Knowledge-Based Systems 250 (2022) 108988

Contents lists available at ScienceDirect

Knowledge-Based Systems

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

An agnostic and efficient approach to identifying features from execution traces

Chun-Tung Li [a,b], Jiannong Cao [a,b], Chao Ma [c,*], Jiaxing Shen [b,*], Ka Ho Wong [b]

[a] Hong Kong Polytechnic University Shenzhen Research Institute, China
[b] Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
[c] School of Cyber Science and Engineering, Wuhan University, China

ARTICLE INFO

Article history:
Received 27 December 2021
Received in revised form 21 March 2022
Accepted 4 May 2022
Available online 13 May 2022

Keywords:
Execution trace
Dynamic analysis
Program comprehension
Sequential pattern mining

ABSTRACT

Program comprehension is a necessary step during software understanding and maintenance. It is usually performed by analyzing data gathered from program execution. These execution traces reveal the relationship between high-level concepts (features) and low-level implementation details. However, identifying features from execution traces is difficult and time-consuming due to their large volume and complexity. Existing work assists the process by semi-automated tools, leveraging either human input or prior knowledge of the program implementation. It remains the key limitation towards a general method to build such mapping. In this paper, we proposed TRASE, an approach to identify features by segmenting the execution traces without the need for any human intervention. The segments are identified by mining recurring patterns on a sequence database, which is constructed by numerous execution traces gathered from normal use of a program. Each segment refers to a feature and the labels are inferred from the traces to assist program comprehension. The performance is evaluated on traces collected from android applications and a synthetic dataset. TRASE achieved up to 86% in F_1 score and the result indicates that it is robust to highly variate traces while efficient for large data.

1. Introduction

Program comprehension aims to make developers understand software programs by analyzing artifacts like the source code and execution traces. The understandings are important for developers since over 50% of maintenance effort is spent on program comprehension alone [1]. In many real-life scenarios like smart cities, cloud management systems [2], and Internet of Drones [3], there is an urgent need for automatic program comprehension for machines with minimized human intervention, to achieve interoperability among heterogeneous operating systems [4]. Understanding the program behavior with diverse hardware design and software implementation is a prerequisite for enabling communication and control among them [5].

To understand a program, it is necessary to know the provided features and its mapping with the implementation details [6]. Various tasks have been proposed by analyzing data, like execution traces, gathered from a running program (a.k.a dynamic analysis). It provides an accurate picture of a software system since it exposes the program's actual behavior, which has been widely used in tasks like feature location [7–9], trace abstraction [6,10], and feature identification [11]. In particular, we are interested in identifying features of a program by analyzing its execution trace for the importance to the aforementioned applications. With the rapid development of modern software, an overwhelming amount of execution traces are generated. To reduce the analysis complexity, a critical step is *execution trace segmentation, ETS* converts the execution trace into meaningful and manageable segments. Existing solutions either posing strong assumption of the program implementation [12–14] or require human intervention to derive proper segmentation [15–17], which greatly limits the potential of automatic program comprehension. Finding the near-optimal segments of a trace is NP-hard [7], since the noise, unpredictability and enormous amount as the key challenges for ETS. Therefore, we ask the question "is it possible to derive an efficient ETS from a large amount of traces automatically?".

TRASE, an agnostic and efficient proposed approach is the answer. A *feature* refers to a unique sequence of executions that exercise some functionality of a program. The intuition of TRASE is when exercising the same feature, similar sets of execution are invoked repeatedly, and the same to the resulting traces generated [10]. Segments corresponding to a feature can be attained by finding the most recurring patterns, while nearly no

* Corresponding authors.
E-mail addresses: chun-tung.li@connect.polyu.hk (C.-T. Li), jiannong.cao@polyu.edu.hk (J. Cao), chaoma@whu.edu.cn (C. Ma), jiaxshen@polyu.edu.hk (J. Shen), khowong@polyu.edu.hk (K.H. Wong).

https://doi.org/10.1016/j.knosys.2022.108988
0950-7051/

assumptions are made on the input traces. First, we construct a sequence database that adopted a compressed representation to abstract the program behavior from the low-level implementation details. Then, we introduced a variation of the closed sequential pattern mining problem, where it is interested only in non-overlapping patterns that maximize the coverage over the sequence database. This variation together with the representation realizes our efficient heuristic algorithm for maximum support sequential pattern (MSSP) mining. Finally, we select a set of non-overlapping MSSPs that maximize the coverage using a branch and bound algorithm.

To evaluate our proposed approach, we focus on a use case of android applications. Experimental evaluation was conducted on traces collected from real-world android applications. It demonstrates that our approach is robust and practical to real-world applications. The approach is also evaluated on a synthetic dataset, which allows us to manipulate different properties of the data such as the length and pattern appeared frequency.

TRASE requires barely any prior knowledge of the program implementation, therefore can potentially lead to a fully automatic program comprehension method. It is variation tolerant while efficient to compute on a conventional machine in a short time. As an important first step of dynamic analysis, it can support numerous downstream applications including feature location, trace abstraction, and feature identification. We summarize our main contribution as follow:

- We proposed a generic and efficient approach to infer the segment of large execution traces that barely any prior knowledge is required.
- We designed and implemented an efficient algorithm for maximum support sequential pattern mining.
- We conducted an extensive evaluation of synthetic datasets and traces collected from real-world android applications, to validate the proposed approach.

The rest of the paper is organized as follow. In Section 2, we provide the motivating example and the challenges of trace abstraction. Section 3.1 provides the necessary background and problem formulation. The detailed approach is presented in Section 4. Section 5 discussed the experimental evaluation. In Section 6, we provide a brief summary of the related work. Finally, the limitations and future directions are summarized in Section 7.

2. Motivation and challenges

Consider a smart home environment where the lights and television are controlled by Google Home[1] and Apple HomeKit[2] accordingly. For an application that wants to manage both devices, software engineers have to study both frameworks to develop the program that to operate the devices. The application now wants to include an air-conditioner controlled by Amazon Alexa,[3] engineers have to spend extra effort to maintain the program for this new device. The same process goes on for every new device. The lack of interoperability limited the use of the devices, and there is little intention for the company to support the rival's platform.

A similar scenario happens on the cloud service provider on a much larger scale, where a cloud platform aims to manage various applications developed by different parties on multiple operating systems [5]. It is nearly impossible to study all applications and develop a system that capable to manage the applications. Here comes the desire for automatic program comprehension, to let the machine understand the functionalities and the behavior of a program. The management system can then discover the functionalities of heterogeneous applications, and coordinate the resources available from different devices.

Without human intervention, generating execution traces that can facilitate program comprehension is difficult. One way is to collect execution traces from real-world user operations and perform analysis from those data. However, such traces can be noisy, unbounded, and unpredictable. Splitting the large execution traces into independent segments that correspond to a specific function greatly reduces the complexity of analyzing those traces. Various sophisticated analyses can then be performed on each segment such as classification, clustering, information retrieval, and visualization of the segments. However, inferring the segment of a trace is non-trivial, it is even harder to segment on multiple large execution traces.

Information Overload. Modern applications become more and more complex, and so do the execution traces. Exercising a single functionality could generate a thousand or even a million events. Analyzing such overwhelming information is challenging, which easily invalidate existing mining algorithms. Some possible solutions could be reducing and compressing the size of the traces, by eliminating low-level implementation detail while preserving necessary information for comprehension. Also, an efficient mining algorithm is needed to extract the recurring patterns from such large traces.

