Data-Driven Web APIs Recommendation for Building Web Applications

Lianyong Qi[1], Qiang He[2,*], Feifei Chen[3], Xuyun Zhang[4], Wanchun Dou[5], Qiang Ni[6]

Abstract—The ever-increasing popularity of web APIs allows app developers to leverage a set of existing web APIs to achieve their sophisticated objectives. The heavily fragmented distribution of web APIs makes it challenging for an app developer to find appropriate and compatible web APIs. Currently, app developers usually have to manually discover candidate web APIs, verify their compatibility and select appropriate and compatible ones. This process is cumbersome and requires detailed knowledge of web APIs which is often too demanding. It has become a major obstacle to further and broader applications of web APIs. To address this issue, we first propose a web API correlation graph built on extensive data about the compatibility between web APIs. Then, we propose WAR (Web APIs Recommendation), the first data-driven approach for web APIs recommendation that integrates web API discovery, verification and selection operations based on keywords search over the web API correlation graph. WAR assists app developers without detailed knowledge of web APIs in searching for appropriate and compatible web APIs by typing a few keywords that represent the tasks required to achieve app developers' objectives. WAR can significantly save app developers' time and effort in searching for web APIs. We conducted large-scale experiments on 18,478 real-world web APIs and 6,146 real-world apps to demonstrate the usefulness and efficiency of WAR.

Index Terms—Web APIs recommendation, Keyword search, Steiner Tree, Dynamic Programming

1 Introduction

With the increasing popularity of web of things, a lot of enterprises and organizations, including software vendors like Google[1], Amazon[2], Spotify[3] have published their business functions online as web APIs that can be accessed remotely. The statistics published on several reputable web APIs repositories, e.g., programmableweb.com and mashape.com, indicate a rapid growth in the number of published web APIs and their users in the past few years.

The web of things allows mobile and web apps (together referred to as apps hereafter) to invoke appropriate web APIs to achieve their goals. For example, a mobile app developer can find and select the right web APIs to be integrated into their app so that it can invoke these web APIs to fulfill its end-user's sophisticated needs. Figure 1 shows the common process for building an interview app that needs to perform four app tasks: (1) *voice recording* for recording interviews; (2) *speech recognition* for transforming interview recordings into transcripts; (3) *document translation* for translating the interview transcripts into target language(s) as necessary; and (4) *file synchronization* for saving the transcripts online. Some web APIs may be available that can perform some or all of those required app tasks, e.g., Scribie Audio and SoundCloud for *voice recording*, Web Speech and Google's Speech Recognition for *speech recognition*, Google Translate and Microsoft Translator for *document translation*, DropBox and OneDrive for *file synchronization*. If those web APIs can be identified and found, the app developer can integrate those web APIs into the app for performing those tasks.

As formally depicted in Figure 1, the process for building this app consists of three phases. The first phase is *app planning* where the app developer analyses the functional requirements and determines the tasks needed to be performed, as well as the execution order of the tasks. The second phase is *web API discovery* where, through manual web search, the app developer identifies four sets of candidate web APIs, each containing a number of candidate web APIs that can perform the tasks. The third phase is *web APIs selection* where the app developer selects one web API from each set of candidate services that collectively realize the app.

This process, sometimes referred to as *web mashup* [1], can be excessively sophisticated even for experienced app developers due to the large number of available web APIs and their wide variety. It has become a major obstacle to further and broader applications of web of things. Google has even developed an API Picker[4] that assists developers understand its APIs and select appropriate ones. However, such assistance tools are still too complicated for non-experts and do not address the following two critical issues:

- *Lianyong Qi is with the School of Information Science and Engineering, Qufu Normal University, China. E-mail: lianyongqi@qfnu.edu.cn.*
- *Qiang He is with the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. E-mail: qhe@swin.edu.au. (Corresponding author)*
- *Feifei Chen is with the School of Information Technology, Deakin University, Australia. E-mail: feifei.chen@deakin.edu.au.*
- *Xuyun Zhang is with the Department of Computing, Macquarie University, Australia, 1010. E-mail: xuyun.zhang@mq.edu.au.*
- *Wanchun Dou is with the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, China, 210023. E-mail: douwc@nju.edu.cn.*
- *Qiang Ni is with the School of Computing & Communications, Lancaster University, UK. E-mail: q.ni@lancaster.ac.uk.*

* Correspondence should be address to: *Qiang He.*

[1] https://developers.google.com/maps/get-started/
[2] https://developer.amazon.com/services-and-apis/
[3] https://developer.spotify.com/web-api/
[4] https://developers.google.com/maps/documentation/api-picker

