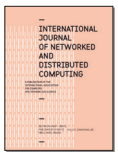
International Journal of Networked and Distributed Computing
Vol. 7(4); September (2019), *pp.* 149–157
DOI: https://doi.org/10.2991/ijndc.k.190911.002; ISSN 2211-7938; eISSN 2211-7946
https://www.atlantis-press.com/journals/ijndc



Research Article

Efficient Searching for Essential API Member Sets based on Inclusion Relation Extraction

Yushi Kondoh*, Masashi Nishimoto, Keiji Nishiyama, Hideyuki Kawabata, Tetsuo Hironaka

Graduate School of Information Sciences, Hiroshima City University, Hiroshima, Japan

ARTICLE INFO

Article History

Received 23 March 2019
Accepted 20 May 2019

Keywords

API member set
frequent pattern mining
application development
open source repositories
Android

ABSTRACT

Search tools for *Application Programming Interface* (*API*) *usage patterns* extracted from open source repositories could provide useful information for application developers. Unlike ordinary document retrieval, API member sets obtained by mining are often similar to each other and are mixtures of several unimportant and/or irrelevant elements. Thus, an API member set search tool needs to have the ability to extract an essential part of each API member set and to be equipped with an efficient searching interface. We propose a method to improve the searchability of API member sets by utilizing inclusion graphs among API member sets that are automatically extracted from source code. The proposed method incorporates the *frequent pattern mining* to obtain inclusion graphs and offers the user a way to search appropriate API member sets smoothly and intuitively by using a GUI. In this paper, we describe the details of our method and the design and implementation of the prototype and discuss the usability of the proposed tool.

1. INTRODUCTION

Application software for mobile devices is essentially developed as a combination of Application Programming Interfaces (APIs) of libraries or frameworks [1]. In fact, on average, 30–50% of the entire source code of Android applications is said to be occupied with method calls related to Android APIs [1]. Application developers are definitely required to have enough knowledge on how to properly choose and use appropriate API methods according to the functionalities to be realized. However, detailed information about the API methods is often not well documented. In addition, library and framework versions are frequently updated [2] and the specifications of API methods may change a lot from version to version. As such, it is not easy to obtain suitable information on the proper usage of combining multiple API methods to implement intended functionalities.

To help software engineers develop API-based applications efficiently, making use of *API usage patterns* obtained by mining open source repositories are supposed to be promising. There could be several styles of API usage patterns, such as those in the forms of API method call sequences or API member sets. There are a number of studies on the development of support tools that utilize API usage patterns [1–11]. Some help the user search examples of API usages [8] and others recommend appropriate API methods [9] or combinations of API methods [2,7] to support smooth development of software.

Search-based support tools for application development try to gather related information to queries given explicitly or implicitly by the user and select as appropriate information as possible to respond to the user. However, picking out appropriate candidates from the huge amount of similar ones is an inherently difficult task. In addition, given queries for search would be quite vague in most cases. Existing tools appear to either impose too much work on the user side or only support the user at a rather later stage of developing where there would not be many choices left.

In this paper, we present a method to let the user efficiently search a large number of API member sets extracted from open source repositories. What we offer in this paper is not an ordinary lexicographical search regarding API member sets as text documents, but a functionality-oriented search focusing on element-wise differences among API member sets. In the method, the *frequent pattern mining* [12] is applied to a large number of API member sets to extract inclusion relationships among API member sets. The extracted information can be expressed using inclusion graphs, where each node is weighted according to statistical information. The graph can be effectively used for searching and/or recommending suitable API method sets.

To evaluate the method, we designed a tool with a GUI for searching API member sets. By using the tool, the user can obtain an API member set that is considered useful for implementing the target functionality, by simply tracing the presented graphs. In this paper, we show the design and implementation of the prototype of the API member set search system and discuss the usefulness of it.

The rest of the paper is organized as follows. In Section 2, we introduce our approach by showing a motivating example. In Section 3, we describe the idea of utilizing inclusion graphs for searching API member sets. In Section 4, we describe the design of an API member set search system. In Section 5, we discuss the effectiveness

*Corresponding author. Email: kondoh@ca.info.hiroshima-cu.ac.jp

of the tool based on the results of some case studies. Section 6 shows a summary of related work and is followed by concluding remarks in Section 7.

2. MOTIVATION AND AIMS

2.1. API Member Set Search and its Effects

It is common that many API method calls are combined to implement a particular functionality in an application. Figure 1 shows a code segment of a typical Android application program. There are several methods and a named constant that belong to Android APIs, used in the program. Among them, `getDefaultSensor`, `registerListener`, `unregisterListener`, and `TYPE_PROXIMITY` are cooperating to implement a sensor-related functionality. On the other hand, `DataToFileWriter`, `writeToFile`, and `closeFiles` are used for accessing files. You can see from the figure a couple of facts that are common for Android applications: (1) multiple method calls from separate method definitions are cooperating for implementing a particular functionality, and (2) although groups of API members are used for implementing relatively independent functionalities, call/used sites of them are mixed and scattered all over the source code. These properties that are typical of event-driven programs make it a burden to separate a code segment that implements a single functionality from Android programs, leading the difficulty of building and maintaining Android applications.

