the query graph represents the normal or fault operation by finding the top-k graphs in the data graphs. This allows us to detect the occurrence of fault operations in unknown message patterns even against the test of unexpected arbitrary conditions. In addition, by analyzing the top-k graphs, we can infer the vehicle operations that caused the fault.

In order to answer the top-k graph query, our framework exploits a graph distance metric between the query graph and the data graphs. However, when we use the conventional graph distance metrics for the *event sequence graph*, the graph distance is likely to be higher than the actual difference. Therefore, to better fulfill our aim, we propose a novel *event sequence graph distance* considering the propagation sequence of CAN messages based on the maximum common subgraph (MCS). In addition, we further propose efficient polynomial-time graph filtering techniques for speed-up, one based on the *neighborhood label-based* lower bound of MCS distance and the other based on edge pruning to reduce the expensive computation cost. We also show experimental results demonstrating that our novel graph distance method can efficiently avoid false negatives in the query processing of data graphs.

Precisely, our contributions can be summarized as follows: 1) a novel graph construction technique that models network message logs as a graph to represent vehicle operations; 2) a novel graph-distance metric that considers the event sequences; 3) two efficient polynomial-time graph filtering techniques; 4) a novel graph search algorithm for efficient graph pattern matching; and 5) experimental analysis of our algorithm against the existing approaches, in terms of performance and accuracy of the query response on real and synthetic datasets.

The rest of this paper is organized as follows. In Section 2, we define our problem, and introduce fault analysis of vehicle operations and a set of preliminary notations. Section 3 discusses the previous studies related to graph pattern matching. Section 4 presents an overview of our solution for in-vehicle fault detection. Section 5 describes our novel graph construction technique to represent vehicle operations in the network message logs, and Section 6 describes the matching problem in the conventional graph distance metrics and presents our novel graph-distance metric. In Section 7, we introduce two efficient polynomial-time graph filtering techniques. Section 8 explains how the queries are processed to detect faulty operations using the optimal mapping method. Section 9 reports the results of our experimental evaluations, and Section 10 concludes the paper.

2. Preliminaries

2.1. Fault analysis of vehicle operations in CAN

CAN is a communication protocol widely used in in-vehicle networks. Its message fields include ID, Data length, Data, CRC, and Ack. The CAN message log is the message traffic among ECUs using the CAN protocol, and consists of timestamps, message ID and data. The CAN protocol has an error detection mechanism that automatically retransmits the messages if an error is detected during communication. CAN also supports priority-based message arbitration using message identifiers to avoid transmission collision on the CAN bus. When multiple nodes in the CAN bus attempt to transmit messages concurrently, the message arbitration ensures that a message with the highest priority is transmitted without any preemption.

In our study, we focus on detecting fault operations in the CAN message logs recorded automatically or manually during autonomous driving tests (i.e., the arbitrary operation tests). In this paper, the failure of the vehicle operations is defined as the occurrence of an unintended function under given conditions. Consider the following cases. In the first case, intermittent external interference or hardware failure during autonomous driving can affect the CAN network bus. These network interferences cause a transmission error and then the messages are retransmitted by the transmitting ECU that detected the transmission error. In this case, the receiving ECU may have an inconsistent message duplicate (Pinho and Vasques, 2003). This effect can allow the receiving ECU to deliver duplicate messages to its application, which can propagate abnormal messages to several other ECUs. If the transmitting ECU fails before retransmitting the message due to internal faults, this results in a message loss. Similarly, in the second case, message retransmission due to the external interferences can be a major threat to the reliability of the vehicle operations in the priority-based CAN bus protocol. When the retransmission of high priority messages continues, none of the other ECUs with a low priority message could use the bus by the arbitration mechanism. This causes the node with a low priority message to miss its deadline. As a result, the message waiting time of some ECUs increases, which leads to the failure of the vehicle operations that require safety and reliability.

2.2. Problem definition

In this section, we first introduce the basic definitions that will be used in the subsequent sections, and formally describe our problem. Table 1 lists notations used in this study.

In this study, we focus on a directed graph g with vertices and edges, and denote it as (V, E, LV, LE), where V is the set of vertices, and E ($\subseteq V \times V$) is a set of edges. LV and LE are functions that assign labels for each vertex $u \in V(g)$ and edge $(u, v) \in E(g)$, respectively. Also, we call the graphs in the database D as *data graphs* and the results of the query as *answer graphs*.

Definition 1 (Graph isomorphism)**.** Given two graphs g_1 and g_2, g_1 and g_2 are isomorphic if there is a bijective function f: $V(g_1) \rightarrow V(g_2)$ such that $\forall u \in V(g_1)$, $f(u) \in V(g_2) \wedge LV_1(u) = LV_2(f(u))$, and $\forall (u, v) \in E(g_1)$, $(f(u), f(v)) \in E(g_2) \wedge LE_1((u, v)) = LE_2((f(u), f(v)))$.

Table 1
Primary notations used in this paper.

Notation	Definition and description
$V(g)$	vertices of graph g
$E(g)$	edges of graph g
$MCS(q, g)$	the maximum common subgraph (MCS) between two graphs q and g
$MCS_{dist}(q, g)$	a graph distance based on the MCS between two graphs q and g
$GED(q, g)$	graph edit distance between two graphs q and g
(q', g')	MCS pair between two event sequence graphs q and g
$w(q', g')$	minimum weight mapping function of (q', g')
$Edist(q, g)$	graph distance between two event sequence graphs q and g

Definition 2 (Maximum common subgraph)**.** Given two graphs q and g, the maximum common subgraph (MCS), denoted as $MCS(q, g)$, is the connected subgraph h with the maximum number of edges common to the two graphs q and g, where h is isomorphic to the subgraph of each of the two given graphs.

Definition 3 (MCS distance)**.** Given two graphs q and g, the MCS distance, denoted as $MCS_{dist}(q, g)$, is defined as $MCS_{dist}(q, g) = |E(q)| + |E(g)| - 2 \times |E(MCS(q, g))|$.

The graph edit distance measures the similarity between two graphs by counting the minimum number of edit operations to convert one graph into the other graph. The edit operations on a graph consist of the following six operations: an insertion of a vertex, a deletion of a vertex, a change of the label of a vertex, an insertion of an edge, a deletion of an edge, and a change of the label of an edge.

Definition 4 (Graph edit distance)**.** Given two graphs q and g, the graph edit distance (GED), denoted as $GED(q, g)$, is the minimum number of edit operations needed to transform q to g.

As mentioned in Section 1, due to the restriction of the conventional graph distance metrics we use the novel event sequence graph distance based on the MCS distance. Therefore, we define our problem as follows: given a query graph q and a graph database $D = \{g_1, \ldots, g_n\}$, we find the top-k answer graphs from D since they have the smallest event sequence graph distances from q.

3. RELATED WORK

A number of applications utilize graphs to model complex data on social networks (Terveen and McDonald, 2005), biology (Raymond et al., 2002a), and web applications (Zeng et al., 2013). In general, these applications support queries on graph databases to obtain answers. There are mainly two ways to query the graph database: 1) query language method, and 2) graph exploration method. In order to retrieve data from the graph database, the query language method uses structured query language such as SQL and SPARQL in relational database or RDF database. However, the query language method merely returns a set of nodes, as answers, that match with the input query. Therefore, this method has limitations in finding structured information in the graph database.

Meanwhile, many graph-based applications exploit the graph exploration method that outputs all the answers that match with an input graph (called query graph). In the graph exploration method, graph pattern matching is applied to obtain the answers from the graph database. In our work, for in-vehicle fault detection, we exploit the graph exploration method based on the pattern matching technique to detect faulty patterns in large graph database. In other domains, graph-based detection methods have been proposed (Collins and Reiter, 2007; Djidjev et al., 2011) for unknown patterns, such as worm intrusion detection in network traffic data. However, they assume the patterns to be unusually large, which is not directly applicable to our problem (typically, having small abnormal patterns against the large database).

Previous algorithms for the graph pattern matching can be broadly classified into the exact graph matching and inexact graph matching algorithms. One of the most common problems in the exact graph matching is the identification of isomorphism in the entire graph or subgraph. The problem of identifying isomorphism for a subgraph has been proven to be *NP*-complete. In contrast, for the problem of identifying isomorphism for the entire graph, it is not yet known whether the problem belongs to the *P* or *NP*-complete problems (Fortin, 1996).

In many cases of real datasets, such as protein analysis (Koch et al., 1996), chemical compound search (Raymond and Willett, 2002b), and machine learning (Conte et al., 2007), noise or deviations can be added to the query data. It is often impossible to find data graphs using the exact graph matching method. Thus, various graph applications exploit the inexact graph matching, relying on graph similarity. The graph similarity typically measures the distance between two graphs. MCS distance and graph edit distance (GED) are common for this purpose.

Unfortunately, directly applying the existing graph pattern matching scheme can lead to failure of the correct mapping when considering event sequences in CAN message logs. To solve this problem, our novel graph distance metric is based on MCS, but additionally incorporates the result of mapping calculation that considers the sequence number of the message in each edge. A similar approach by Zhu et al. (2014) exists for the matching problem among heterogeneous event data, where they also considered the event sequence to model the heterogeneous event logs. Their work differs from ours in that edge label represents the normalized frequencies of the events in a set of event logs, and it thus is impossible to find the right mapping when the number of event logs is insufficient.