Dynamic Program Behavior. Although exercising the same functionality generate similar traces. However, there exist unpredictable variations due to different operation sequences, state of the program, system intervention, and so on. It is difficult to allow slight variation while robust enough to distinguish between different functionality. The representation of the traces should be able to abstract the implementation details. Also, the mining algorithm should be able to tolerate slight variation while still efficient enough to the extra computational overhead.

3. Background

3.1. Problem formulation

This section provides the necessary background and related concepts on execution trace segmentation. First, we introduce the essential definitions and the problem formulation. Then, we provide a brief description of sequential pattern mining that is related to our problem. Finally, we discuss the limitations of applying the existing approach to our problem.

We will provide the definition of the essential concepts to help introduce our problem. A *program* is an executable software composed of a set of *methods* $M = \{m_1, m_2, \ldots, m_q\}$ to provide a set of *features* $F = \{f_1, f_2, \ldots, f_p\}$. A *sequence* $s = (x_1, x_2, \ldots, x_l)$ is a series of methods invoked consecutively where $x_i \in M$. A *feature pattern* $s_f = (x_i, x_j, \ldots, x_k)$ is a sequence where $x_i \in M$ and $i, j, \ldots, k$ are arbitrary integers, such that it contains all the methods required for a feature f. That is, a feature pattern is a combination of methods in some particular order. A feature typically exercises more than one method, and a method could be utilized for different features.

The *distance* between two sequences s_a and s_b is measured in the Jaccard distance between the set of methods extracted from the two sequences denote as A and B accordingly, defined as $dist(s_a, s_b) = 1 - \frac{|A \cap B|}{|A \cup B|}$, $0 \leq dist(s_a, s_b) \leq 1$. A sequence s_a is a *subsequence* of s_b if there exist integers $i_1 < i_2 < \cdots < i_l$ such that $a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots, a_l = b_{i_l}$ denote as $s_a \sqsubseteq s_b$. Exercising the feature f produce an *execution pattern* s'_f, which is a variation of the feature pattern s_f with noise: $\exists i : (x_i \in s'_f$ and $x_i \notin s_f)$,

[1] https://assistant.google.com.
[2] https://www.apple.com/ios/home/.
[3] https://alexa.amazon.com.

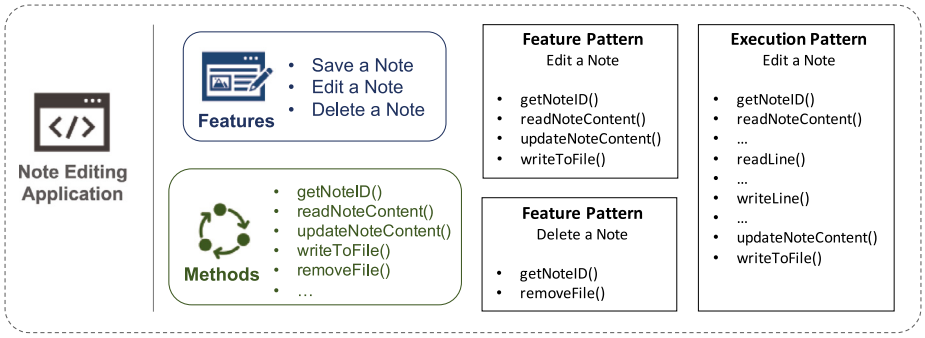

Fig. 1. An example of a note editing application to illustrate the concept of the related definitions including methods, features, feature patterns, and execution patterns.

and missing methods: $\exists i : (x_i \in s_f$ and $x_i \notin s'_f)$. The variability is controlled by some *threshold* θ, such that $dist(s_f, s'_f) < \theta$. It is based on the intuition where exercising the same feature will invoke a similar set of methods in similar order, but variation may occur due to run time dynamics.

A *trace* s_t is composed by a series of execution patterns $(s'_{f_{i_1}}, s'_{f_{i_2}}, \ldots, s'_{f_{i_k}})$, obtained by exercising one or more features as $g(s_t) = (f_{i_1}, f_{i_2}, \ldots, f_{i_k})$ for some arbitrary integers i_j. Take a note editing application as an example, in which the application provides three features, to save/edit/delete a note. There are a set of methods available to provide these features. The feature pattern composes of the essential methods, for example the delete note feature requires to obtain the identifier of a particular note and remove it from the file system. The execution of a feature produces a trace different from the ideal feature pattern due to the run time dynamics, but the essential methods are similar (see Fig. 1).

Given a *trace database* $\mathcal{D} = \{s_1, s_2, \ldots, s_n\}$ as a set of traces of a program, and a user-defined threshold θ. The frequency of a sequence s occur in $\mathcal{D}$ is denoted as *coverage*, and is defined as $cov_{\mathcal{D}}(s) = \sum_{s_t \in \mathcal{D}} s \sqsubseteq s_t$ and $\forall s, s_t : dist(s, s_t) \leq \theta$. Assume that $\mathcal{D}$ contains traces that exercise features in different order such that $\exists (i, j) : g(s_i) \neq g(s_j)$. Our problem is to find a minimum set of non-overlapping sequences $P = \{\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k\}$ that maximize the total coverage over $\mathcal{D}$ defined as $\max_{\hat{s}_i \in P} \sum cov_{\mathcal{D}(\hat{s}_i)}$, while $\forall \hat{s}_a \in P, \hat{s}_b \in P : \hat{s}_a \sqsubseteq s^t$ and $\hat{s}_b \sqsubseteq s^t \rightarrow idx(\hat{s}_a, s_t) \cap idx(\hat{s}_b, s_t) = \emptyset$, where $idx(s, s_t)$ is the set of indices for a sequence s in a trace s_t that is the locations of which s appears in s_t.

Generally, execution trace segmentation aims to divide the execution trace into non-overlapping segments, so that each segment belongs to the execution of exactly one feature. Since a feature produces a unique execution pattern of a program, exercising a feature multiple times produce similar patterns over the traces. Therefore, the ETS problem can be considered as finding the non-overlapping sequential patterns from the trace database, and therefore the existing methods have shed light on our problem.

3.2. Sequential pattern mining

Sequential pattern mining is a problem of mining interesting subsequences in a set of sequences, where interesting patterns are usually defined by their support — the frequency of the pattern within the sequence dataset. It is one of the most popular data mining tasks on sequences that have many applications ranging from bioinformatics to clickstream analysis. This problem is difficult since the search space grows rapidly even for small datasets. To tackle this problem, numerous algorithms have been proposed in the last two decades to improve the efficiency of mining the sequential patterns [18]. The intuition is to avoid exploring all possible subsequences but only those necessary to find the frequent sequential patterns. One of the directions is mining closed sequential pattern [19], which is the frequent sequential patterns that have no super-sequence with the same support.

Numerous mining algorithms have been proposed for mining sequential patterns in the past decades. Most of them rely on the support to prune the search space for better efficiency. However, sequences with mostly recurring patterns contain a great number of patterns that satisfy the minimum support, which makes the pruning ineffective. Also, existing approaches enforce specific constraints on the patterns to be efficient, like the sequence must appear the same in every occurrence. Although some previous work pushing down such constrain introduced a maximum gap between the events occurring in the sequence, the mining becomes intractable even for a small gap due to the tremendous search space. The noisy and highly recurring execution patterns invalidate the existing approaches.

4. Approach

In this section, we introduce our approach for feature identification by analyzing execution traces collected from the normal use of the software. The intuition of the proposed approach is that exercising a feature of software exhibits similar traces, which can be identified as non-overlapping recurring patterns. We proposed a framework to find the patterns from a set of traces, which is efficient and robust to minor variations of the execution details. Then we choose a set of non-overlapping patterns which is similar to finding the trace segmentation, such that each segment belongs to exactly one feature.