If developers can easily obtain in advance the sets of API members that should be used to implement a specific feature, the user can avoid spending much time to decide which API methods to use and can write code focusing only on how to combine API methods in the set. Searching for the API methods that satisfies several conditions such as cooperability with other API methods may be much more laborious than just arranging elements in a fixed set of API methods. Tools that can recommend suitable *sets of API members* seem to be of great help for application developers.

```
public class ProximityDataRunnable
              extends Thread implements SensorEventListener {
  private SensorManager mSM;
  private Sensor ms;
  private DataToFileWriter mDTFW;
  ...
  public ProximityDataRunnable (SensorManager sM) {
    mSM = sM;
    mDTFW = new DataToFileWriter("Proximity.java");
    mS = mSM.getDefaultSensor(Sensor.TYPE_PROXIMITY);
    mDTFW.writeToFile("Time, Distance", false);
  }
  @Override
  public void run() {
    ...
    mSM.registerListener(this, ms, 2000000);
    ...
  }
  ...
  public stopDumping (SensorManager ...) {
    mSM.unregisterListener(this);
    mDTFW.closeFile();
  }
  ...
}
```

Figure 1 | Common pattern of Android programs composed of API method calls.

2.2. Realizing API Member Set Search

Application Programming Interface member set search systems should be based on the mining of open source repositories because the result should be up-to-date and full of variety. However, a search system based on a naive ranking capability would make the user stray in the sea of similar API member sets on the recommended list that is output as the result of searching.

Figure 2 shows an example of the result of an API member set search obtained by using the CAPIS system [7] (http://capis.ca.info.hiroshima-cu.ac.jp:8090/). CAPIS treats API member sets as a kind of text documents and ranks each one by using the TF-IDF (term frequency-inverse document frequency) measure. This approach would be effective for the ordinary document search where each document is sufficiently large and contains many kinds of words so that each one is rather distinguishable form others. The results in Figure 2 shows numbers of similar API member sets that are related to the given keywords of "text" and "view". It would be difficult for the user to scan the lengthy list to select the one that should correspond to the functionality the user had desired to implement.

As can be seen from the example in Figure 2, the search results of API method sets in a one-dimensional list format is not suitable for the user to effectively utilize the search results. One reason for this is that the search target is not a single API method to invoke but a set of API methods to combine. That is, a query expressed by a small number of search terms cannot be specific enough to pinpoint a suitable *set of API calls* in the large database obtained from open source repositories.

In order to alleviate the situation and realize an effective API member set search, we focus on the relevance of each API member in a set to restructure API member sets to construct a kind of search tree. We observe that there are a couple of cases: (1) a few members

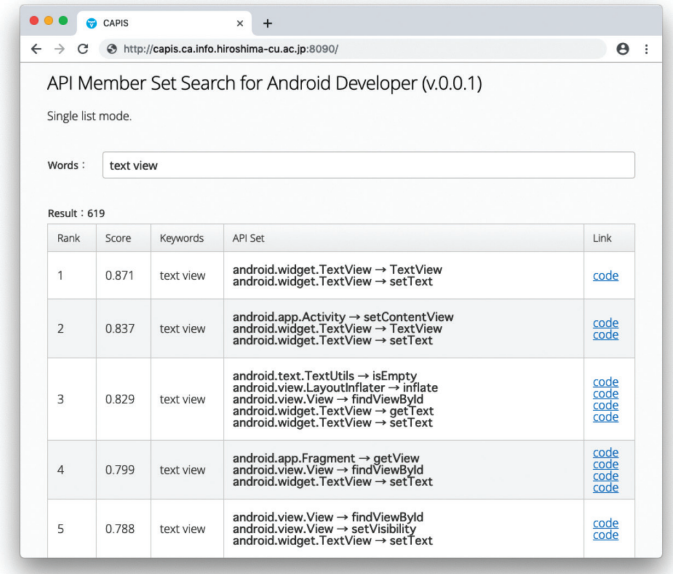


Figure 2 | Screenshot of CAPIS, an API member set search tool.

in an API member sets play important roles and the others are not, and (2) an API member set consists of several loosely related groups of closely related members. To refine and reorganize API member sets, we carry out the frequent pattern mining to extract inclusion relations mediated by newly produced subsets. The inclusion graphs constructed from the inclusion relations among API member sets are directly usable for API member set search; the user (or an algorithm) can trace the edges from a suitably small start point to an appropriate one, checking if the addition of a few API members is reasonable, step-by-step.

2.3. Designing API Member Set Search System

In this paper, we propose an API member set search system. The system consists of two parts; one is for collecting API usage information from open source repositories, extracting inclusion relations among API member sets by using the frequent pattern mining, and constructing inclusion graphs as a database for the API member set search, and the other is a GUI for the search system. We describe each part in Sections 3 and 4, respectively, in detail.

3. EXTRACTING INCLUSION RELATIONS BETWEEN API MEMBER SETS BY FREQUENT PATTERN MINING

In order to facilitate an efficient search of the API member sets, we clarify the relationships among sets in the large set of API methods that are obtained from open source repositories. By expressing the hierarchy extracted from the relationships explicitly, we could construct a database that would be usable as a basis that is easily usable for the developer or a recommendation system.

3.1. Various Roles of API Methods

Each API method in a set of API methods that are required to implement a specific functionality might be roughly classified into two groups based on the roles for each set, i.e., those that play a central role and inevitable to implement the functionality and those that are used to just extend and/or adjust the behavior of the functionality. The former API methods are considered to appear in common in API member sets for a specific functionality, while the others are not. The more important a method is for a functionality, the more often it seems to appear in API member sets related to the functionality. From the observation, we can expect that the API member sets that are related to the same functionality can be organized to extract their hierarchical relationships by analyzing inclusion relations. The hierarchical relationships of sets can be represented by using a graph structure.