Because the graph similarity calculation requires an extremely large computational cost, nearly all the existing algorithms employ graph decomposition, pruning, or filtering techniques to increase the speed of the computation, while trading accuracy for speed to some extents. Wang et al. (2012) proposed a GED-based algorithm that decomposes a query graph and a data graph into k-adjacent tree patterns, respectively, called *k-AT*s and calculates the GED using the number of common *k-AT* patterns between the two graphs. The *k-AT* is defined as a tree composed of vertices adjacent to a vertex v, where the vertices reside within k hops from the vertex v. To achieve early termination, Zhang et al. (2010) and Ding et al. (2014) proposed the SAPPER and NH-TA method using a *k-AT* based indexing structure, respectively. The hybrid structure used in SAPPER is the same structure as *k-AT* when $k = 2$. The NH-index used by NH-TA exploits the 1-*AT* structure of a vertex v and the number of edges existing among the neighbors of the vertex v. In addition, variations similar to *k-AT* such as *star* (Zeng et al., 2009), *branch* (Zheng et al., 2015), and *path-based q*-gram (Zhao et al., 2012) were presented.

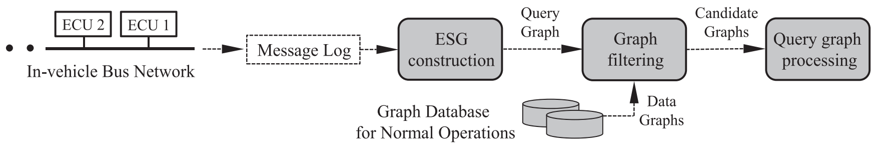


Fig. 1. Overview of our fault detection framework.

Several studies (Wang et al., 2012; Zhu et al., 2014; Zeng et al., 2009; Hu et al., 2013; Zheng et al., 2015; Zhao et al., 2012) use bounding methods to avoid the expensive graph similarity calculation by filtering out graphs that are unqualified in the graph database and thus improve the search performance in the large graph database. Zeng et al. (2009) filter out data graphs using the lower bound method; given a query graph q and the distance threshold t, for each data graph g in the graph database, they filter out the data graph g if the lower bound between the two graphs q and g is greater than t. Zhu et al. (2012) exploit multiple lower bounds to reduce the number of actual MCS distance computations. The first lower bound method, called $dist_1$, counts the occurrences of each distinct edge ($LV(u)$, $LV(v)$) in a graph, i.e., the number of distinct edges directed from a vertex u to another vertex v. This computation is performed for both the graph q and g, and they compare the counts of distinct edges in the two graphs. The time complexity of $dist_1$ computation is $O(|E(q)|+|E(g)|)$, but the lower bound is not tight. So, they also used another method proposed by Raymond and Willett (2002b), called $dist_2$, where it adopts a tighter lower bound. The $dist_2$ attempts to find a vertex mapping M that maximizes $\Sigma|\theta(adj(u)) \cap \theta(adj(v))|$ between two graphs q and g, where $u \in V(q)$, $v \in V(g)$, $LV_q(u)=LV_g(v)$, and $\theta(adj(u))$ denotes a multi-set having all labels of vertices adjacent to vertex u. The time complexity of $dist_2$ computation is $O((\max\{|V(q)|, |V(g)|\})^3)$. Hu et al. (2013) propose a filtering-and-verification architecture to prune non-promising graphs and return candidate graphs, where they use the lower bound on the number of relaxed edges based on MCS. The relaxed edges are edges which belong to a query graph and do not belong to MCS.

The lower bound-based filtering methods, however, suffer from high time complexity or difficulty in calculating the tightest lower bound. Our algorithm also filters out unnecessary data graphs in the graph database. We therefore use a neighborhood label-based lower bound of MCS distance while processing the query. The neighborhood label-based lower bound is not as tighter as the previous work (Raymond and Willett, 2002b), yet performs more efficiently for filtering, which is confirmed in our experiments.

4. Overview of the fault detection framework

In this section, we introduce fundamental concepts underlying our framework to detect in-vehicle fault operations. The fault detection framework consists of three components: *query graph construction, graph filtering*, and *query graph processing* (see Fig. 1). In our fault detection framework, there are several challenges in detecting fault operations. We briefly describe how to handle the challenges in each component as follows.

Event sequence graph construction. When a vehicle operation is triggered to work, several ECUs associated with the operation communicate through the network messages. The network messages can be collected into the message log from the bus network using network analysis software. Our graph construction method translates the network messages recorded in the log to *event sequence graphs* (ESGs), which will be described in Section 5. In Fig. 1, the *query graph* and *data graph* are also ESGs and describe the pattern of messages generated from an arbitrary operation and a normal operation in a vehicle, respectively. We assume that the graph database holds all the data graphs for normal operations of the vehicle.

Graph filtering. We measure the graph distances in terms of MCS. Since MCS computation is NP-hard and finding the query graph in the graph database is too expensive, we employ two efficient graph filtering techniques with polynomial-time complexity. We first select candidate graphs by filtering out data graphs in the graph database using the lower bound of the MCS distance. Then, we prune out the redundant edges of the candidate graphs to reduce the search space on the candidate graphs.

Query graph processing. Given the filtered candidate graphs, we compare the query graph with the candidate graphs using a much stricter algorithm that calculates the distance between the query graph and the candidate graphs (Section 8). In the calculation for the distance of the two graphs, we use a novel graph distance metric, because the use of conventional distance metrics such as MCS distance and GED can find many false negatives; more details of this problem are discussed in Section 6. If the query graph represents normal operation, it will be matched exactly at least one data graphs in the graph database and the distance value between the two graphs will become zero. In this way, we can identify the vehicle operation of the query graph. In contrast, the query graph of a fault operation will not be exactly matched with any candidate graph. Thus, from the graph database, we retrieve the top-k answer graphs having the smallest event sequence graph distances with the query graph. This allows us to find a data graph that is most similar to the query graph with fault operations.

5. Event sequence graph construction

In order to detect a fault operation in the CAN message log, we first model the vehicle operation in the CAN message log as a graph. Each message M in the CAN message log is defined by a tuple (s_a, d_a), where s_a and d_a represent the source and destination ECUs, respectively. In the message M, multiple d_as (destination ECUs) can be defined. An intuitive way to construct the graph is to connect the vertex s_a with the vertex d_a using a directed edge. For example, in Fig. 2, the message log has M_1, M_2, M_3, and M_4 which were generated from four ECUs (A, B, C, and D). For M_1, A is connected with B and C and the connections can be represented by a set of edges {(A, B), (A, C)}.

While this approach is straightforward, it may cause ambiguities due to the lack of message order representations. Consider the messages M_1 and M_2 in Fig 2. Given a message order of M_1 and M_2, we can finally obtain three edges (A, B), (A, C), and (A, B). However, a different message order of M_2 and M_1 leads to the same set of edges, which causes the ambiguity. Thus, we need an alternative policy to construct the graph from the message logs.

To address this ambiguity problem, we define graph creation rules that incorporate the order of message generation. To simplify the representation of the connection among the vertices, we define the latest vertex datasets (LVDs). A LVD consists of two sets: the set of labels (LVL) and the set of IDs (LVI) of vertices. In order to avoid the ambiguity, each edge is assigned a sequence number. As soon as an edge has been created, we update the sequence number in the graph. We call this graph an *event sequence graph*.

Message log

M	s_a	d_a
M_1	A	B,C
M_2	A	B
M_3	B	A,C,D
M_4	B	D

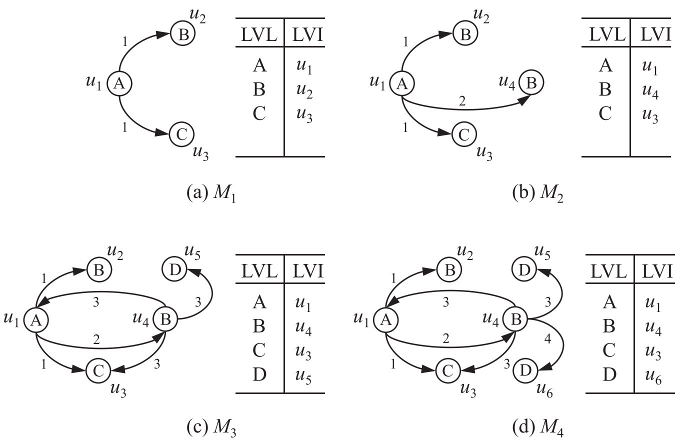

(a) M_1
LVL | LVI: A u_1, B u_2, C u_3
(b) M_2
LVL | LVI: A u_1, B u_4, C u_3
(c) M_3
LVL | LVI: A u_1, B u_4, C u_3, D u_5
(d) M_4
LVL | LVI: A u_1, B u_4, C u_3, D u_6

Fig. 2. Example of ESG creation process.

Algorithm 1 CreateESG(ML).

```
Input: a message log ML
Output: an ESG g
{
 1:  initialize array P with MAX integer size;
 2:  LVL ← ∅; LVI ← ∅; sequence number t ← 1;
 3:  for each (s_a, d_a) ∈ ML do {
 4:    if (s_a ∈ LVL) then
 5:      select the vertex s_a in V(g);
 6:    else
 7:      add a new vertex s_a with ID u_i in V(g);
 8:      i ← i+1;
 9:    for each d ∈ d_a do {
10:      if (d ∈ LVL and (s_a, d) ∉ E(g)) then
11:        select the vertex d in V(g);
12:      else
13:        add a new vertex d with ID u_i in V(g);
14:          i ← i+1;
15:      Add an edge from s_a to d with t;
16:    }
17:      t ← t+1;
18:    if ∃u ∈ V(g) such that P[u] = 0 then
19:      if (L(u) ∈ LVL) then
20:        update the ID of u in LVI to new u_i;
21:      else
22:        insert the u and the ID of u to the LVL and LVI, respectively;
23:      P[u] = 1;
24: }
25:  return g;
}
```

Definition 5 (Event sequence graph)**.** An event sequence graph (ESG) g is a directed graph, denoted by (V, E, L, s), where V represents the set of ECUs, E represents a set of message flows among the ECUs, and L is a function that assigns a label (ECU's ID) for each vertex $u \in V(g)$. The directed edge $e = (u, v)$ represents an edge from vertex u to v, and its label is denoted by $s(e)$, indicating a sequence number of the message in $E(g)$. In order for each vertex in the ESG to be uniquely identified, each vertex is assigned an identifier (ID), which is represented by a lower-case letter with an index. The ESG will not contain self-loops or multiple edges.