Fig. 2 provides an overview of our approach, where the system has three stages. The instrumentation stage captures the program behavior and collects the traces for analysis. The preprocessing stage converts the traces to a sequence database, which significantly reduces the complexity of the traces while preserving critical information for trace segmentation. The segmentation stage takes the sequential database as input and finds the recurring patterns as a unique feature and divides the traces accordingly. The details of each stage are discussed in the following.

4.1. Instrumentation of program behavior

In this work, we adopted YanCloud [5], an integrated cloud management system that supports multiple types of virtualization technologies. It provides an abstraction of resources within the cloud infrastructure. YanCloud is recently extended to support the emerging IoT technologies, in which the ability to adapt to heterogeneous devices is key to realize management among the

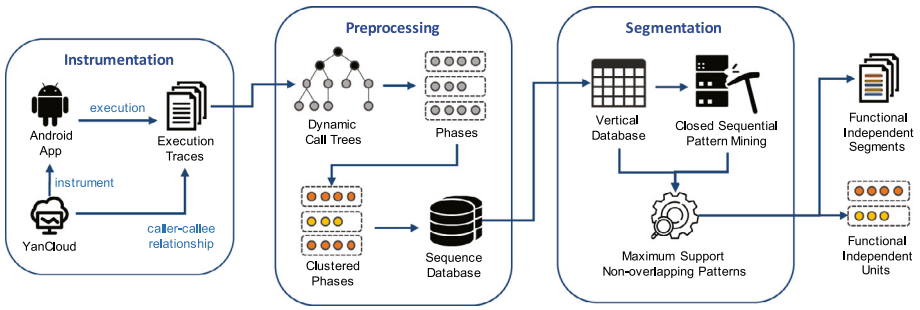

Fig. 2. An overview of the System Framework: (1) the system applies YanCloud to instrument an android application during execution to collect traces; (2) dynamic call trees are then converted from the traces and phases were identified from the tress to build a sequence database; (3) the database is processed in vertical form to perform closed sequential pattern mining efficiently; (4) eventually, segments are detected by finding the non-overlapping patterns.

cloud and IoT. Given the ubiquity of smartphones, it has become an integral part of IoT that coordinating the resources among different devices. Therefore, it is critical to understand smartphone applications automatically and the platform is recently developed to support this task.

The YanCloud operates on a modified Android firmware, collecting the program behavior on a smartphone and transmit the data collected to the server. Modern programs are typically organized as a collection of methods, and usually a series of methods are invoked to achieve one feature. This invoke behavior is a chain of caller methods that invoke a called method, while the called method may also invoke another method, or even a series of methods. For example, to display a note in a note editing app, this feature is initiated by a caller method - *onButtonTouch()*. It is a system method available in the android system, and triggered by the user who clicked on one of the note. Then it will invoke an app method called *readNoteContent()* implemented in the app. This method will further invoke a series of *readline()* methods to load the contents from a file to the memory. Finally, the *readNoteContent()* invoke the *displayNoteContnet()* method that further called a series of *printline()* methods to display the note content.

We leverage only this caller–callee relationship for execution trace segmentation, since there has little implementation assumption made that allowing a more generic approach. A filter was applied to collects traces only under the package of the target app. Most of the system methods have been ignored except the system methods that trigger the app methods. Because we are only interested in the behavior of the target app. It also significantly reduces the overwhelming information generated by system operations. For each application, we manually define a set of features and generate a set of test cases. Each test case contains a random number and combination of features, to simulate traces collected from users.

4.2. Preprocessing execution traces

With the execution traces collected from the previous step, the preprocessing stage is responsible for converting the data to a sequence database for mining purposes. It first builds the Dynamic Call Tree (DCT) for each of the traces. The phases are then identified from the DCTs. To handle a small variation of the phases, we perform clustering to group similar phases into the same class. Finally, the sequence database is built containing the list of clustered phases and passed to the next stage. The following provides detail of the preprocessing procedure.

The traces captured from YanCloud is a sequence of events containing the caller–callee relationship as a tuple, and the completion of each method invoked. We then build a DCT for each trace since it is the most precise data structure to represent the calling behavior [16]. To improve the space efficiency, we divide

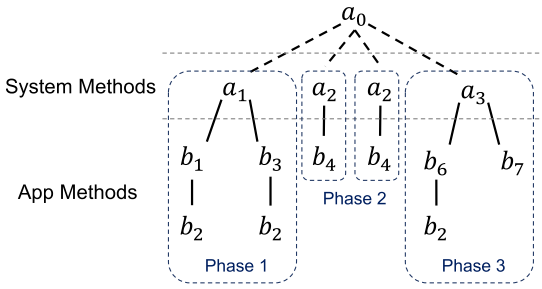

Fig. 3. An Example of a Trace Represented as a Dynamic Call Tree. The system methods defined the subtrees where each subtree is considered as a phase.

the DCT by the system events such that each DCT is composed of multiple sub-trees, denoted as phases [6]. It is reasonable as the methods invoked by the same system event must belong to the same feature. Each phase is treated as a set such that each method invoked within the same phrase appears once only, and only the first occurrence of each phase is recorded. It can abstract the implementation details such as loop and recursive calls.

Consider an example DCT in Fig. 3, in which the nodes represent the methods being invoked and the edges represent the caller–callee relationship. For example, method a_0 called methods a_1, a_2, and a_3, in which a_1 called b_1 and b_3, and so on. The methods a_i are the system methods (i.e. *onButtonTouch()*) and b_i are the methods implemented in the app (i.e. *readNoteContent()*). The root node a_0 is an abstract representation to all system implementation and its child (a_1, a_2, a_3) are the system events. In this example, the DCT contains 4 subtrees initiated by the system methods (a_1, a_2, a_2, a_3). Since (a_2, b_4) just repeated itself which will be counted once only. Therefore, we convert the DCT to 3 phases by choosing the unique methods for each subtree as a set represented as: $\{a_1, b_1, b_2, b_3\}$, $\{a_2, b_4\}$, and $\{a_3, b_3, b_6, b_7\}$.

With the sequence of phases, we then perform clustering to group similar phases into the same cluster, result in a set of clustered phases. The distance between the two phases is defined as $1 - Jaccard(A, B)$.

$$dist(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

Hierarchical Agglomerative Clustering [20] is applied to the phases such that the pairwise distances of all phases within the same cluster are less than some threshold θ. The original phases are replaced by the clustered phases. This allow similar phases to be treated as the same so that it better handle variation due to run time dynamics. In this work, θ is 0.1 if not specified. After the preprocessing procedure mentioned above, the example trace becomes a list of phases in which the phase is represented by the

Table 1
An example of sequence database in horizontal representation.

SID	Sequences
1	(A, B, C)
2	(A, B, C, D, E)
3	(D, E, A)
4	(B, C, D, E)

Table 2
The vertical representation of the sequence database in Table 1.

SID	A	SID	B	SID	C
1	1	1	2	1	3
2	1	2	2	2	3
3	3	3		3	
4		4	1	4	2

SID	D	SID	E
1		1	
2	4	2	5
3	1	3	2
4	3	4	4

phase ID. The sequence database is then built and pass to the next stage for segmentation.