3.2. Extraction of Hierarchical Relationships by using Frequent Pattern Mining

The frequent pattern mining [12] is considered to be effective in constructing the hierarchical graph structures. The frequent pattern mining is a method to extract frequently appearing subsets from a set of input item sets. We can obtain the hierarchical relations by analyzing the inclusion relations among the sets.

Applying the frequent pattern mining to the sets of API method sets might reveal the fact that, e.g., a particular couple of methods are always used simultaneously. In order to obtain those kinds of facts in the inclusion graphs, we use FP-Close algorithm [13] while carrying out the frequent pattern mining so that such kind of important pairs are not treated separately. For example, API methods `registerListener` and `unregisterListener` are a representative pair of the kind. Closed itemsets generated by the FP-Close algorithm should contain both of the two or none of them if the two methods are always used together.

3.3. Usability of Inclusion Graphs among API Method Sets

From the subsets obtained by applying the frequent pattern mining to the original API member sets and analyzing the inclusion relationships among sets, we can obtain, for example, the graph structure depicted in Figure 3 from the five API member sets shown in Figure 2. Rectangles and ovals in Figure 3 represent API member

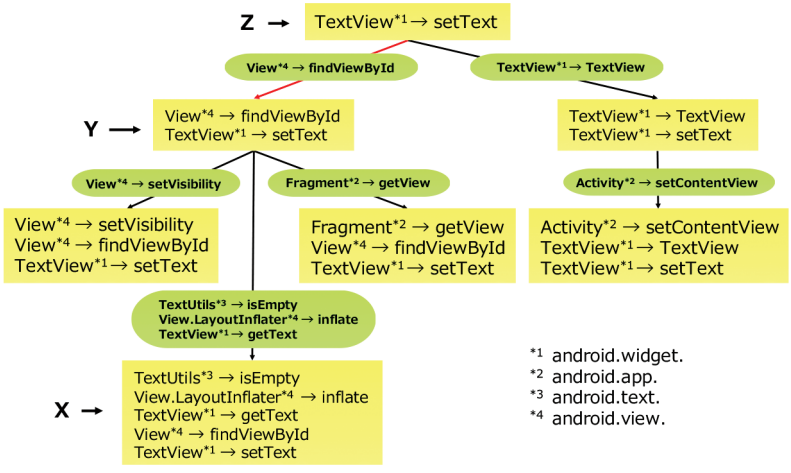


Figure 3 | Inclusion relations among API member sets represented as a graph.

sets and the differences between adjacent sets, respectively. Directed edges between rectangles through ovals are drawn such that when API member set A is a subset of B, there is always a path from A to B. The seven API member sets in Figure 3 are closed itemsets obtained from five sets in Figure 1. Among them, two are newly created API method set. Although what is shown in Figure 3 is a single tree, the result of the application of the frequent pattern mining to API member sets is, in general, a set of graphs.

Inclusion graphs are apparently useful for guiding searching for an appropriate API member set. For example, presenting a graph of Figure 3 to the developer would help him/her search for the right API member set by first selecting important API methods and then modifying the set by checking if some API methods should be added or deleted for the functionality to be implemented. Compared with the task of selecting the target set from the API member sets expressed in a one-dimensional form such as a list, it is considered that the burden on the developer is greatly reduced. However, there might be a problem since the graph of Figure 3 could become quite huge. In order to construct an API method set search system that uses the inclusion graphs directly, a specially arranged GUI would be inevitable. The graph representation of the inclusion relations can also be utilized by recommendation systems of API member sets by algorithmizing the procedure of tracing the edges of an inclusion graph.

4. API MEMBER SET SEARCH SYSTEM BASED ON INCLUSION GRAPHS

4.1. Overview of the Tool

We propose an API Member Set Search System with an interactive interface for developers to search API member sets efficiently. By using the system, the user can search for an API member set consisting of API methods that are inevitable to implement the functionality in his/her mind. In order to support the task of selecting one from a large number of API member sets collected from open source repositories, the system extracts the inclusion relationships among API member sets to obtain inclusion graphs such as the one shown in Figure 3. By tracing this graph, the user can search for an appropriate set of API methods as if collecting a set of API methods adding one-by-one in order of appearance frequently. However, the generated graphs could become huge in general. The proposed system offers a way to trace the tree structure easily and effectively by using the mouse.

The GUI of the system is as shown in Figure 4. The left pane of the window is used for listing tag clouds where each tag cloud is linked to a graph of API member sets. We expect that, by using tag cloud representations for describing overviews of functionalities, the user can intuitively select the right graph corresponding to the sets of API methods that are closely related to the functionality the user desire to implement.