Algorithm 1 describes how the ESG g is created. We first prepare an array P that is used to update LVD. Each vertex of message M is added to $V(g)$ as a new vertex with an ID depending on its presence in the LVL (Line 7, 13). Then, a source vertex and destination vertices in each message M are connected to an edge with a sequence number t. Finally, all changes in $V(g)$ are restored in LVD (Lines 18 – 23).

Consider the example of applying the above graph creation rules in Fig. 2. In Fig. 2(a), the message M_1 is represented by a set of vertices $\{u_1, u_2, u_3\}$ and a set of edges $\{(u_1, u_2), (u_1, u_3)\}$. The labels and IDs of the vertex set are stored into LVL and LVI, respectively. In Fig. 2(b), a vertex u_4 and an edge (u_1, u_4) are newly created. Then, u_2 in LVI is replaced with u_4. Also, in Fig. 2(c), a vertex u_5 and a set of edges $\{(u_4, u_1), (u_4, u_3), (u_4, u_5)\}$ are added into ESG. Fig. 2 is self-explanatory for the remaining steps and can be understood easily.

In the graph creation rule, the source and destination addresses of CAN messages are required to connect the corresponding vertices. However, ECUs have no addresses in the CAN network. ECUs only filter out the CAN messages that they cannot handle, using the IDs of the CAN messages. Therefore, the CAN message has no source and destination addresses. This problem can be solved by utilizing a CAN database file[1] which contains ID information for each CAN message, such as the source and destination ECUs.

6. Event sequence–based graph distance

Given a graph database and a query graph, many applications for biological, chemical, and protein interaction networks exploit graph distance measures based on MCS distance or GED. Unfortunately, when these graph distance measures are used for ESGs, a serious problem might occur. For example, consider a query graph q and data graphs g_1 and g_2 in Fig. 3. The figure clearly shows that q is more similar to g_1 than g_2, because g_1 shares more common part with q than g_2. However, the measurement of GEDs indicates that g_2 has a lower distance value than g_1; $GED(q, g_1) = 9$ and $GED(q, g_2) = 8$. The symptom is worse with the MCS distance; $MCS_{dist}(q, g_1) = 12$ and $MCS_{dist}(q, g_2) = 7$. The MCS distance and GED commonly determine that q is more similar to g_2 than g_1. This results from the fact that both the metrics check the edge labels in calculating the distance of the two graphs. On the other hand, assume that the example graphs in Fig. 3 have no edge labels. The MCS distance and GED then incur another problem. Consider the two graphs q and g_3. Indeed, even though the two graphs q and g_3 are different from each other, they are proven to be graph isomorphic. Therefore, if we ignore the edge labels on the graphs, the results of queries on the graph database can contain false negatives.

In order to avoid the above problem, there is a clear need to adopt weighting techniques based on the mapping between the edges of two graphs. A number of studies have been carried out

[1] http://vector.com/vi_candb_en.html

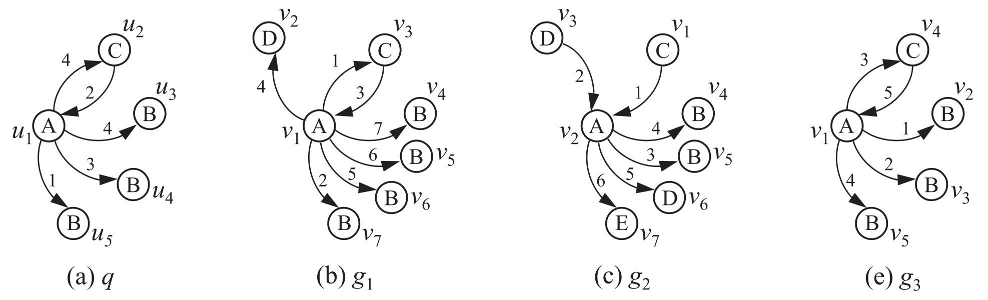


Fig. 3. Example of query and data graphs.

on the weighting techniques for keyword search (Kargar and An, 2011; Tran et al., 2009), traffic analysis (Djidjev et al., 2011), relational database (Kasneci et al., 2009), and co-authorship network (Qin et al., 2009), where the edge labels represent the shortest path, session timeout, taxonomic relations, and the number of citations, respectively. Since the target applications of the previous methods differ, they are difficult to directly apply to our problem. Hence, we devise another graph distance metric for our problem, which measures weights calculated by mapping each edge between two graphs.

To get an exact graph distance measure in our study, we first need to extract MCS between the two graphs without edge labels. Then, the total weight calculated by edge mapping of the MCS pair is calculated. This approach, however, may encounter a non-trivial issue. Consider the two graphs q and g_1 in Fig. 3. In order to obtain the MCS pair between q and g_1, we can choose a vertex mapping order such as $q' = (u_1, u_2, u_3, u_4, u_5)$ and $g_1' = (v_1, v_3, v_4, v_5, v_6)$, where $q' \subseteq q$ and $g_1' \subseteq g_1$. Similarly, if we select a mapping order $(v_1, v_3, v_4, v_5, v_7)$ in g_1 for q', we can get another MCS pair between q and g_1. Therefore, when obtaining the MCS of the two graphs, we can obtain multiple MCS pairs according to the vertex mapping order. So, we need to find all MCS pairs to exactly calculate the graph distance.

Definition 6 (Event sequence graph distance)**.** The MCS pair between two ESGs q and g is denoted by (q', g'), where we do not consider the edge labels of the two ESGs. The set of all MCS pairs of the ESGs q and g is denoted by $Ms(q, g)$, where we briefly denote it by Ms when its meaning is obviously clear. The ESG distance, denoted as $Edist(q, g)$, is defined as follows:

$$Edist(q,g) = MCS_d(q,g) + \min_{(q',g')\in Ms} w\left(q',g'\right), \tag{1}$$
where $w\left(q',g'\right)$ is explained indetail in Section 8.2.

In Eq. (1), we first find all MCS pairs between two the ESGs q and g without edge labels. We then find the MCS pair (q', g') that minimizes $Edist(q, g)$ using the minimum weight mapping function $w(q', g')$. In Section 8, we explain the algorithm to find all MCS pairs and the minimum weight mapping function $w(q', g')$ in detail.

7. Graph filtering and edge pruning

A straightforward approach to get the top-k answer graphs from the graph database is to compute the smallest ESG distance (Edist) for each Ms of the query graph and each data graph in the graph database. However, the MCS computation of two arbitrary graphs has been proven to be NP-hard, which requires very costly computation. Our strategy, in order to reduce the overhead of the MCS computation, basically follows the approach of Zhu et al. (2012). Their basic idea is to reduce the number of MCS computations by filtering unqualified candidate graphs from the graph database. For this, they use the lower bound of the MCS distance. In our work, we use a more efficient lower bound method, namely the *neighborhood label-based* lower bound, which is based on the frequency of distinct edges incident to a vertex. In addition, we incorporate an edge pruning method that increases the speed of MCS computation.

Algorithm 2 outlines our strategy to obtain the top-k answer graphs from the graph database. We first define a max-priority queue QA and a min-priority queue QC (Lines $1-2$). Each queue contains elements that are composed of the pair P = (gi, dist), where gi and dist denote the ID of data graph g and the distance value between the query graph q and the data graph g, respectively. Each dist in QA and QC is initialized to ∞ and 0, and is sorted in each queue. We calculate the lower bounds of all data graphs in D and insert each P into QC (Lines $3-6$). Then, we compute Edist between q and g by calling CalculateEdist, popping the element with the minimum dist in QC iteratively (Lines $8-14$). Before calculating Edist, we reduce the search space of the MCS calculation by performing NonSearchPathFilter (Line 13). If the Edist is smaller than the maximum Edist in QA, we remove the current element with the maximum Edist in QA, and insert the new element into QA. We stop the above process if the minimum lower bound in QC is not less than the maximum Edist in QA (Lines $10-11$). This is because the Edist between q and the current g with the minimum lower bound in QC must be not less than the maximum Edist in QA. Finally, we return the top-k answer graphs from QA, where the size of the answer graphs can be less than k.

In the following subsections, we explain in detail the two efficient methods adopted in Algorithm 2 to reduce the overhead of MCS computation.

7.1. Neighborhood label-based lower bound

As described in Section 6, the MCS distance can be used to measure the similarity between a query graph q and a data graph g. However, the exact MCS distance computation in Eq. (1) is very expensive. Therefore, to reduce the cost for the exact graph distance computation, we need an efficient lower bound method. In this section, we introduce our novel neighborhood label-based lower bound method.

The proposed neighborhood label-based lower bound method is based on the frequency of distinct edges incident to a vertex u. We call these distinct edges as the *neighborhood label set* of the vertex u. Given a query graph q and a data graph g, we derive neighborhood label sets from each graph and then calculate our lower bound in each neighborhood label set of the two graphs q and g. The neighborhood label set is defined as follows.

Definition 7 (Neighborhood label set)**.** Given $u \in V(q)$, the neighborhood label set of u, denoted by $NLS(u)$, is a set of 2-tuple $LS(u) = (ES(u), \delta(u))$, where each $ES(u)$ is a distinct edge incident to the vertex u. $ES(u)$ can be represented as $(L(u), L(v))$, where v is the vertex adjacent to u. Also, $\delta(u)$ is the number of occurrences of the $ES(u)$ among the edges incident to u.