4.3. Segmentation of trace sequences

The segmentation stage takes a sequential database as input and finds the set of non-overlapping closed sequential patterns with the maximum support. It first converts the sequence database to a vertical database, then finds a set of potential closed sequential patterns from the vertical database. Finally, it finds the set of patterns that maximize the sum of their supports by reducing the problem as a maximum weighted independent set problem and solving it with a branch-and-bound approach. In the following, we discuss the detail and rationale of each step in this stage.

4.3.1. Construct vertical database

A sequence database in *horizontal format* is a database where each entry is a sequence; a sequence database in *vertical format* is a database where each entry represents an item and indicates the list of sequences where the item appears and the position(s) where it appears [21]. Previous work suggested that mining sequential patterns from vertical database format is much more efficient than horizontal database format. With the sequence database obtained from the previous step, we build the IDList [22] as a vertical format of the database for efficient sequential pattern mining.

Consider a sequence database as shown in Table 1. Each entry is a sequence of phases obtained from the original trace, in which each phase is a set of methods. The corresponding vertical database is shown in Table 2. We also create an extension map, a data structure to store the coherence relationship in the sequence database. Given the maximum gap *max_gap*, the extension map of a phase A is all phases that exists in the same trace after A in which the position offset is at most *max_gap*: $\{X|(B_a, B_b) \in s_i \in S$ and $b - a \leq max\_gap\}$. For example, given the $max\_gap = 2$, the extension map of phase B in Table 1 is $\{C, D\}$.

Since execution traces are highly repetitive and therefore the number of candidates in the extension map is limited. It significantly reduces the search space when finding the closed sequential pattern, and both IDList and extension map can be constructed in one scan of the sequence database efficiently. Note that the original IDList contains the raw transaction sequences, while we consider the phase (set) as the element in this work. The reason is that we are not interested in finding sequential patterns at the method level, if the phases are very different, then they must belong to another feature. With the clustered phases, it allows slight variation on the phases, while significantly reduce the computation overhead searching for method level sequential patterns. The mining algorithm is then developed based on the vertical database.

4.3.2. Maximum support closed sequential pattern mining

Although execution traces are highly repetitive, finding those recurring patterns to segment the execution trace is non-trivial, and actually is NP-hard [7]. Inspired by sequential pattern mining, we proposed a heuristic approach to find the closed sequential patterns efficiently. Then, we reduce the problem to a maximum weighted independent set problem to find a set of non-overlapping patterns that maximize the total sum of support. The detail of the proposed method is provided in the following.

Before we introduce the solution, we clarify how existing sequential pattern mining approaches failed in our problem. The sequential pattern mining problem aims to discover all "interesting" patterns that frequently appear in the sequence database. It is assumed that a majority of candidate patterns are infrequent such that the search space can be drastically reduced. However, the execution traces are highly repetitive, and therefore a large number of pattern candidates are qualified as interested that invalidated the existing mining algorithms. We leverage this property to design the pruning strategy such that it is much more efficient than the existing method in our problem at hand.

Given the *min_sup*, the search space is constructed including all phases where their supports are greater than *min_sup* and are sorted by their supports in descending order. For each candidate phase in the search space, the pattern is explored by adding one phase at a time in a depth-first search manner. The pattern is extended recursively until the coverage of the closed sequential pattern decrease, in which the coverage is defined in Eq. (2). It is different from the original sequential pattern mining, which keeps exploring until the support is less than *min_sup*. We demonstrate in the following example why existing sequential pattern mining is not suitable for our problem.

Consider an example where exercising features (f_1, f_2, f_3) produce phase sequences as $(\{A\}, \{B, C\}, \{D, E\})$ accordingly. Traces generated by exercising features (f_1, f_2) and (f_1, f_2, f_3) is the first and second sequences as shown in Table 1. Finding patterns in these traces alone could not identify all the features separately, but treat $\{A, B, C\}$ as one feature. By increasing the sequence database, it covers different combinations of feature sequences that help separate each feature from the others. In this case, the coverage of $\{A, B, C\}$ is 6 while the coverage of $\{A\}$ and $\{B, C\}$ are 9. Therefore, instead of accepting $\{A, B, C\}$ as a maximum support sequential pattern, we choose the patterns with the largest coverage to identify traces that belong to the same feature.

The proposed heuristic has a delayed checking on the coverage only perform when a closed sequential pattern is found. Fig. 4 shows an example of the coverage while extending a candidate pattern. As we can see the coverage contains multiple peaks indicated by the blue plus sign. The peaks indicate where closed sequential patterns are found and check if the coverage $w(x)$ increases comparing to the last best coverage, and terminate the search of the candidate if the coverage decrease. The heuristic realizes the discovery of the maximum support sequential pattern by avoiding local optimum. On the other hand, we find that most applications have commonly used utility functions. Those patterns corresponding to the utility functions have relatively high support, which potentially dominates the coverage comparing to other patterns. It might separate a pattern into multiple segments as a result. To avoid this, we also introduced the minimum size

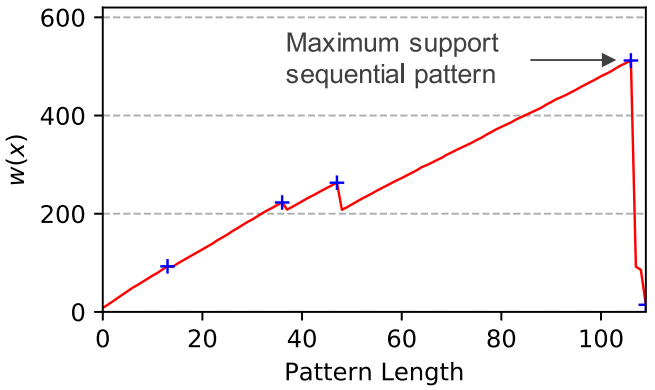

Fig. 4. An example of the coverage of a pattern at different lengths.

min_size of a phase specified by the user to eliminate those utility functions. This intuition can help significantly prune the search space while performing the search. On one hand, it reduces the search space vertically by terminate earlier from exploring the same candidate if the support of the pattern decreases. On the other hand, it reduces the search space horizontally by eliminating those phase candidates that are already included in another closed sequential pattern. Since the pattern obtained from such a candidate must be a subsequence of that closed sequential pattern.

After finding all the closed sequential patterns, it is possible that some patterns are overlapped which violate our constrain. To segment the traces using the patterns identified, we find the set of non-overlapping patterns that maximize the total coverage. As the problem is NP-hard, we reduce it to the maximum weighted independent set (MWIS) problem and using a graph-based approach [23] to solve the problem efficiently. In particular, we construct the graph $G = \{V, E\}$ for vertices $V = \{v_1, v_2, \ldots, v_{|V|}\}$. Each vertex v_i represent a closed sequential pattern s'_i and the weight of the vertex is defined as the *total number of methods in the pattern* * *support of the pattern*:

$$w(s') = \sum_{X \in s'} (|X| * sup_{\mathcal{D}(s')}) \tag{2}$$

The set of edges E represents if two closed sequential patterns are overlapped. More specifically, $(i, j) \in E$ if s'_i and s'_j are overlapped in at least one sequence $s \in \mathcal{D}$. The objective is then defined as:

$$\max_{V_{opt} \subseteq V'} \sum_{v_i \in V_{opt}} v_i$$
$$s.t. \forall u, v \in V_{opt} : (u, v) \notin E \tag{3}$$

V_{opt} is the selected set of independent vertices that maximize the sum of weights. The graph generated is usually sparse with numerous components, or disconnected subgraphs, since the patterns overlap only with similar patterns. To find the solution more efficiently, we divide the problem into multiple subproblems by the components in G and finally concatenate the result to obtain the optimal solution. The detailed algorithm is shown in algorithm 1 and 2.