When the user selects a tag cloud, the root node and its neighboring parts of the corresponding API member set graph is drawn in the right pane. Only one API member set (in a rectangle) is shown in the right pane at a time. The rectangle node in Figure 4 corresponds to the set Z in Figure 3. Oval nodes in the periphery indicate the difference between the central API member set and the sets connected in a parent–child relationship (we call those oval nodes *Diffnodes*). The user can intuitively grasp the differences between

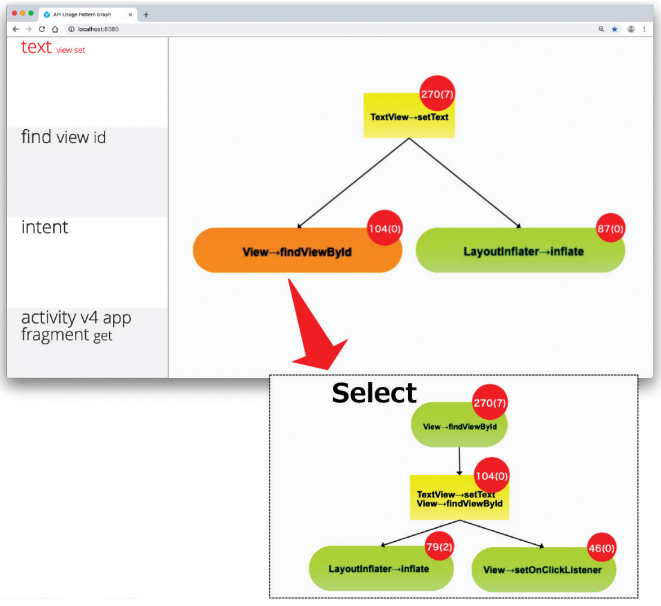


Figure 4 | Screenshot of the API member set search tool.

API member sets by looking at the small number of API methods in Diffnodes. When the user clicks a Diffnode, corresponding API member set is drawn in the center of the right pane. Thus, the user can trace the inclusion graph transferring the focus from one API member set to another. The lower part of Figure 4 illustrates a transfer of the focus from set Z to set Y in Figure 3.

As shown in Figure 4, each rectangle's frequency information is available. Numbers in and out of the parentheses are the number of source files and the number of API member sets that include the members in each rectangle, respectively. Numbers attached to Diffnodes are the same as the numbers attached to the hidden rectangles that should exist beyond the Diffnodes.

For example, We can see the rectangle shown in the center of the lower part of Figure 4 is labeled as "46(0)", meaning that there are a total of 46 source files that include calls for both "findViewById" and "setText", but no API member set consisting of only these two API methods exists, meaning that the child node beyond the Diffnode is a closed itemset generated by the frequent pattern mining.

4.2. System Structure

The structure of the proposed tool is illustrated in Figure 5. As shown in Figure 5, our tool consists of two parts; the *Database Constructor* and the *Browser*.

In the Database Constructor, three components are combined to construct the database for facilitating the API member set search. First, API member sets are extracted from a set of source files by the *API Member Set Extractor*. The grouping is carried out by using a dependency analysis; similar techniques have been used elsewhere, e.g., in the SSS system [14]. The obtained sets of API member sets are gathered and undergo the frequent pattern mining [12] by the *API Member Set Graph Generator*. In the process, obtained closed itemsets are analyzed to construct inclusion graphs of API sets. The frequency information is used to annotate nodes of the graphs.

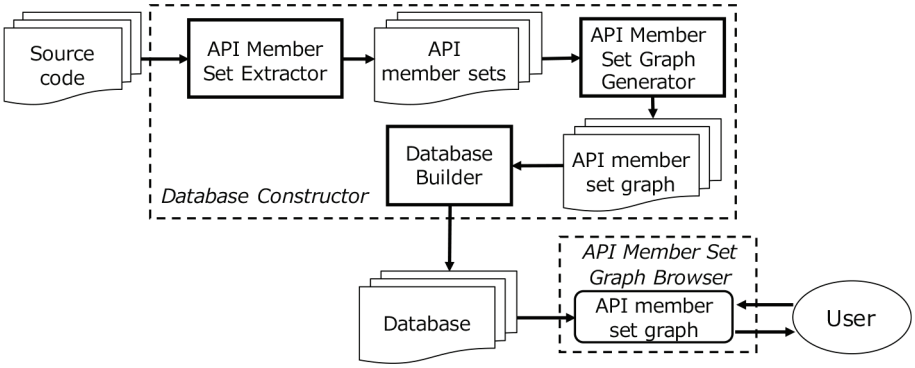


Figure 5 | Overall structure of the API member set search system.

The major part of the *Database Builder's* job is to index multiple graphs by using a list of annotations where each annotation is represented in the form of a tag cloud. Tag cloud generation is described in the next section.

The Browser is a GUI through which the user can interact with the system. The behavior of the GUI is as outlined in Section 4.1.

4.3. Tag Cloud Annotation Generation for Each Inclusion Graph

There might be multiple inclusion graphs constructed by the API Member Set Graph Generator. Some graphs could have multiple root nodes that have no parents. Each of these root nodes likely corresponds to an important API member set for implementing a particular functionality. We treat the root nodes of multiple graphs as starting points of searching.

We will explain the way to generate tag cloud annotations corresponding to all root nodes. First, we extract class names and method names from the names of API methods that make up the root node of each graph and extract words by them. Second, we apply the morphological analysis to the extracted words to normalize each word and weight them by using the TF–IDF method regarding that each set of words constructs a document.

The generated list of tag cloud annotations is shown to the user and used for deciding from which node to start tracing the inclusion graphs.

4.4. Implementation of the Prototype

We have implemented a prototype of the API member set search system. We used Java language to write the prototype. SPMF [15] is used for the frequent pattern mining in API Member Set Graph Generator, and Apache Lucene (https://lucene.apache.org/core/) libraries are used to build Database Builder.

5. EVALUATION

To evaluate the usefulness of the proposed method and the tool, we conducted a set of experiments on the API member set search by using the prototype of the tool.