Algorithm 2 GetAnswer(q, D, k).

```
Input: q: a query graph, D: graph database, k: number of answer graphs
Output: top-k answer graphs with the smallest Edist
{
 1:  $Q_A$ ← a priority queue containing a data graph g and the Edist between q and g;
 2:  $Q_C$ ← a priority queue containing a data graph g and the lower bound between q and g;
 3:  for all {g | g ∈ D} do {
 4:      $dist_{LO}$ ← Get$NLS_{LO}$(q, g);
 5:      $Q_C$.push(($g_i$, $dist_{LO}$));
 6:  }
 7:  while not $Q_C$.empty() do {
 8:      c ← $Q_C$.pop();
 9:      a ← $Q_A$.top(); // get the element with the maximum Edist in $Q_A$
10:      if a.dist ≤ c.dist then
11:          break;
12:      get the data graph g with $c.g_m$;
13:      NonSearchPathFilter(q, g);
14:      Edist ← CalculateEdist(q, g);
15:      if a.dist > Edist then
16:          $Q_A$.pop();
17:          $Q_A$.push(($g_m$, Edist));
18:  }
19:  return the top-k answer graphs in $Q_A$;
}
```

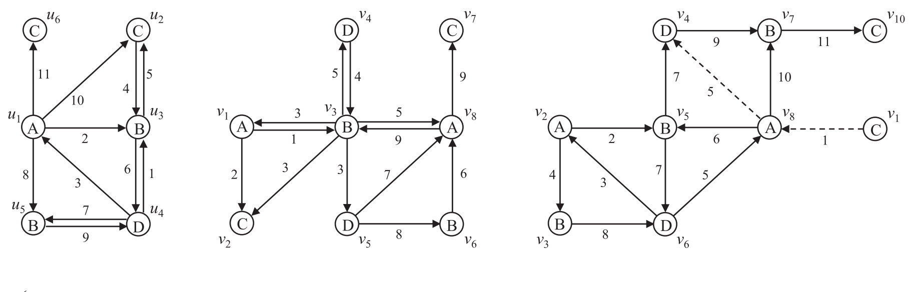


Fig. 4. A running example of our non-search path filtering.

Each $LS(\cdot)$ may appear multiple times in calculating the neighborhood label sets of a graph. We denote the union of NLSs as $_{U}NLS(\cdot)$, which is defined as follows. (In this case, union refers to multi-set union.)

$$_{U}NLS(q) = \cup_{u\in V(q)} NLS(u) \tag{2}$$

Now, given the two graphs q and g, we calculate the neighborhood label-based lower bound, denoted by $NLS_{LO}(q,\ g)$, for $MCS_{dist}(q,\ g)$ using $_{U}NLS(q)$ and $_{U}NLS(g)$. $NLS_{LO}(q,\ g)$ is defined as follows:

$$NLS_{LO}(q,g) = |E(q)| + |E(g)| - 2 \times \left(\sum_{n=1,\ m=1\ and\ u\in V(q),\ u'\in V(g)}^{n<|_{U}NLS(q)|,\ m<|_{U}NLS(g)|} \rho\left(LS(u)_n, LS(u')_m\right) \right), \tag{3}$$

where

$$\rho\left(LS(u)_n, LS(u')_m\right) = \begin{cases} \min\left(\delta(u)_n, \delta(u')_m\right) & \text{if } ES(u)_n = ES(u')_m \\ 0 & \text{otherwise.} \end{cases}$$

To speed up the computation of $NLS_{LO}(q,\ g)$ in Eq. (3), we change the vertex label into a number, such as A→0, B→1, and C→2. For example, for graph q in Fig. 4(a), $NLS(u_1)$ = {((A, B), 2), ((A, C), 2)} can be represented as {((01), 2), ((02), 2)}. Then, we sort the elements of $_{U}NLS(\cdot)$ in non-increasing order. Using the two multi-sets $_{U}NLS(q)$ and $_{U}NLS(g)$ sorted in the non-increasing order, we quickly compare each element of $_{U}NLS(q)$ and $_{U}NLS(g)$. Algorithm 3 explains Eq. (3) in detail. In the algorithm, ExtractNLS returns $_{U}NLS(\cdot)$ of a graph sorted in the non-increasing order (Lines 2 – 3). Then, for each $LS(u)$ from $_{U}NLS(q)$ and each $LS(u')$ from $_{U}NLS(g)$, we calculate the sum of $\rho(LS(u)_n,\ LS(u')_m)$ (Lines 4 – 12). We finally calculate the lower bound between the two graphs q and g (Line 13).

Algorithm 3 GetNLS_{LO}(q, g).

```
Input: q: a query graph, g: a data graph
Output: $NLS_{LO}$ between q and g
{
 1:  n ← 0, m ← 0, μ ← 0;
 2:  $_{U}NLS(q)$ ← Extract$NLS(q)$;
 3:  $_{U}NLS(g)$ ← Extract$NLS(g)$;
 4:  while n < |$_{U}NLS(q)$| && m < |$_{U}NLS(g)$| do {
 5:      if $ES(u)_n = ES(u')_m$ then
 6:          μ ← μ + min($\delta(u)_n$, $\delta(u')_m$);
 7:          n ← n + 1; m ← m + 1;
 8:      else if $ES(u)_n > ES(u')_m$ then
 9:          n ← n + 1;
10:      else
11:          m ← m + 1;
12:  }
13:  $NLS_{LO}$ ← |E(q)| + |E(g)| − 2μ;
14:  return $NLS_{LO}$;
}
```

For instance, for graph q in Fig. 4(a), $_{U}NLS(q)$ = {(A, B), 2), ((A, C), 2), ((B, C), 1), ((B, D), 1), ((B, D), 1), ((C, B), 1), ((D, A), 1), ((D, B), 2)} and $_{U}NLS(q)$ converted and sorted in the non-increasing order is {((31), 2), ((30), 1), ((21), 1), ((13), 1), ((13), 1), ((12), 1), ((02), 2), ((01), 2)}. Similarly, for d_1 in Fig. 4(b), $_{U}NLS(d_1)$ = {((A, B), 1), ((A, B), 1), ((A, C), 1), ((A, C), 1), ((B, A), 2), ((B, A), 1), ((B, C), 1), ((B, D), 2), ((D, A), 1), ((D, B), 1), ((D, B), 1)} and $_UNLS(d_1)$ converted is {((31), 1), ((31), 1), ((30), 1), ((13), 2), ((12), 1), ((10), 2), ((10), 1), ((02), 1), ((02), 1), ((01), 1), ((01), 1)}. Therefore, the number of common edges between $_UNLS(q)$ and $_UNLS(d_1)$ is 6, while the actual $|E(MCS(q, d_1))|$ is 5. When using $dist_1$ (Zhu et al., 2012) and $dist_2$ (Raymond and Willett, 2002b) for this example, the number of the common edges in each method is 10 and 6, respectively. Eventually, the $dist_1$, $dist_2$, NLS_{LO}, and actual MCS distance between q and d_1 are 4, 12, 12, and 14, respectively.

The neighborhood label-based lower bound is less tight compared against $dist_2$ (Raymond and Willett, 2002b), where its time complexity is $O(n\log n)$ and $n = \max\{|E(q)|, |E(g)|\}$. In general, calculating a tighter lower bound for the MCS distance requires higher computational cost. Therefore, there is a tradeoff between the tightness of the lower bound and the computational cost. In Section 9, we will discuss these tradeoffs in detail.

7.2. Edge pruning with non-search path filtering

In our work, we use a backtrack-based search to obtain all MCS pairs. In each step of the backtrack-based search, edge pruning can reduce the entire search space. Non-search path filtering is simple. Given a query graph q and data graph g, its goal is to reduce the number of vertex visits during the MCS search process by pruning out the edges in $DES(g) - DES(q)$ from the graph g, where $DES(g)$ and $DES(q)$ denote the sets of distinct edges in the graphs g and q, respectively.

Consider a running example in Fig. 4, where $DES(q)$ = {(A, C), (A, B), (B, C), (B, D), (C, B), (D, A), (D, B)}. In Fig. 4(c), the edge $(L(v_8), L(v_4))$ of d_2 is not an element of $DES(q)$, i.e., (A, D) $\notin DES(q)$. The non-search path filtering removes these edges (denoted by the dotted line) from the graph d_2 because they have no effect in computing the similarity of the two graphs q and d_2. Suppose one hundred vertices have label D and are adjacent to $v_8 \in V(d_2)$. If we try to match a vertex v_8 in d_2 with a vertex u_1 in q on the search space, the number of attempts to traverse the child vertices of v_8 will be one hundred. Thus, in the DES-filtered graph shown in Fig. 4(c), we do not need to explore irrelevant edges in the search space. The time complexity of our non-search path filtering is $O(|E(g)|)$.

8. Query graph processing

8.1. MCS algorithm based on backtracking

As mentioned in Section 6, we find all MCS pairs between the two graphs, and then, measure the total weight calculated by the edge mapping of the MCS pairs using the minimum weight mapping function. Algorithm 4 describes the overall process of calculating the distance between the ESGs. Input parameters to the algorithm CalculateEdist are query graph q and data graph g, where g is a candidate graph generated from the process described in Section 7. To find all MCS pairs between the two graphs, CalculateEdist maintains two arrays, CM_V and CM_E, that represent the set of current vertex pairs (u, u') and the set of current edge pairs $((u, v), (u', v'))$, respectively, i.e., the vertex and edge pairs used for the current mapping in the search space, where $u, v \in V(q)$ and u', $v' \in V(g)$. We first choose the root vertex pair u and u' that has the same vertex label (Lines 2 – 3). We then insert the vertex pair into the CM_V (Line 4). Then, to find all MCS pairs, we call the recursive function MCSSearch (Ms, u, u'), which will be explained in the next paragraph, and all the MCS pairs are stored in Ms (Line 5). Then, we calculate the minimum weight by edge mapping of each MCS pair (q', g') in Ms by calling CalcMinWM$((q', g'))$ (Lines 9 – 13). Finally, we return Edist of the two graphs q and g using the minimum weight from the MCS pairs (Lines 14 – 15).