The algorithm 1 is the overall method to find the maximum support sequential patterns. It takes the sequence database $\mathcal{D}$ as input and the minimum support *min_sup*, maximum gap *max_gap*, and minimum size of pattern *min_size* as parameters. It first initializes the set of closed sequential patterns as Z and the set of maximum support sequential patterns as S' in lines 1 and 2. Then the search space C is constructed that contains all the unique phases with support greater than or equals to *min_sup* in line 4. Note that the search space is in descending order of the support. From lines 5 to 13, it performs the depth-first search mining the closed sequential patterns for each of the candidates in C. The recursive method extend(Q, Z, qs) in line 7 is described in algorithm 2.

This recursion implemented the vertical pruning of the search space. The recursion takes query pattern Q as input and explores for each candidate in the extension map *xmap*. The largest coverage by far is stored as *qs*, and is updated when Q is a closed sequential pattern. The recursion goes on until the coverage of the closed sequential pattern is less than *qs*. Finally, if no pattern is found from all the candidates, it returns Q if it is a closed sequential pattern.

After the recursion, it back to algorithm 1, where the horizontal pruning of the search space is performed in lines 8 and 12. Specifically, the candidate in C is removed from the search space if it is already included in any found closed sequential pattern Z that has the same support as the candidate. Finally, from lines 14

Algorithm 1 Maximum Support Sequential Pattern Mining

Require: $\mathcal{D}$, double *min_sup*, int *max_gap*, int *min_size*
Ensure: S'
1: $Z \leftarrow []$;
2: $S' \leftarrow []$;
3: $(id\_list, xmap) \leftarrow$ build_verticalDB($\mathcal{D}$);
// Mining closed sequential patterns
4: $C \leftarrow$ unique_desc(id_list);
5: **for** $Q \leftarrow C$ **do**
6: $qs \leftarrow w(Q)$;
7: $Z \leftarrow$ extend(Q, Z, min_sup, qs);
8: **for** $z \in Z$ **do**
9: **if** $sup_D(Q) = sup_D(z)$ **then**
10: remove(C, z);
11: **end if**
12: **end for**
13: **end for**
// Find non-overlapping closed sequential patterns
14: $G \leftarrow$ generate_graph(Z);
15: **for** $G' \leftarrow sub\_graph(G)$ **do**
16: $S' \leftarrow$ MWIS(g);
17: **end for**
18: **return** S';

Algorithm 2 Recursive Extension Method: extend(Q, Z, qs)

Require: Q, int Z, double *min_sup*, int *qs*
Ensure: P
1: $P \leftarrow []$;
2: **if** $sup_D(Q) \geq min\_sup$ **then**
3: **if** is_closed(Q) **then**
4: **if** $w(Q) < qs$ **then**
5: **return** P;
6: **else**
7: $qs \leftarrow w(Q)$;
8: **end if**
9: **end if**
10: **for** $X \leftarrow xmap(Q)$ **do**
11: $P \leftarrow$ extend($Q + X, Z, qs$);
12: **end for**
13: **if** is_closed(Q) and $|P| = 0$ **then**
14: $P \leftarrow Q$;
15: **end if**
16: **end if**
17: **return** P;

to 16, the maximum weighted independent set is found to select the non-overlapping patterns with maximum support.

With the identified patterns, we then extract the name of the method and its class and separate the name by either delimiter or capital letter. For example, *edit_memo.init* is transformed as a set of keywords {*edit*, *memo*, *init*}. Each method call is then treated as a sentence, and the pattern is a document that contains multiple sentences. Then we apply Rapid Automatic Keyword Extraction [24] to extract the top 5 keywords from each of the patterns to label the feature of the pattern.

5. Evaluation

We implemented our approach in Python as a tool, called TRASE. We empirically evaluate our approach on trace sequences collected from five open-source android applications and a synthetic dataset. The reason for a synthetic dataset is that we have full knowledge of the groundtruth patterns as well as complete control of the characteristic of the data. Given the large traces of android applications, it is difficult for human experts to determine the exact segment of each trace as well as the patterns that correspond to the features. With the synthetic dataset, the patterns are known beforehand, and we can adjust the characteristics of data including the length of the pattern, the number of patterns repeated, and the number of traces, to examine the performance of our approach. We examined the performance in terms of efficiency, effectiveness, and robustness, and the evaluations are motivated by three research questions as follow:

- RQ1. How the proposed approach performs comparing to the existing mining algorithm in terms of efficiency?
- RQ2. Can the proposed approach precisely identify recurring patterns from a large sequence database?
- RQ3. Is it robust enough to identify features from traces collected from real-world applications?

5.1. Synthetic dataset

The synthetic dataset is generated to estimate the efficiency and effectiveness of mining the recurring patterns. There are two major components to generate a dataset. First, we construct a set of patterns to simulate the trace of a feature. Each pattern is a list of phrases, and each phase contains a set of methods. Second, we construct the set of trace sequences as the sequence database. Each sequence is a combination of different patterns, which is to simulate the real traces generated by multiple features. There are multiple parameters that can be adjusted to generate different sequence databases: (1) the number of methods in phase, (2) the number of phases in patterns, (3) the number of patterns in sequences, (4) the number of repetitions for each pattern. For each parameter, the values are not fixed but follow Gaussian distribution that varies with a standard deviation as 10% of its mean. In this study, we choose 20 and 2 as the mean and standard deviation for all the parameters as default values if not specified. Finally, we perform noise injection to the generated traces in the phase level randomly with at most 30% chance, and the noise phase is inserted from all the phases over all the feature patterns.

5.2. Evaluation metrics

To examine the quality of patterns recovered from the sequence database, we estimate the relevance of the identified patterns $\hat{s} \in P$ comparing to the groundtruth patterns $\{s_f | f \in F\}$ of the features. Inspired by the previous work [25], we treat the result as a multi-class classification problem. Each feature f is a class label, and the methods within the corresponding pattern s_f are the labeled examples. A good prediction implies that $\hat{s}$ contains more methods from the same f while excluding methods from the other features. The precision of $\hat{s}$ on any feature f is defined as $\frac{|s_f \cap \hat{s}|}{|\hat{s}|}$. We said $\hat{s}$ is predicting class f if it has the maximum precision over all the features, denoted as $g(\hat{s}) = f$. With that we classify each $\hat{s}$ into one of the class in F. Then, we combine the methods from all the $\hat{s}$ that predicting the same class as $\{\hat{s} | g(\hat{s}) = f\}$ to obtain a large method set as $Y' = \{m_i, m_{i+1}, \ldots, m_{i+|Y'|}\}$. Similarly, we obtain a large method set by combining all methods in s_f as $Y = \{m_j, m_{j+1}, \ldots, m_{j+|Y|}\}$. The quality of the prediction of each class can then be measured by the precision, recall, and F_1 score defined as follow:

$$prec = \frac{|Y \cap Y'|}{|Y'|} \tag{4}$$

$$recall = \frac{|Y \cap Y'|}{|Y|} \tag{5}$$

$$F_1 = 2 \times \frac{prec \times recall}{prec + recall} \tag{6}$$

The overall performance is finally calculated by the macro average of each of the measures. Note that a pattern s_f can be recovered by one or more $\hat{s}$. Ideally, each s_f should be recovered by exactly one $\hat{s}$. Therefore, we also measure the number of $\hat{s}$ used to reconstruct s_f. The evaluation is defined as the root-mean-squared-error (RMSE) for each class:

$$RMSE = \sqrt{\frac{\sum_{f \in F} (|\{\hat{s} | g(\hat{s}) = f\}| - 1)^2}{|F|}} \tag{7}$$

5.2.1. Effectiveness evaluation

Effectiveness evaluation measures the performance of our approach on finding the underlying feature patterns. The setting of the evaluation contains the following steps. First, we constructed the synthetic trace databases with different maximum noise factors ranging from 10% to 30%. For each noise setting, 5 databases were generated with different random seeds to avoid any bias by chance. Then, TRASE is applied on those datasets with $min\_sup = 0.5$ and $min\_size = 100$. The result is aggregated with macro-average as shown in Section 5.2.1. The different noise setting aims to simulate unpredictable execution patterns, and providing an estimation on the effect of noise on performance.