Table 1 | Common Android components

AlertDialog: Display warning messages on the screen
Toast: Display messages on the screen for a short time
TextView: Display messages on the screen
Snackbar: Show messages at the bottom of the screen
Notification: Display messages on the notification drawer
Seekbar: Help adjust parameters by a draggable thumb
WakeLock: Change sleeping states of the device
BroadcastReceiver: Respond to broadcast messages

5.1. Datasets Used for the Experiments

We use two kinds of datasets in the experiments:

Google Samples: The set of API methods obtained from Google Samples (https://github.com/googlesamples). It contains 6112 API member sets extracted from 158 Android Java projects, obtained in January 2017.

Github: The set of API methods collected from Android projects on Github. The dataset was collected in July 2019. A selected part of the set is used as a dataset in each experiment. In all cases, the size of the dataset is larger than that of Google Samples.

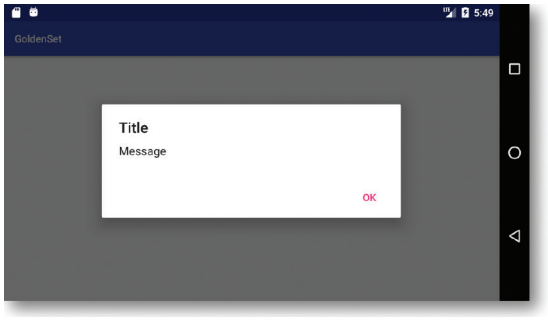
In both cases, the minimum support threshold is set to 0 while applying the frequent pattern mining. All experiments were performed on a MacBook Pro (CPU: Intel Core i7 3.5 GHz, Memory: 16 GB).

5.2. Usability of the Tool: Search for API Member Sets by Tracing Inclusion Graphs

We conducted several attempts of the API member set search to retrieve information that should be useful for implementing the Android components listed in Table 1. The dataset from Google Samples was used for each search. In fact, all the experimental results were similar. We show the usability of the tool by describing how the tool would be used for searching for an API member set to implement the AlertDialog component.

Suppose that you are to implement the AlertDialog pop-up shown in Figure 6a and search for the set of APIs required for implementing using the tool. An AlertDialog pop-up in Figure 6a consists of a title, a message, and a clickable button. Search for the AlertDialog component would be carried out smoothly and intuitively as follows.

(a)



(b)

```
new AlertDialog.Builder(this)
                .setTitle("Title")
                .setMessage("Message")
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int whichButton) {}
                })
                .show();
```

Figure 6 | AlertDialog component of the Android API. (a) AlertDialog on the Android screen. (b) AlertDialog implemented by combining API methods.