Algorithm 4 CalculateEdist(q, g).

```
Input: $q$: a query graph, $g$: a data graph
Output: Edist
{
 1:  $Ms \leftarrow$ empty MCS pair set between $q$ and $g$;
 2:  for all $\{u \mid u \in V(q)\}$ do {
 3:    for all $\{u' \mid u' \in V(g)$ and $L(u) = L(u')\}$ do {
 4:      $CM_V \leftarrow (u, u')$;
 5:      MCSSearch($Ms$, $u$, $u'$);
 6:      remove $(u, u')$ from $CM_V$;
 7:    }
 8: }
 9:  for all $\{(q', g') \mid (q', g') \in Ms\}$ do {
10:    $w \leftarrow$ CalcMinWM($q'$, $g'$);
11:    if ($w < {}_{min}W$)
12:      ${}_{min}W \leftarrow w$;
13: }
14:  Edist $\leftarrow |E(q)| + |E(g)| - 2 \times |E(MCS(q, g))| + {}_{min}W$;
15:  return Edist;
}
```

The recursive function MCSSearch to obtain all MCS pairs is shown in Algorithm 5. In MCSSearch, we count the number of maximum common edges using the parent-child relationships of vertices. In Algorithm 5, we denote the parent vertex as u_p and the child vertex as u_c. We first generate a set of candidate child vertex pairs by combining child vertices of u_p and child vertices of u'_p using CreateCandiSet(u_p, u'_p) (Line 1). Each vertex of the child vertex pair (u_c, u'_c) must have the same vertex label, i.e., $L(u_c) = L(u'_c)$. In order to prune out non-promising child vertex pairs, in each iteration of the loop, IsSuitablePath((u_p, u_c), (u'_p, u'_c)) checks the following two pruning rules. The first rule checks the mapping relation of the vertex pair (u, u') in CM_V. When a vertex pair (u, u') exists in CM_V, we check if the already mapped vertex u of q has the corresponding vertex u' of g. The second rule checks whether the edges (u_p, u_c) and (u'_p, u'_c) do not exist in CM_E. If the child vertex pair (u_c, u'_c) is not qualified, we prune out the child vertex pair without exploring it (Lines 2 – 3). If the child vertex pair (u_c, u'_c) is suitable, we update CM_V and CM_E with the vertex and edge pair, respectively (Lines 4 – 5). In order to obtain Ms, if $|CM_E|$ between q and g is greater than the number of edges discovered so far, denoted as ${}_{max}EC$, UpdateMs(Ms, CM_V, CM_E) generates the current MCS pair (q', g') (Lines 6 – 9). In UpdateMs(Ms, CM_V, CM_E), if $|CM_E|$ of the MCS pair (q', g') is equal to ${}_{max}EC$, we add it to Ms. Otherwise, if $|CM_E|$ of the MCS pair is greater than ${}_{max}EC$, Ms only has the MCS pair (q', g'). Then, we recursively call MCSSearch itself for the child vertex of u_c (Line 10).

For instance, consider a running example shown in Fig. 5. The running example represents the search space of vertex matching between q and d_2 in Fig. 4. In the search space, the root vertex pair (u_1, v_2) is first matched. The u_1 has four child vertices u_2, u_3, u_5, and u_6 whose labels are C, B, B, and C, respectively. In contrast, v_2 has two child vertices v_3 and v_5 whose labels are all Bs. The child vertex pairs of the root vertex pair (u_1, v_2) are made of the combination among child vertices having the same label, as shown in Fig. 5(1). The child pair (u_1, v_8) in Fig. 5(2) is pruned by IsSuitablePath, because the u_1 has already been matched with v_2 in the root vertex pair. In Fig. 5(3), two MCS pairs exist between q and d_2. In case that there is an additional path with arbitrary child vertex pairs as shown in Fig. 5(4), Update Ms generates a new MCS pair and the Ms has the new MCS pair only.

8.2. Optimal mapping computation

In Section 8.1, we discussed an algorithm to obtain all MCS pairs between the two ESGs q and g without edge labels. In this section, for each MCS pair (q', g') between the two ESGs q and

Algorithm 5 MCSSearch(Ms, u_p, u'_p).

```
Input: $Ms$: a set of all MCS pairs, $u_p$ and $u'_p$: a vertex pair, where $u_p \in V(q)$, $u'_p \in V(g)$
{
 1:  for each ($u_c$, $u'_c$) ∈ CreateCandiSet($u_p$, $u'_p$) do {
 2:    if not IsSuitablePath(($u_p$, $u_c$), ($u'_p$, $u'_c$)) then
 3:      continue;
 4:    $CM_V \leftarrow (u_c, u'_c)$;
 5:    $CM_E \leftarrow ((u_p, u_c), (u'_p, u'_c))$;
 6:    if ($|CM_E| > {}_{max}EC$) then {
 7:      ${}_{max}EC \leftarrow |CM_E|$;
 8:      UpdateMs($Ms$, $CM_V$, $CM_E$);
 9:    }
10:    MCSSearch($Ms$, $u_c$, $u'_c$);
11:    remove ($u_c$, $u'_c$) from $CM_V$;
12:    remove (($u_p$, $u_c$), ($u'_p$, $u'_c$)) from $CM_E$;
13: }
14:  return;
}
```

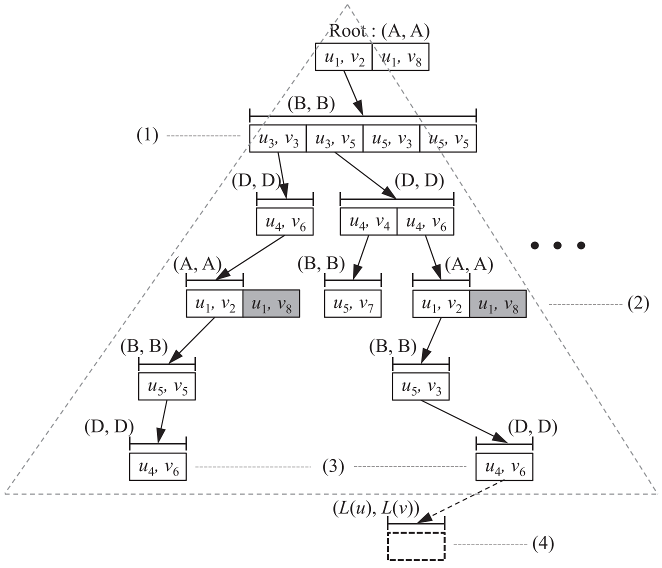


Fig. 5. A search space between q and d_2.

g, we explain how to compute the optimal mapping using the minimum weight mapping function $w(q', g')$ in order to calculate the exact distance between the two ESGs. As mentioned in Definition 5, each edge e in ESG has label $s(e)$ that indicates a sequence number of the message. For each $e \in E(q')$ and $e' \in E(g')$ of the MCS pair (q', g'), where $E(q')$ and $E(g')$ obviously have the same cardinality, $w(q', g')$ finds the mapping that minimizes the total weight calculated by mapping e to e', considering the sequence number of the message in each edge. Here, we need to consider the range of $w(q', g')$ between the two ESGs q and g. We first measure $MCS_{dist}(q, g)$ to calculate the Edist of the two ESGs q and g. The number of $MCS_{dist}(q, g)$ is calculated based on the number of common edges of the two ESGs. This means that the value of $w(q', g')$ should not affect the value of $MCS_{dist}(q, g)$. For this purpose, the range of $w(q', g')$ should be in [0, 1).

Definition 8 (Minimum weight mapping function)**.** Given the MCS pair (q', g') between graph q and g, the minimum weight mapping function, denoted as $w(q', g')$, is defined as follows:

$$w(q', g') = \min_{W} \sum_{e \in E(q'),\ e' \in E(g'),\ q' \subseteq q,\ and\ g' \subseteq g} c(e, W(e')),$$

where

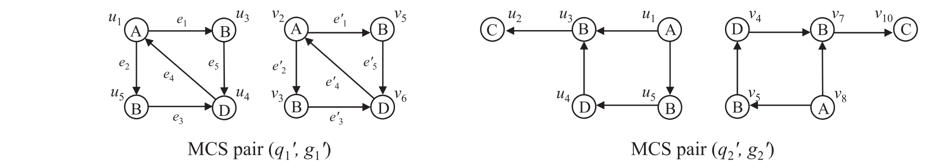
$$c(e, e') = \begin{cases} 0 & if\ s(e) = s(e') \\ \left| \frac{s(e)}{s_{\max}(q)+1} - \frac{s(e')}{s_{\max}(g)+1} \right| & otherwise \end{cases}, \ and \tag{4}$$

$$s_{\max}(q) = \max\{s(e) | e \in E(q)\}$$

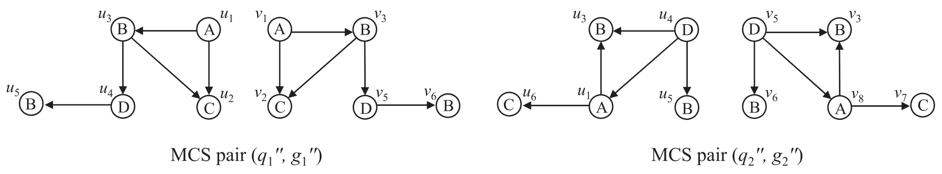
In Eq. (4), the mapping W is a bijective function. Also, the range of $s(e)$ is in $[1, s_{max}(q)]$. The weight calculated by mapping e to e', denoted by $c(e, e') \in [0, 1)$, is normalized by dividing $s(e)$ by $s_{max}(q)$.