Max Noise:	Max-Gap	RMSE	Prec.	Recall	F_1
10%	1	0.72	40.00%	29.26%	33.41%
	2	0.39	81.00%	68.27%	73.66%
	3	0.21	92.00%	77.36%	83.74%
	4	0.19	94.00%	78.81%	85.44%
	5	0.17	95.00%	79.37%	86.18%
20%	1	0.93	14.00%	10.55%	11.83%
	2	0.64	50.00%	41.97%	45.30%
	3	0.51	68.00%	53.64%	59.57%
	4	0.38	79.00%	61.96%	69.08%
	5	0.37	83.00%	64.43%	72.13%
30%	1	0.95	8.00%	4.52%	5.75%
	2	0.72	39.00%	31.56%	34.60%
	3	0.60	55.00%	42.74%	47.72%
	4	0.52	66.00%	49.92%	56.44%
	5	0.46	72.00%	53.57%	61.00%

We observed that the performance in terms of recovering the feature pattern is fairly sensitive to the noise level of the input data, which increases the variation of the execution patterns. The noisy patterns have a higher chance to exceed the distance threshold or the maximum gap allowed. From the evaluation result, we affirm that allowing the flexibility of the target pattern by

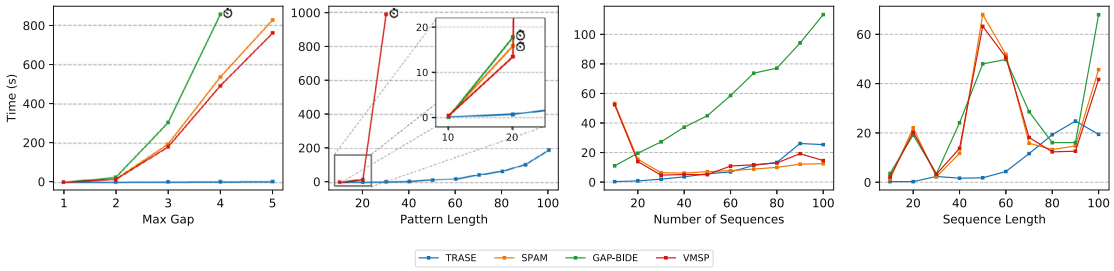

Fig. 5. Empirical execution time over different configuration of the dataset.

increasing the *max_gap* achieved much better performance than less flexibility. Simply allowing one gap can significantly improve the segmentation result up to 40% in F_1 score. Better performance can be achieved with higher *max_gap*, but not as significant as from none to just one gap allowed. The same for the RMSE, which implies that the gap also avoids breaking a pattern into many pieces of segments. This property is important in automatic program comprehension, as the boundary of a segment has to be accurately identified in order to support many downstream applications. However, the improvement comes with a price. The running time of the mining algorithm also increases with higher *max_gap*. Theoretically, the running time would be exponential to the *max_gap*. From our observation, let $max\_gap = 3$ achieve a reasonable balance between computational complexity and the pattern mining quality. Therefore, we will apply $max\_gap = 3$ for the rest of the evaluations.

5.2.2. Evaluation of efficiency

For efficiency, we empirically evaluate the running time with different sizes of input on the proposed approach as well as other baseline sequential pattern mining algorithms including Gap-Bide [26], SPAM [22], and VMSP [21]. Specifically, we examined the effect on the running time of the maximum gap, pattern length, sequence length, and the number of sequences. In this evaluation, the datasets were generated with a maximum noise factor of 10% for each setting of the input size. The parameters are $min\_sup = 0.5$, $max\_gap = 3$, and $min\_size = 100$ for all methods. We let the maximum computation time as 1000 s since allowing the algorithms to continue after that can easily trigger memory overflow that crashes the system.

Fig. 5 shows the effect of different sizes of the database on the empirical execution time. Starting from the left, the first plot shows the running time of different maximum gaps. The next shows the result under different pattern lengths in terms of the number of phases in each pattern. Then, the third is the result of different sequence lengths in terms of the number of patterns in each sequence. The right shows the result of a different number of sequences by varying the number of repetitions for each pattern.

As shown in the figure, TRASE significantly better than the existing methods in terms of running time in general. The existing methods are not scalable to the pattern length, and the running time increases drastically with the increasing maximum gap. Most existing methods cannot complete the computation for patterns with average pattern length equals to or larger than 30. Note that the average pattern length of the real-world data in this study is around 35. TRASE completed the computation in 0.59 s, while other methods were timeout on *Notes*. For the *Memo Notes*, there are similar results for SPAM, VMSP, and TRASE, recorded 0.44 s, 0.5 s, and 0.16 s accordingly. Except for GAP-BIDE, the running time was 363.13 s.

The performance of TRASE and existing methods are similar in the number of sequences. It is mostly because of the vertical representation that enables efficient mining for large sequence databases. TRASE is generally faster and less likely to stuck in the worst case for different sequence lengths. Since it only focuses on a small set of sequential patterns, that is more efficient and providing a more compressed set of patterns as a result. On the contrary, Gap-Bide and SPAM can easily identify more than 10,000 patterns for only a few small traces. Although it theoretically provides a better solution, the computation complexity to find the set of non-overlapping patterns will be intractable for such a large input size. Therefore, TRASE is shown to be more efficient and practical for execution trace segmentation.

5.2.3. Evaluation on feature identification

The proposed method is also evaluated for feature identification on traces collected from two android applications: *Notes - Memo*[4] and *Notes*.[5] The features were defined by domain experts, and the groundtruth is obtained by instrumenting the trace solely exercising the target feature. For each feature, three traces are collected and the methods are retained only if they appeared in both three traces. The traditional mining algorithms were causing the thrashing of the computer due to a large amount of memory consumed. It was unable to produce any result in most cases, so the evaluation only included the output of TRASE. It is expected that the identified patterns of TRASE are a proper subset of the patterns returned from those traditional mining algorithms. So the result should be similar though future studies are required to confirm this hypothesis.

Fig. 6 shows the feature identification result with different *max_gap* and *min_size* allowed. To be more adaptive to different applications, the *min_size* here is defined as a factor to the average phase size of the application. For example, $min\_size = 2$ means the minimum size is two times the average phase size. The RMSE is the measure of an error on the number of segments being identified for each feature. With the increasing maximum gap allowed, it is less likely to divide a pattern with random noise. Therefore, the RMSE decreases with increasing the maximum gap. As we can see from the figures, $max\_gap = 3$ can achieve comparable results to higher gaps allowed. The minimum size of a pattern poses constraints to the identified pattern that favors longer patterns and rejecting more short patterns that are possibly related to a feature. Therefore, the RMSE is lower with higher *min_size* but the F_1 score is lower. From the above result, we suggest that $max - gap = 3$ and $min - size = 2$ can produce better results in general.