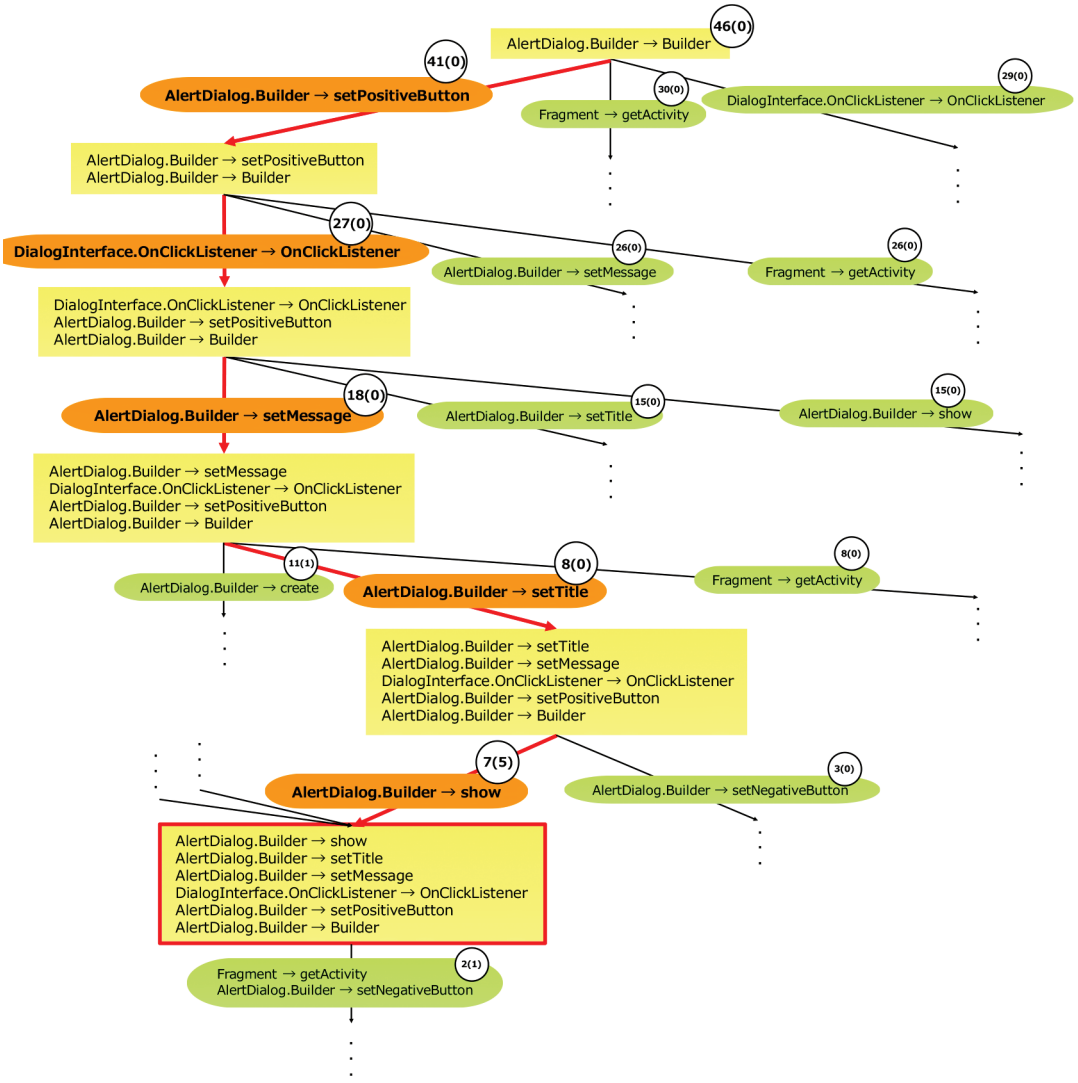


Figure 7 | Searching for an API member set by tracing an inclusion graph.

You might choose a tag cloud with the words {Builder, Alert, Dialog} to start searching. Then the tool would display an API member set that includes a single API method `AlertDialog.Builder.Builder`, which is at the root of the inclusion graph shown in Figure 7.

In Figure 7, we can see a path from the root to the box at the lower left, on which API method names shown in Diffnodes include words such as “title”, “message”, “button”, and “click”, which indicate that they are related to the components that make up parts shown in Figure 6a. We can easily reach to the lower left box taking frequency information into account; in fact, the number on the lowermost Diffnode in Figure 6a indicates that taking a further step would not gain the appropriateness of the set very much.

Figure 6b shows an example of a code segment that uses all API methods in the obtained set.

Table 2 | Numbers of API member sets extracted from Google Samples

Upper limit of API member set size	–	50	30	20	10
API member sets	6112	6099 (99.8%)	6067 (99.3%)	6001 (98.2%)	5731 (93.8%)
API member sets (duplicates omitted)	2373	2360 (99.5%)	2330 (98.2%)	2270 (95.7%)	2080 (87.7%)
Number of closed itemsets	8375	7435 (88.8%)	6178 (73.8%)	4885 (58.3%)	2312 (27.6%)
Elapsed time for extracting[s]	29.6	11.3 (62.3%)	7.3 (24.3%)	5.3 (17.7%)	4.71 (15.8%)

Table 3 | Numbers of API member sets related to specific functionalities

Upper limit of API member set size	–	50	30	20	10
Music playback (android.media.MediaPlayer)	7	7	5	3	3
Camera (android.hardware.camera2)	4	1	1	1	1
Recording (android.media.MediaRecorder)	2	1	1	0	0

5.3. Constructing an Inclusion Graph: the More You Pay, the More You Get

The frequent pattern mining is known to be a computationally expensive method. Table 2 summarizes the effect of putting a restriction on the maximum size of API member sets while processing the data form Google Samples. The numbers in parentheses in Table 2 show the percentages against the leftmost numbers. As shown in Table 2, we can see that the time required to construct the database for API member set search significantly decreases when the maximum size of each API member set is limited. The negative effect of the limitation of maximum API member set size might look small in Table 2. However, the lost information includes that on API methods related to major classes as listed in Table 3.

All the listed classes in Table 3 tend to require to combine many API methods to realize a specific functionality. Table 3 shows the number of API method sets that include indispensable API methods to implement each functionality. A set of functionality, such as Camera, which requires many API methods to implement and which can often be combined with other functionalities, is heavily influenced by the size limitations.

The loss shown in Table 3 might be attributed to the fact that there are too few sample projects that use cameras or music recording facilities in Google Samples. However, it might not be a good idea to limit the size of the set because there is a risk of losing some amount of important information.

5.4. Searching Github: a Larger Source of Information

Although Google Samples contains many informative examples for developing Android applications, the set of examples does not cover all of the Android APIs and some functionalities are barely involved. In this section, we describe what happens when you use a dataset obtained from a wider range of Github.

Figure 8 shows the distribution of the API method sets obtained from Google Samples and Github. All API method sets counted in Figure 8 are related to the Android components listed in Table 1 (498 projects on Github). Figure 8 illustrates that Github offers a richer variety of data. Figure 9a and b shows the same situation of searching API method sets related to the WakeLock component. Comparing Figure 9a and b, you can see that the API method

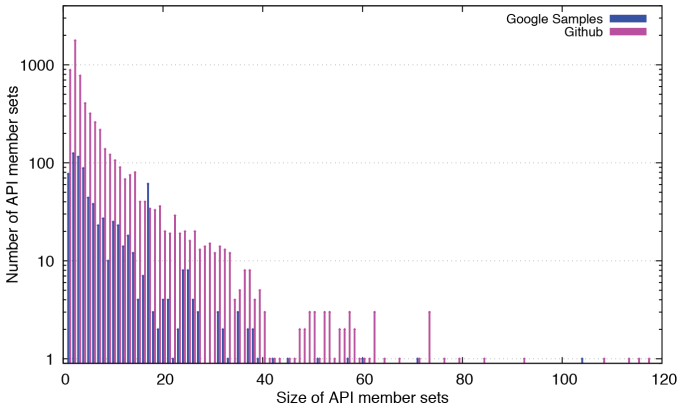


Figure 8 | Distribution of the sizes of API member sets related to the components in Table 2.

usages demonstrated in Google Samples are quite limited. Figure 10 shows one of the extreme cases where Google Samples can not offer a candidate but Github presents a promising set with modification choices.

As such, larger dataset leads a better performance in general. The proposed tool appears to be able to make the most use of larger dataset.