Fig. 6 provides a set of the MCS pairs between the query graph and data graphs in Fig. 4. Fig. 6(a) denotes two MCS pairs (q_1', g_1') and (q_2', g_2') between q and d_2, where $q_1', q_2' \subseteq q$ and $g_1', g_2' \subseteq d_2$. Similarly, two MCS pairs (q_1'', g_1'') and (q_2'', g_2'') exist between q and d_1 as shown in Fig. 6(b). To facilitate explanation, we assign an edge identification, such as $e_1, e_2, \ldots, e_n$, to each edge in q_1' and g_1'. Once we do not consider the edge labels on q_1' and g_1', the edge e_1 can be mapped to e'_1 or e'_2, because their endpoint vertices have the same labels. Similarly, e_5 can be mapped to e'_3 or e'_5. As shown in Fig. 7, this mapping relationship can be expressed as a bipartite graph with $E(q_1')$ on the left side and $E(g_1')$ on the right side. In the bipartite graph, an edge exists between $e \in E(q_1')$ and $e' \in E(g_1')$ if their endpoint vertices have the same label.

In order to obtain the best mapping for individual edges in the bipartite graph, we use an $n \times n$ matrix, where each cell in the matrix has a value of $c(e, e')$, as shown in Fig. 7. Note that if no edge exists between e and e' in the bipartite graph, the cell value is defined as infinite. Finding the best mapping on the matrix is equivalent to finding combinatorial optimization such that the sum of $c(e, e')$ on the matrix is minimized, where each edge e in $E(q_1')$ is mapped to exactly one edge e' in $E(g_1')$. In order to solve this problem, we use the Munkres algorithm (Munkres, 1957). Through calculating $w(q', g')$ of each MCS pair (q', g') between the two ESGs q and d_2, we finally get the minimum weight in all MCS pairs between the two ESGs. Given the query graph and data graphs in Fig. 4, if we use the conventional graph distance metric such as MCS distance, d_1 and d_2 with the same graph distance value will be returned as answers since $MCS_{dist}(q, d_1) = MCS_{dist}(q, d_2) = 14$. On the other hand, our method Edist can return better answers by computing the optimal mapping for the sequence number in each edge between the MCS pairs. In this way, we can find the top-k answer graphs from the large graph database, avoiding false negatives.

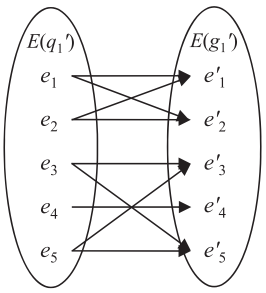


(a) MCS pairs (q_1', g_1') and (q_2', g_2') between the two graphs q and d_2.



(b) MCS pairs (q_1'', g_1'') and (q_2'', g_2'') between the two graphs q and d_1.

Fig. 6. A set of MCS pairs between the query graph and data graphs in Fig. 4.



	e'_1	e'_2	e'_3	e'_4	e'_5
e_1	$c(e_1, e'_1)$	$c(e_1, e'_2)$	∞	∞	∞
e_2	$c(e_2, e'_1)$	$c(e_2, e'_2)$	∞	∞	∞
e_3	∞	∞	$c(e_3, e'_3)$	∞	$c(e_3, e'_5)$
e_4	∞	∞	∞	$c(e_4, e'_4)$	∞
e_5	∞	∞	$c(e_5, e'_3)$	∞	$c(e_5, e'_5)$

Fig. 7. A bipartite graph (left) and $n \times n$ matrix (right) in MCS pair (q_1', g_1').

9. Experimental results

We performed experiments to assess the efficiency of our approach compared to the expert-based method and similarity matching algorithms. We first show the efficiency of our approach by providing the accuracy and the running time compared to the method adopted by expert technicians at the Hyundai Motor Company. We then compared our Edist with the two algorithms gs-topk-noidx (GTN) (Zhu et al., 2012) and m-edge-i (Hu et al., 2013) that are based on the MCS approach. Additionally, we evaluated the performance of GED based algorithms SAPPER (Zhang et al., 2010) and NH-TA (Ding et al., 2014) compared to Edist. Then, we conducted experiments to compare the running time and the *tightness ratio* in terms of $dist_1$ (Zhu et al., 2012), $dist_2$ (Raymond and Willett, 2002b), and NLS_{LO}. We use two kinds of datasets in our experiments, a real dataset and a synthetic dataset, as summarized below:

Real dataset. A real graph dataset with 1,012 graphs was generated from the CAN message log provided by the Hyundai Motor Company. The CAN message log includes network messages generated from vehicle operations such as a door lock device, an anti-lock braking system, a tire-pressure monitoring system, and a smart key. The real dataset has an average number of 48.7 vertices and 1,935.3 edges, and a maximum number of 148 vertices and 2,872 edges.

Synthetic dataset. A synthetic dataset with 40,000 graphs was generated based on a CAN database that consists of the 58 ECUs and 127 message IDs. In order to obtain a set of vehicle operation patterns, we arbitrarily configured the message sequences from the CAN database. Then, we generated the synthetic graph dataset using Algorithm 1 in Section 5. The synthetic dataset has an average number of 62.4 vertices and 2,682.8 edges, and a maximum number of 300 vertices and 3,914 edges.

For the CAN message log used in the real dataset, we artificially injected a simulation-based fault (Suwatthikul et al., 2011). The simulation-based fault is classified into the analogue disturbance (denoted by *ad*) and the digital disturbance (denoted by *dd*). The *ad* generates faults by distorting the CAN signal lines. Meanwhile, the *dd* generates faults by simulating the CAN signal directly during message communication. To inject the simulation-based faults into the CAN message log, we exploited a CAN simulation tool, Vector CANoe, which simulates CAN bus with CAN message logs. Then, we used an ECU to inject the *ad* and the *dd* into the CAN bus during the simulation. For the synthetic dataset, we injected a random fault. To this end, we used a query graph that is a data graph selected from the synthetic dataset and then modified based on the *error ratio*. The error ratio is the ratio of the number of newly inserted vertices or edges in the query graph. In the case of a vertex insertion (denoted by *vi*), the newly inserted vertex has only a single edge, while an edge insertion (denoted by *ei*) connects two arbitrary vertices in the query graph using a directed edge.

We evaluate several experiments by varying the size of the query graph up to the maximum size of each graph dataset. The default value of the query graph is the average vertex size of each graph dataset. All the experiments were performed on a Windows 7 platform with a 1.7 GHz Intel i5 processor and 4GB memory.

9.1. Efficiency results

In order to detect fault operations in the large number of CAN message logs, the expert technicians exploit Vector CANalyzer, which provides network monitoring among the ECUs for offline analysis of CAN message logs. Their general approach using the analysis tools is to reduce the amount of data for analysis by filtering the entire data using simple conditions or predefined values. For example, they can configure predefined values such as mes-

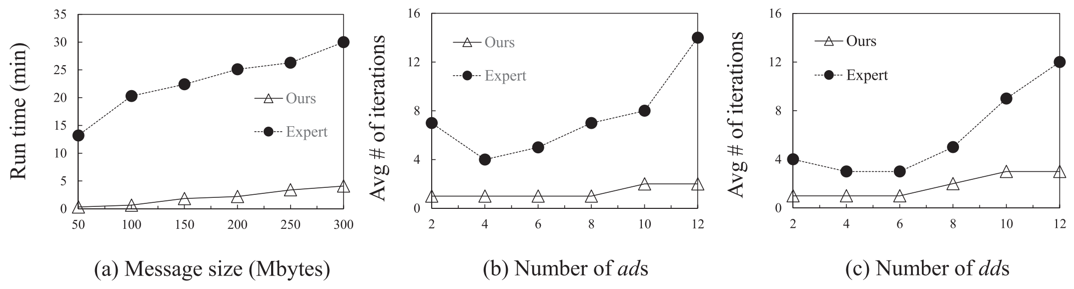


Fig. 8. Comparison results.

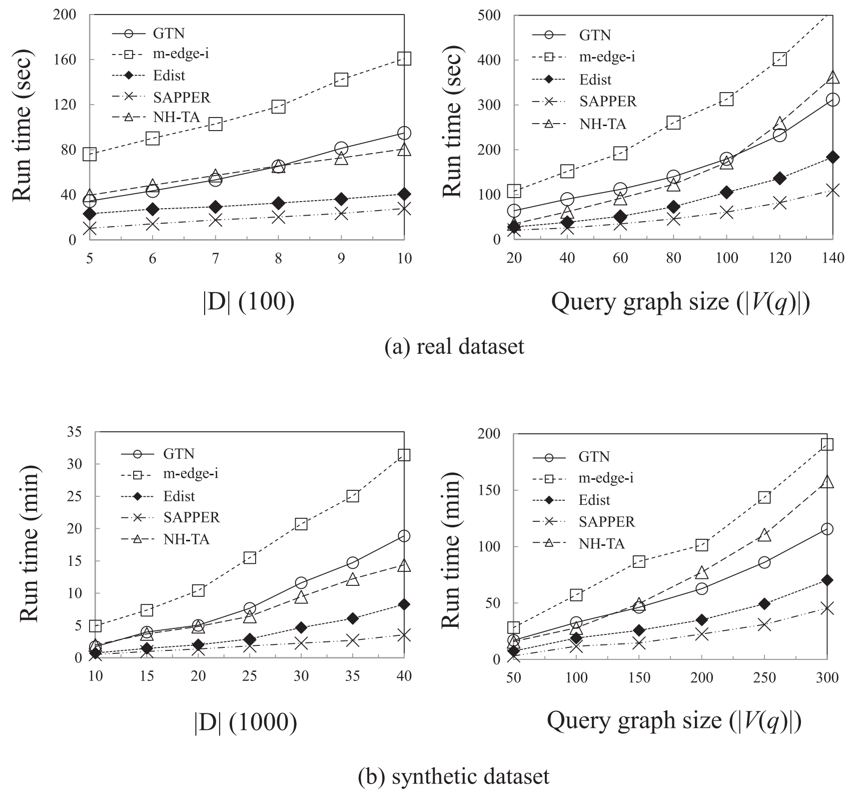


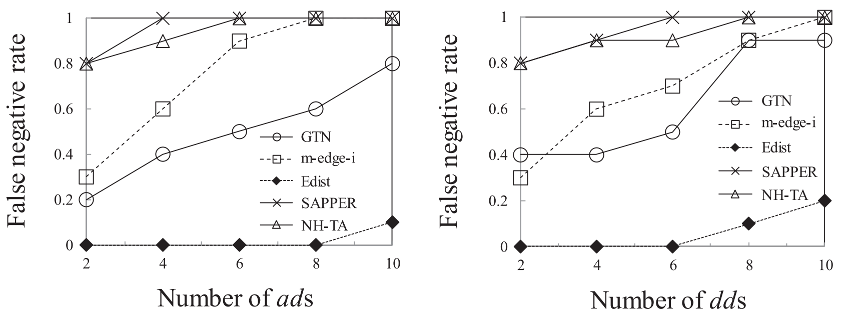
Fig. 9. Performance results.