The keywords are then extracted from each of the segments identified. Participants are recruited to recognize the features given the keywords. The result suggests that although the participants cannot correctly recognize all the features, it is mainly because of the naming convention of the program. For instance, the methods of adding a memo have about 90% in common with the methods of editing a memo. One can hardly identify between

[4] https://play.google.com/store/apps/details?id=com.abhi.newmemo.
[5] https://play.google.com/store/apps/details?id=com.ogden.memo.

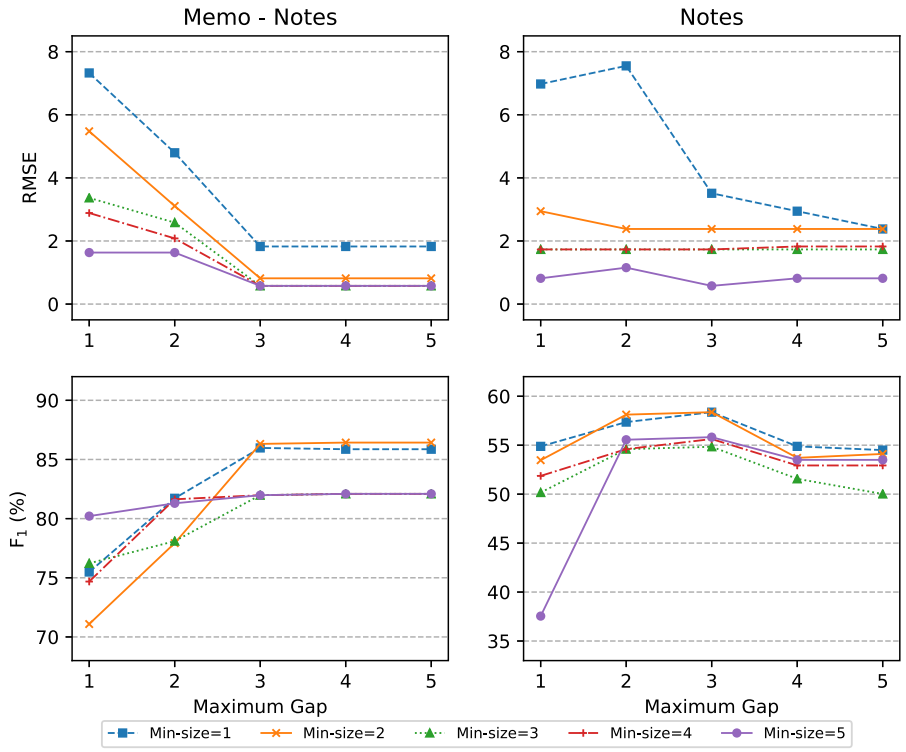

Fig. 6. The feature identification result on real traces.

adding and editing a memo by looking into the raw execution traces. In fact, participants who spent less than 5 min studying the keywords, can get a similar understanding of the trace segment that needs to be analyzed for more than an hour.

Finally, we recruited 14 participants who have at least 1 year to more than 5 years experience of programming for the comprehension test. We provided the set of features exercised and the keywords extracted from each of the segments, the participants are then trying to recognize the feature that is being exercised for each of the segments. The features are (a) add memo, (b) edit memo, and (c) delete a memo. With the proposed method, 8 segments were identified from the traces for the first memo app and 5 for the second. The participants achieved 66.2% and 45.2% accuracy accordingly for the first and second memo application. The performance is significantly better than random guess, but relatively lower than the performance of segmentation in the synthetic dataset. The key reason is that the quality of the method name highly depends on the developer practice, and some may name the method that is entirely different from the actual feature.

6. Related work

Execution trace segmentation is crucial to many program comprehension tasks, in which feature location and execution trace abstraction are the commonly studied area. Many approaches have been proposed in the last decade and this section will discuss some of the previous work relevant to this work.

Feature location is a problem that aims to identify source code that implements a specific functional requirement. The early work that leverages execution trace for feature location takes two test cases as input, in which one invoked the target feature and the other excluded [15,27]. Collecting such data input is however labor-intensive, so Eisenberg and Volder [28] proposed to leverage one large test suite for all features instead of two test cases for each feature. It still requires the developer to manually create the mapping of features and the test cases to build the input test suite which is time-consuming. In [7], the authors proposed to segment the execution trace to conceptually cohesive segments using a genetic algorithm. Medini [8] proposed a dynamic programming approach to further improve the speed of the computational time. However, the previous approaches pose strong assumption to the input traces, which require extensive manual input that remains a key limitation to an agnostic method.

On the other hand, trace compression and abstraction techniques were studied to improve the comprehensibility of execution traces, while some require few human intervention. Trace compression and abstraction is typically performed by finding recurring patterns within executions, then group the similar phases to retain high-level information from the execution details [6,10,29]. Feng et al. [6] and Alimadadi [10] both proposed a hierarchical abstraction of trace. The former one aggregates executions into clusters perform frequent pattern mining to identify the recurring patterns. The latter proposed SABALAN, a tool to infer models of motifs, which is the abstract and flexible recurring patterns in program execution. However, these approaches only group and extract the traces in order to assist user to perform program comprehension manually. The mapping between the feature and the implementation details still require extensive human input.

The idea of using sequential pattern mining to segment execution trace is not new [17]. However, as the system becomes more and more complex in recent years, previous approaches become intractable even for small applications. Xin et al. [11] proposed a heuristic to segment execution trace of android apps by the user events given that each feature is usually separated by the user interaction. This method leverage a machine learning model trained on manual labeled data to decide if two segments should merge. Obtaining such training data is time-consuming and the validity over different application is uncertain. To fill the research gap mentioned above, this work aims to provide a agnostic and efficient method to identify features from large execution traces, in which barely no assumption is made to the

traces that no human intervention is need to gather the input data. The approach also provide a mapping between the features and the implementation details that greatly reduce the need of human input during program comprehension.

7. Conclusion

In this work, we studied the problem of feature identification using data collected from program execution. We proposed an agnostic and efficient approach to identify features by segmenting the execution traces. Specifically, we introduced a heuristic mining algorithm to find a set of sequential patterns that maximize the span over a trace database. Our approach has barely any assumption to the program implementation as well as the input execution traces, so there is limited human intervention required. The proposed algorithm has greatly reduced the computational overhead while introducing flexibility to the target pattern. The approach is evaluated empirically with traces collected from real-world android applications and a synthetic dataset. The result shows that our approach is more efficient than the state-of-the-art sequential pattern mining algorithm on highly recurring sequences like execution traces. Also, the target feature patterns can be recovered accurately, achieving the best F_1 score at 86% on both real and synthetic datasets.

However, the proposed approach also comes with several limitations. First of all, it is assumed there contains a different order of feature execution within the trace database such that the feature patterns can be recovered from the traces. It is reasonable as the features have a great chance that being executed in a different order during normal use. Also, the proposed algorithm still scales exponentially to the pattern length and the number of sequences in the worst case. It actually is a trade-off between finding the optimal solution and more efficient pruning. The proposed method is robust to the noise of input traces since it explores most of the possible search space. One may introduce a more greedy heuristic but it may easily trap in the local optimal, which results in breaking one feature pattern into large numbers of segments.

Feature identification without human intervention is the first step towards automatic program comprehension for machines. There is still room for improvement, and problems to explore. In the future, a more efficient pruning approach will be studied that ideally scales linearly to the pattern length and the number of sequences. Non-convex optimization techniques could be applied to further improve the performance of searching the optimal segment of the traces [3,30–32]. Also, different downstream applications will be applied to the segments identified, such as privacy leakage detection, malware detection, and task scheduling in the cloud and fog computing environment [33–35]. Finally, the approach could be applied to larger datasets containing applications from different platforms.