6. RELATED WORK

Extraction of API usage patterns for supporting program development has been studied for a decade. MAPO [3] extracts API usage patterns by applying clustering to the API method call sequences collected from the source code. The obtained information is used for API method recommendation and code snippet presentation. The presentation of candidates is in the form of a list. UP-Miner [4], that claims it outperforms MAPO [3], incorporates a method to mine frequent API method call sequences and visualize the mined sequence information with a probability graph. Their main focus seems to be dealing with information on method call sequences for recommending API usages. Our approach is different in that we rather pay attention to dealing with related method calls scattered in a program to support the development of event-driven applications for which you can not depend only on method call sequences. Instead of recommending a piece of code to be inserted at a certain place in the source code, our tool helps search for an API member set to be used together in the source code. This supports the development of event-driven applications.

DroidAssist [5], which supports the development of Android applications, is an API recommendation tool based on Hidden Markov Model. DroidAssist supports the step-wise development of a program efficiently. Our approach offers the other kind of support of

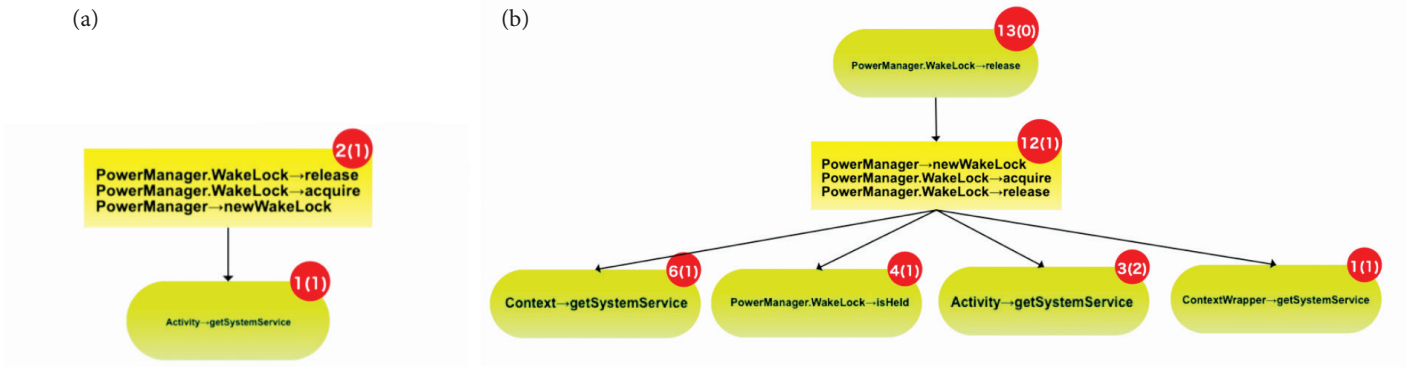


Figure 9 | Parts of API member set graphs related to a WakeLock component. (a) Based on Google Samples. (b) Based on Github.

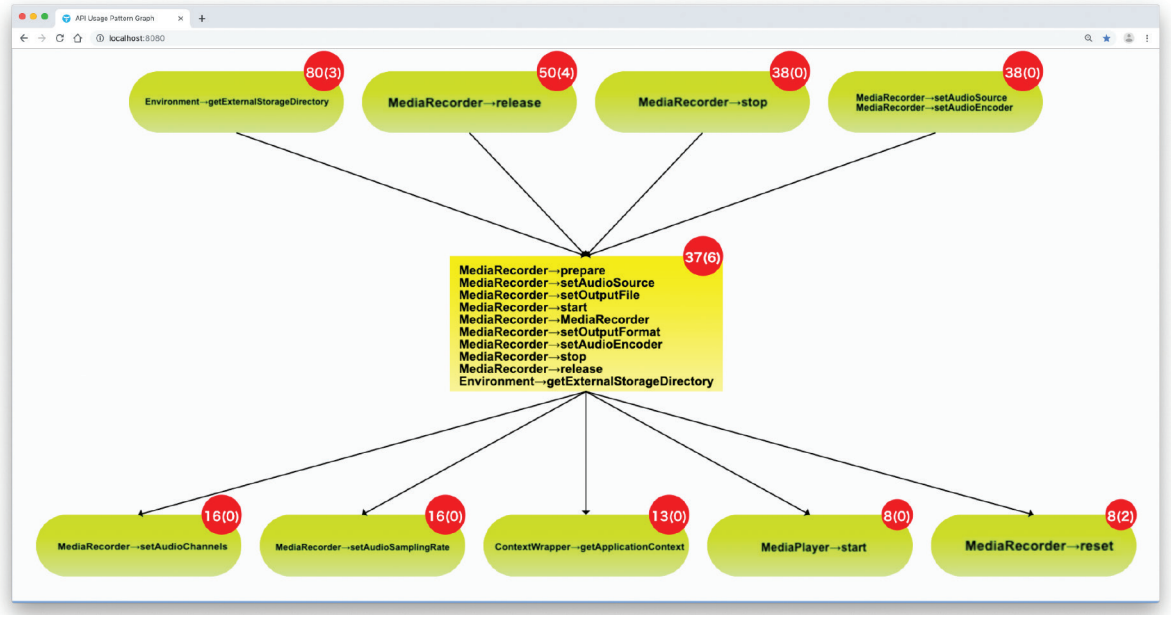


Figure 10 | Part of an API member set graph related to the MediaRecorder component based on Github.

the user; our tool can be used for superposing functionalities one at a time like the SSS tool [14].