sage IDs, periods, and the number of messages as filtering values to easily trace message logs (Vector, 2010). Then, the technicians determine the occurrence of faults by analyzing abnormal patterns in the filtered message logs. Thus, given the large number of CAN message logs, it is hard for the technicians to specify predefined values suitable for filtering the message logs. As a result, they may need to try several attempts to obtain a truly useful set of answers. In our experiment, the predefined values were configured based on the experience of the expert technicians. In order to compare the efficiency of our method with the expert-based method, we measured the total running time and the number of iterations attempted to find the actual fault operation. The total running time for the analysis of CAN message logs reflects the total time taken to detect fault operations injected based on the simulation-based fault.

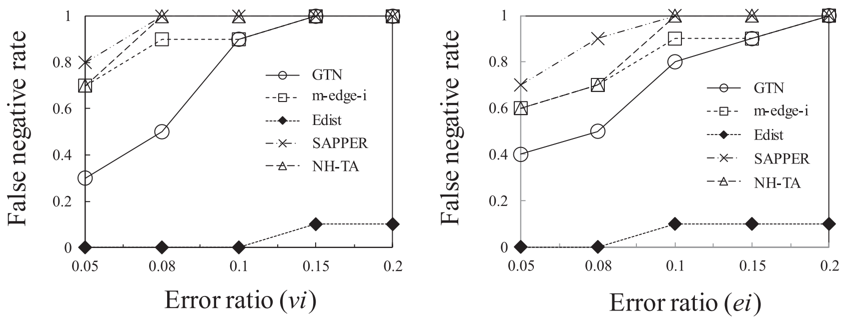
Fig. 8(a) shows the running time for the size of the accumulated message log for the two methods. Based on this figure, we could observe that our method could significantly speed up the fault detection processes in the message logs. We also compared the accuracy of the two methods. The accuracy is computed as the average number of iterations required to detect actual fault operations. To this end, our method used the top-1 answer. If the actual fault operation was not found in the top-1 answer, then *k* was increased. Fig. 8(b) and (c) show the average number of iterations for the two methods. We could observe that the average number of iterations for the expert-based method decreased as we increased the number of *ads* (*dds*) up to 4(6). However, when we further increased the number of simulation-based faults, the average number of iterations for the expert-based method increased, because the range of data to be analyzed by the experts was significantly increased. On the other hand, the average number of iterations of our method was linearly scaled.

9.2. Performance of query processing

We evaluated the performance of Edist against the state-of-the-art algorithms, GTN, m-edge-i, SAPPER, and NH-TA. GTN and m-edge-i using the MCS based similarity measure find only one MCS pair in their procedures. For fair comparison, we need to modify their algorithms, since we find all MCS pairs between two graphs. Fortunately, GTN and m-edge-i are based on a backtracking approach. Thus, we could slightly modify the algorithms so that they can produce all MCS pairs between the two graphs. On the other



(a) the accuracy of the simulation-based fault in real dataset



(b) the accuracy of the random-based fault in synthetic dataset

Fig. 10. Top-k accuracy of answer graphs.

hand, SAPPER and NH-TA, using the GED based similarity measure, find all approximate matches of the query graph in a data graph.

We first conducted experiments to measure the performance of GTN, m-edge-i, SAPPER, NH-TA, and Edist. For experiments conducted on the real and synthetic graph datasets, we randomly selected query graphs by varying the size of the data graph in each graph dataset. Then, we modified the query graphs using the error ratio ($vi = 0.1$ and $ei = 0.1$). Fig. 9 shows the performance results of GTN, m-edge-i, SAPPER, NH-TA, and Edist in the two graph datasets. Regarding the runtime, SAPPER is superior to that of the other methods, because SAPPER provides fast search performance by retrieving approximate matches of the query graph. The number of vertices for all matches found by SAPPER's searching schemes must be equal to the number of vertices in the query graph because SAPPER only considers edge additions in the GED calculation. Thus, the accuracy of the query results is not high. On the other hand, Edist is not only competitive in performance but also highly accurate.

We measured the accuracy of Edist by calculating the false negative rates of the algorithms in the graph datasets. To represent the false negative rate, the ranking number of answer graphs divided by k is computed, where the ranking number starts from 0. (e.g., $k = 10$; if the second answer is matched to the query graph, the false negative rate is 0.1). For each dataset, the simulation-based fault and the random-based fault were used again. Then, we fixed the graph datasets and varied the query graphs using the simulation-based fault and the random-based fault.

Fig. 10 shows the average false negative rates of GTN, m-edge-i, SAPPER, NH-TA and Edist for different query graphs. As shown in the figure, Edist substantially outperforms all the competitors for all the query graphs. In particular, Fig. 10(a) shows that Edist has perfect accuracy, with the number of *ads* up to 8 and the number of *dds* up to 6. This is due to the minimum weight mapping function of Edist. The simulation-based fault causes corruption to the CAN message, so the normal operation will have an inconsistent message duplicate (Pinho and Vasques, 2003) according to the message retransmission. As mentioned previously, the minimum weight mapping function calculates the exact distance between the two ESGs, considering the sequence number of the message in each edge. Regarding SAPPER and NH-TA, their methods, considering only an edge addition or deletion in the GED calculation, permit structural differences in each vertex. Thus, SAPPER and NH-TA generate many false candidates and cause serious inaccuracies in the graph datasets. Regarding GTN and m-edge-i, they show better performance than SAPPER and NH-TA. However, when we compare these two methods to Edist, we observe that the gap between them and Edist increase further as we increase the number of faults.

9.3. Evaluating performance of lower bound methods

We evaluated the performance of our lower bound method NLS_{LO} with lower bound methods $dist_1$ and $dist_2$. Indeed, m-edge-i also exploits the lower bound method to filter out non-promising data graphs from the graph datasets. However, given a query graph q and a data graph g, m-edge-i calculates the lower bound of $RE(q)$. $RE(q)$ is the number of relaxed edges that belong to a query graph q but not to MCS between the two graphs; i.e., $RE(q) = |E(q)| - |E(MCS(q, g))|$. Calculating the lower bound of $RE(q)$ is equivalent to calculating the upper bound of the MCS distance between the two graphs. Thus, we only compare our lower bound method with $dist_1$ and $dist_2$. In order to evaluate the performance of the lower bound methods of $dist_1$, $dist_2$, and NLS_{LO}, we reported the runtime and the *tightness ratio* of $dist_1$, $dist_2$, and NLS_{LO} by varying the graph size in the two datasets. The tightness ratio is defined as follows.

$$\tau = 1 - \left| \frac{MCS_{dist}(q, g) - \Delta dist(q, g)}{MCS_{dist}(q, g)} \right| \tag{5}$$

In Eq. (5), $\Delta dist(q, g)$ refers to the result of the lower bound methods between the two graphs q and g, such as $dist_1$, $dist_2$, and NLS_{LO}. The tightness ratio remains close to 1 if $\Delta dist(q,$

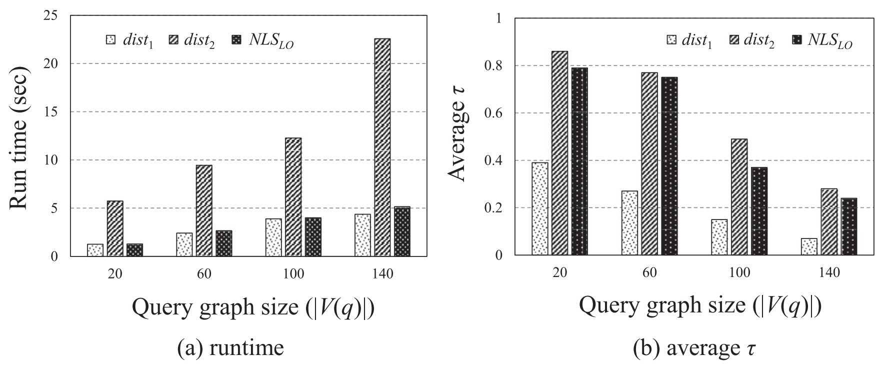


Fig. 11. Performance of lower bound methods over real dataset.