CRediT authorship contribution statement

Chun-Tung Li: Conceptualization, Methodology, Investigation, Writing – original draft. **Jiannong Cao:** Supervision, Writing – review & editing, Project administration, Funding acquisition, Resources. **Chao Ma:** Supervision, Validation, Software, Writing – review & editing. **Jiaxing Shen:** Conceptualization, Supervision, Writing – review & editing. **Ka Ho Wong:** Writing – review & editing, Data acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by Key-Area Research and Development Program of Guangdong Province (No. 2020B010164002).

References

[1] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, R. Koschke, A systematic survey of program comprehension through dynamic analysis, IEEE Trans. Softw. Eng. 35 (5) (2009) 684–702.
[2] M. Abd Elaziz, L. Abualigah, I. Attiya, Advanced optimization technique for scheduling IoT tasks in cloud-fog computing environments, Future Gener. Comput. Syst. 124 (2021) 142–154.
[3] L. Abualigah, A. Diabat, P. Sumari, A.H. Gandomi, Applications, deployments, and integration of internet of drones (iod): a review, IEEE Sens. J. (2021).
[4] H. Mei, Y. Guo, Toward ubiquitous operating systems: A software-defined perspective, Computer 51 (1) (2018) 50–56.
[5] X. Chen, Y. Zhang, X. Zhang, Y. Wu, G. Huang, H. Mei, Towards runtime model based integrated management of cloud resources, in: Proceedings of the 5th Asia-Pacific Symposium on Internetware, 2013, pp. 1–10.
[6] Y. Feng, K. Dreef, J.A. Jones, A. van Deursen, Hierarchical abstraction of execution traces for program comprehension, in: Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 86–96.
[7] F. Asadi, M. Di Penta, G. Antoniol, Y.-G. Guéhéneuc, A heuristic-based approach to identify concepts in execution traces, in: 2010 14th European Conference on Software Maintenance and Reengineering, IEEE, 2010, pp. 31–40.
[8] S. Medini, P. Galinier, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, A fast algorithm to locate concepts in execution traces, in: International Symposium on Search Based Software Engineering, Springer, 2011, pp. 252–266.
[9] S. Medini, V. Arnaoudova, M. Di Penta, G. Antoniol, Y.-G. Guéhéneuc, P. Tonella, SCAN: an approach to label and relate execution trace segments, J. Softw.: Evol. Process 26 (11) (2014) 962–995.
[10] S. Alimadadi, A. Mesbah, K. Pattabiraman, Inferring hierarchical motifs from execution traces, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 776–787.
[11] Q. Xin, F. Behrang, M. Fazzini, A. Orso, Identifying features of Android apps from execution traces, in: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2019, pp. 35–39.
[12] A. Zaidman, S. Demeyer, Managing trace data volume through a heuristical clustering process based on event execution frequency, in: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings, IEEE, 2004, pp. 329–338.
[13] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, Feature location via information retrieval based filtering of a single scenario execution trace, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, 2007, pp. 234–243.
[14] Y. Watanabe, T. Ishio, K. Inoue, Feature-level phase detection for execution trace using object cache, in: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), 2008, pp. 8–14.
[15] N. Wilde, M.C. Scully, Software reconnaissance: Mapping program features to code, J. Softw. Maint.: Res. Pract. 7 (1) (1995) 49–62.
[16] G. Ammons, T. Ball, J.R. Larus, Exploiting hardware performance counters with flow and context sensitive profiling, ACM Sigplan Not. 32 (5) (1997) 85–96.
[17] H. Safyallah, K. Sartipi, Dynamic analysis of software systems using execution pattern mining, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), IEEE, 2006, pp. 84–88.
[18] P. Fournier-Viger, J.C.-W. Lin, R.U. Kiran, Y.S. Koh, R. Thomas, A survey of sequential pattern mining, Data Sci. Pattern Recognit. 1 (1) (2017) 54–77.
[19] X. Yan, J. Han, R. Afshar, CloSpan: Mining: Closed sequential patterns in large datasets, in: Proceedings of the 2003 SIAM International Conference on Data Mining, SIAM, 2003, pp. 166–177.
[20] A.K. Jain, R.C. Dubes, Algorithms for Clustering Data, Prentice-Hall, Inc., 1988.
[21] P. Fournier-Viger, C.-W. Wu, A. Gomariz, V.S. Tseng, VMSP: Efficient vertical mining of maximal sequential patterns, in: Canadian Conference on Artificial Intelligence, Springer, 2014, pp. 83–94.
[22] J. Ayres, J. Flannick, J. Gehrke, T. Yiu, Sequential pattern mining using a bitmap representation, in: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2002, pp. 429–435.
[23] P.M. Pardalos, N. Desai, An algorithm for finding a maximum weighted independent set in an arbitrary graph, Int. J. Comput. Math. 38 (3–4) (1991) 163–175.

[24] S. Rose, D. Engel, N. Cramer, W. Cowley, Automatic keyword extraction from individual documents, Text Min.: Appl. Theory 1 (2010) 1–20.
[25] A. Gensler, B. Sick, Novel criteria to measure performance of time series segmentation techniques, in: LWA, Citeseer, 2014, pp. 193–204.
[26] C. Li, Q. Yang, J. Wang, M. Li, Efficient mining of gap-constrained subsequences and its various applications, ACM Trans. Knowl. Discovery Data (TKDD) 6 (1) (2012) 1–39.
[27] W.E. Wong, S.S. Gokhale, J.R. Horgan, K.S. Trivedi, Locating program features using execution slices, in: Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122), IEEE, 1999, pp. 194–203.
[28] A.D. Eisenberg, K. De Volder, Dynamic feature traces: Finding features in unfamiliar code, in: 21st IEEE International Conference on Software Maintenance (ICSM'05), IEEE, 2005, pp. 337–346.
[29] O. Benomar, H. Sahraoui, P. Poulin, Detecting program execution phases using heuristic search, in: International Symposium on Search Based Software Engineering, Springer, 2014, pp. 16–30.
[30] L. Abualigah, D. Yousri, M. Abd Elaziz, A.A. Ewees, M.A. Al-Qaness, A.H. Gandomi, Aquila optimizer: a novel meta-heuristic optimization algorithm, Comput. Ind. Eng. 157 (2021) 107250.
[31] L. Abualigah, M. Abd Elaziz, P. Sumari, Z.W. Geem, A.H. Gandomi, Reptile Search Algorithm (RSA): A nature-inspired meta-heuristic optimizer, Expert Syst. Appl. 191 (2022) 116158.
[32] L. Abualigah, A. Diabat, S. Mirjalili, M. Abd Elaziz, A.H. Gandomi, The arithmetic optimization algorithm, Comput. Methods Appl. Mech. Engrg. 376 (2021) 113609.
[33] L. Abualigah, A. Diabat, M.A. Elaziz, Intelligent workflow scheduling for big data applications in IoT cloud computing environments, Cluster Comput. 24 (4) (2021) 2957–2976.
[34] L. Abualigah, A. Diabat, A novel hybrid antlion optimization algorithm for multi-objective task scheduling problems in cloud computing environments, Cluster Comput. 24 (1) (2021) 205–223.
[35] L. Abualigah, M. Alkhrabsheh, Amended hybrid multi-verse optimizer with genetic algorithm for solving task scheduling problem in cloud computing, J. Supercomput. 78 (1) (2022) 740–765.