Examplore [10] recommends the usage of an API method designated by the user. When recommending, the system presents a code skeleton that contains information such as enclosing control structures and the hints on how to handle arguments and has an interactive interface that completes the code interactively. It is similar to our tool in that it can present and search certain API member sets, but our tool does not require the user to provide the key API method in advance. In addition, our tool can treat multiple API methods as a set where each of which might be used in separate locations in a source program.

ExPort [10] supports hierarchical API method search utilizing Relational Topic Model. After determining the key API method from the global view, the presentation of information based on the call graph is done. It is similar to our system's idea in that it focuses not only on local sequences but also on the relationship between global API methods.

Application Programming Interface usage patterns are useful for debugging as well as program development support. PR-Miner [6] uses the frequent pattern mining to automatically extract undocumented implicit programming rules. The extracted patterns can be used to detect violations of API usages in the program. DynaMine [16] extracts application-specific patterns using revision history and has been shown to be effective for violation detection in large-scale applications.

7. CONCLUSION AND FUTURE WORK

We have proposed a method for effectively presenting the inclusive relations among API member sets extracted from a large number of the set of API member sets for the purpose of improving the searchability of appropriate API member sets to support the development of event-driven applications. We have developed the prototype of the system and carried out several experiments to evaluate the system. Experimental results show that the proposed

method is easy to use and the obtained information is useful for application development.

There are several approaches to improve our method. The proposed system provides the user with a technique to search for the API member set that is required for implementing the desired functionality based on the keyword given by the user. However, since the information obtained as a search result is a set of API method names, it would take some amount of time to reflect the obtained information into the program at hand. It would be effective to integrate the feature [14] of automatically merging (superposing) the selected code skeleton into the code at hand.

The proposed system offers a simple way for searching API member sets to the user based on API member set graphs. By utilizing API member set graphs, we expect that an effective recommendation of API member sets with high utility could be realized.

We have used Google Samples as an open source repository for extracting API member sets and evaluating the usability of the proposed system as a support tool for developing Android applications. In addition, we confirmed that the tool can make use of larger datasets. Our future work includes detailed evaluations of the tool's applicability and effectiveness in the case of the usage of frameworks for other than Android. We also plan to carry out the evaluation based on the user study.

CONFLICTS OF INTEREST

The author declares they have no conflicts of interest.

REFERENCES

[1] J.E. Montandon, H. Borges, D. Felix, M.T. Valente, Documenting APIs with examples: lessons learned with the APIMiner platform, 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, Koblenz, Germany, 2013, pp. 401–408.

[2] Y. Lamba, M. Khattar, A. Sureka, Pravaaha: mining android applications for discovering API call usage patterns and trends, Proceedings of the 8th India Software Engineering Conference (ISEC), ACM, Banglore, India, 2015, pp. 10–19.

[3] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, MAPO: mining and recommending API usage patterns, in: S. Drossopoulou (Eds.), European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 5653, Springer-Verlag, Berlin Heidelberg, 2009, pp. 318–343.

[4] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, D. Zhang, Mining succinct and high-coverage API usage patterns from source code, 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, San Francisco, CA, USA, 2013, pp. 319–328.

[5] T.T. Nguyen, H.V. Pham, P.M. Vu, T.T. Nguyen, Recommending API usages for mobile apps with hidden Markov model, 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, Lincoln, NE, USA, 2015, pp. 795–800.

[6] Z. Li, Y. Zhou, PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code, Proceedings of the 10th European Software Engineering Conference held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), ACM, Lisbon, Portugal, 2005, pp. 306–315.

[7] M. Nishimoto, H. Kawabata, T. Hironaka, A system for API set search for supporting application program development. IEICE Trans. Inf. Syst. J101-D (2018), 1176–1189 (in Japanese).

[8] R. Hoffmann, J. Fogarty, D.S. Weld, Assieme: finding and leveraging implicit references in a web search interface for programmers, Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST), ACM, Newport, Rhode Island, USA, 2007, pp. 13–22.

[9] E. Duala-Ekoko, M.P. Robillard, Using structure-based recommendations to facilitate discoverability in APIs, Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP), ACM, Lancaster, UK, 2011, pp. 79–104.

[10] E.L. Glassman, T. Zhang, B. Hartmann, M. Kim, Visualizing API usage examples at scale, Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI), ACM, Montreal QC, Canada, 2018, pp. 580:1–580:12.

[11] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, M. Gethers, Export: Detecting and visualizing API usages in large source code repositories, 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, Silicon Valley, CA, USA, 2013, pp. 646–651.

[12] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD), ACM, Dallas, Texas, USA, 2000, pp. 1–12.

[13] G. Grahne, J. Zhu, Fast algorithms for frequent itemset mining using FP-trees, IEEE Transactions on Knowledge and Data Engineering, IEEE, 2005, 1347–1362.

[14] M. Nishimoto, K. Nishiyama, H. Kawabata, T. Hironaka, Easy-going development of event-driven applications by iterating a search-select-superpose loop, J. Inform. Process. 27 (2019), 257–267.

[15] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.W. Wu, V.S. Tseng, SPMF: a Java open-source pattern mining library. J. Mach. Learn. Res. 15 (2014) 3569–3573.

[16] B. Livshits, T. Zimmermann, DynaMine: finding common error patterns by mining software revision histories, Proceedings of the 10th European Software Engineering Conference held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), ACM, Lisbon, Portugal, 2005, pp. 296–305.