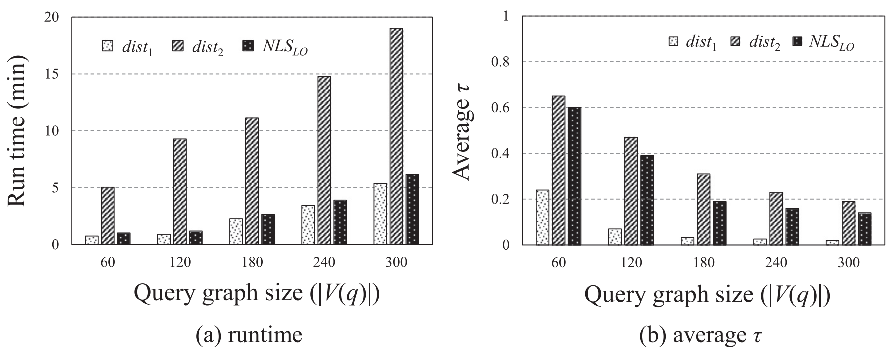


Fig. 12. Performance of lower bound methods over synthetic dataset.

$g) \approx MCS_{dist}(q, g)$. This means that the tightness ratio increases, the ability to filter out non-promising data graphs increases.

Figs. 11(a) and 12(a) show the run time required to compute $dist_1$, $dist_2$, and NLS_{LO} over the graph dataset. As shown in the figure, the runtime of $dist_2$ increases exponentially as the query size increases. We could also observe no significant differences between NLS_{LO} and $dist_1$ with respect to the runtime. In addition, we measured the tightness ratio of $dist_1$, $dist_2$, and NLS_{LO}. As shown in Figs. 11(b) and 12(b), we observed that the $dist_1$ was not very tight. In contrast, NLS_{LO} and $dist_2$ are much tighter than $dist_1$. NLS_{LO} is inferior to $dist_2$ in terms of the tightness ratio, but only with small differences.

10. Conclusion

In this article, we proposed a novel fault detection framework that uses graph pattern matching on the network message logs. In order to identify the unknown condition in the driving tests, we converted the network message logs into a graph. We then found fault operations in a large amount of in-vehicle network data using our efficient graph pattern matching scheme. To alleviate the limitations of conventional methods for graph pattern matching such as MCS distance and GED, we proposed a novel graph distance metric based on MCS using the minimum weight mapping function, which can avoid false negatives in the large database. We also introduced two efficient graph filtering techniques to reduce the computational cost. The experimental results showed that the proposed method outperforms the expert-based method and existing algorithms, and is very efficient in detecting fault operations in the message log of in-vehicle networks.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696) and by ICT R&D program of MSIP/IITP. [R0126-15-1065, (SW StarLab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays].

References

Collins, M.P., Reiter, M.K., 2007. Hit-list worm detection and bot identification in large networks using protocol graphs. In: Recent Advances in Intrusion Detection, pp. 276–295.
Conte, D., Foggia, P., Vento, M., 2007. Challenging complexity of maximum common subgraph detection algorithms. A performance analysis of three algorithms on a wide database of graphs. J. Graph Algorithms Appl. 11 (1), 99–143.
Davis, R.I., Burns, A., Bril, R.J., Lukkien, J.J., 2007. Controller Area Network (CAN) schedulability analysis: refuted, revisited and revised. Real Time Syst. 35 (3), 239–272.
Ding, X., Jia, J., Li, J., Liu, J., Jin, H., 2014. Top-k similarity matching in large graphs with attributes. In: DASFAA, pp. 156–170.
Djidjev, H., Sandine, G., Storlie, C., Vander Wiel, S., 2011. Graph based statistical analysis of network traffic. In: Proceedings of the Ninth Workshop on Mining and Learning with Graphs.
Fortin, S., 1996. The graph isomorphism problem. Department of Computing Science, University of Alberta. Hansson, H. A., Nolte, T., Norström, C., Punnekkat, S., 2002. Integrating reliability and timing analysis of CAN-based systems. IEEE Trans. Ind. Electron. 49 (6), 1240–1250.
Hu, H., Li, G., Feng, J., 2013. Fast similar subgraph search with maximum common connected subgraph constraints. In: IEEE International Congress on Big-Data, pp. 181–188.
ISO Standard 11898, 1993. Road Vehicles Interchange of Digital Information - Controller Area Network - ISO 11898. International Organization for Standardization.
Kargar, M., An, A., 2011. Keyword search in graphs: finding r-cliques. VLDB 4 (10), 681–692.
Kasneci, G., Ramanath, M., Sozio, M., Suchanek, F.M., Weikum, G., 2009. Star: steiner-tree approximation in relationship graphs. In: ICDE, pp. 868–879.
Keskin, U., 2013. Evaluating message transmission times in Controller Area Network (CAN) without buffer preemption revisited. In: Vehicular Technology Conference (VTC Fall), pp. 1–5.
Koch, I., Lengauer, T., Wanke, E., 1996. An algorithm for finding maximal common subtopologies in a set of protein structures. J. Comput. Biol. 3 (2), 289–306.
Mubeen, S., Mam-Turja, J., Sjödin, M., 2011. Extending schedulability analysis of controller area network (CAN) for mixed (periodic/sporadic) messages. In: Emerging Technologies & Factory Automation (ETFA), pp. 1–10.

Munkres, J., 1957. Algorithms for the assignment and transportation problems. J. Soc. Ind. Appl. Math. 32–38.
Navet, N., Song, Y.-Q., Simonot, F., 2000. Worst-case deadline failure probability in real-time applications distributed over CAN (Controller Area Network). J. Syst. Archit. 46 (7), 607–618.
Pinho, L.M., Vasques, F., 2003. Reliable real-time communication in CAN networks. IEEE Trans. Comput. 52 (12), 1594–1607.
Qin, L., Yu, J.X., Chang, L., Tao, Y., 2009. Querying communities in relational databases. In: ICDE, pp. 724–735.
Raymond, J., Gardiner, E., Willett, P., 2002a. Rascal: calculation of graph similarity using maximum common edge subgraphs. Comput. J. 45 (6), 631–644.
Raymond, J., Willett, P., 2002b. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput. Aided Mol. Des. 16 (7), 521–533.
Suwatthikul, J., 2010. Fault Detection and Diagnosis for in-Vehicle Networks. INTECH Open Access Publisher.
Suwatthikul, J., McMurran, R., Jones, R.P., 2011. In-vehicle network level fault diagnostics using fuzzy inference systems. Appl. Soft Comput. 11 (4), 3709–3719.
Taylor, J.E., Amor-Segan, M., Dhadyalla, G., Jones, R.P., 2014. Discerning the operational state of a vehicle's distributed electronic systems from vehicle network traffic for use as a fault detection and diagnosis tool. Int. J. Automot. Technol. 15 (3), 441–449.
Terveen, L.G., McDonald, D.W., 2005. Social matching: a framework and research agenda. ACM Trans. Comput. Hum. Interact. 12 (3), 401–434.
Tindell, K., Burns, A., Wellings, A.J., 1995. Calculating controller area network (CAN) message response times. Control Eng. Practice 3 (8), 1163–1169.
Tran, T., Wang, H., Rudolph, S., Cimiano, P., 2009. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In: ICDE, pp. 405–416.
Vector Informatik GmbH, 2010. CAN Analyzer User Manual http://vector.com/vi_manuals_en.html.
Wang, G., Wang, B., Yang, X., Yu, G., 2012. Efficiently indexing large sparse graphs for similarity search. IEEE Trans. Knowl. Data Eng. 24 (3), 440–451.
Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z., 2013. A distributed graph engine for web scale RDF data. In: VLDB, 6, pp. 265–276.
Zeng, Z., Tung, A.K., Wang, J., Feng, J., Zhou, L., 2009. Comparing stars: on approximating graph edit distance. In: VLDB, 2, pp. 25–36.
Zhang, S., Yang, J., Jin, W., 2010. Sapper: subgraph indexing and approximate matching in large graphs. In: VLDB, 3, pp. 1185–1194.
Zhao, X., Xiao, C., Lin, X., Wang, W., 2012. Efficient graph similarity joins with edit distance constraints. In: ICDE, pp. 834–845.
Zheng, W., Zou, L., Lian, X., Wang, D., Zhao, D., 2015. Efficient graph similarity search over large graph databases. IEEE Trans. Knowl. Data Eng. 27 (4), 964–978.
Zhu, X., Song, S., Wang, J., Yu, P.S., Sun, J., 2014. Matching heterogeneous events with patterns. In: ICDE, pp. 376–387.
Zhu, Y., Qin, L., Yu, J.X., Cheng, H., 2012. Finding top-k similar graphs in graph databases. In: EDBT, pp. 456–467.

Sun Geol Baek received the B.S. degree in Computer Science and Engineering from Konkuk University in 2006. He is currently a M.S. student at Sungkyunkwan University, and a software engineer of Hyundai Motors Group. His research interests include database systems, graph databases, and query processing.

Dong Hyun Kang received the B.S. degree in Computer Engineering from Korea Polytechnic University, Korea, in 2007, and the M.S. degree in College of Information and Communication Engineering from Sungkyunkwan University, Korea, in 2010. He is currently a Ph.D. student at Sungkyunkwan University. His research interests include storage systems, operating systems, and embedded systems.

Sungkil Lee is an associate professor of Software Department at Sungkyunkwan University, Korea. He obtained his B.S. (2002) and Ph.D. (2009) degrees in Materials Science and Engineering and Computer Science and Engineering at POSTECH, respectively. He was a postdoctoral researcher at the Max-Planck-Institute (MPI) Informatik (2009–2011). His research interests include computer graphics, information visualization, haptics, and human computer interaction, with emphasis on perception-based rendering and GPU algorithms.

Young Ik Eom received his B.S., M.S., and Ph.D. degrees from the Department of Computer Science and Statistics of Seoul National University in Korea, in 1983, 1985, and 1991, respectively. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine from Sep. 2000 to Aug. 2001. Since 1993, he is a professor at Sungkyunkwan University in Korea. His research interests include system software, operating system, virtualization, and system securities.