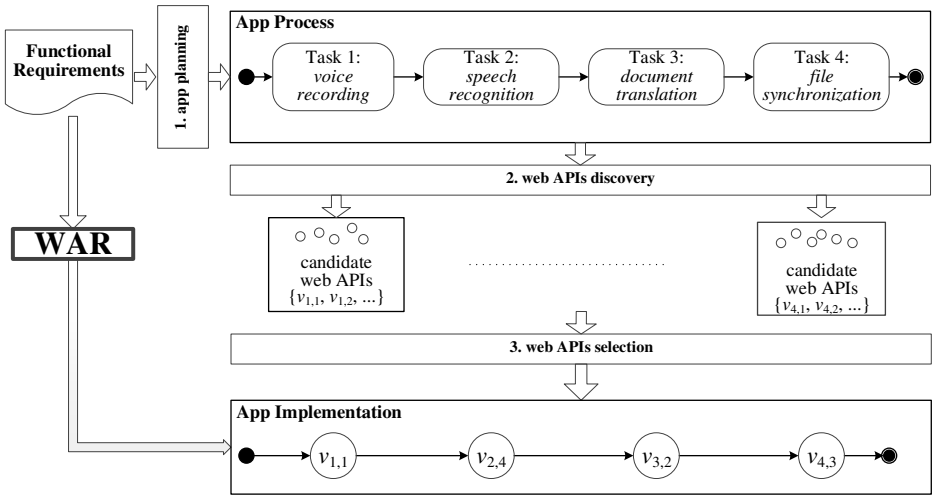


Figure 1. An example interview app. (Note: Here, we assume that all four app tasks are implemented by remote web APIs, which is not always necessary.)

(1) In the *web APIs discovery* phase, the app developer needs to perform an intensive manual search over the web to identify candidate web APIs. This requires the identification of and visits to a large number of software vendors' API websites.

(2) Even if the candidate web APIs can be identified, in the *web APIs selection* phase, the app developer needs to evaluate and verify the compatibility between the candidate web APIs, i.e., whether the output of one web API can be directly fed to the next web API as input without making significant changes. This requires in-depth analyses of all candidate APIs.

The above operations are often too cumbersome and time-consuming, especially for non-experts. Thus, there has a rapid increase in the need for an approach that allows app developers to find web APIs for developing their apps without having to go through the above cumbersome and time-consuming phases individually.

Online web APIs repositories, such as programmableweb.com and mashape.com provide a large amount of data about web APIs and allow app developers to search for web APIs by keywords. This keyword search method has long been popularised by web search engines like Google and Bing in locating information from web documents [2][3], as well as the databases community in locating information from databases [4][5][6]. However, none of the existing keyword search techniques can be directly applied to effectively address the abovementioned two critical issues.

In this paper, we propose WAR (**W**eb **A**PIs **R**ecommendation), a novel approach to web APIs recommendation that assists app developers in searching for multiple compatible web APIs based on keyword search. As shown in Figure 1, WAR integrates and automates the *app planning*, *web APIs discovery* and *web APIs selection* operations, offering a novel data-driven approach for building apps. It allows app developers to search for multiple appropriate and compatible web APIs by entering only a few keywords that represent the required tasks for the app. WAR runs on a directed data graph, where web APIs are modeled as nodes connected by edges representing whether the web APIs are compatible. Given a set of keywords, WAR returns subgraphs of the data graph that represent diversified solutions to target app. Each solution includes the web APIs that perform the app tasks, the bridging web APIs (if any) needed however not specified by the keywords, and the composability of those web APIs, i.e., whether and how they can be integrated.

An app developer, without having to manually discover and evaluate candidate web APIs from the web, can easily use WAR to find the web APIs needed to build the interview app depicted in Figure 1. Suppose the app developer would like this app to be able to record voices, recognize speeches from voice recordings, translate speech transcripts and synchronize translated speech transcripts online. Using WAR, the app developer only needs to enter the keywords that describe those app tasks: *voice recording*, *speech recognition*, *document translation* and *file synchronization*. WAR will take those keywords, search its web API repository, and return diversified app solutions that consist of different sets of compatible web APIs. The app developer can select one of the app solutions that best suits their needs and implement the app according to the selected solution. WAR can find an app solution even when the app developer is not able to provide all the keywords for describing the app tasks. For example, an app developer enters three keywords *voice recognition*, *document translation*, and *file synchronization*, hoping to build an interview app. However, this app requires a *speech recognition* web API that succeeds the *voice recording* web API and precedes the *document translation* web API. WAR can automatically identify the missing *speech recognition* web API and provide the app developer with a complete app solution. This way, WAR can save app developers a lot of time and efforts for finding out what web APIs are needed and can be used to implement their apps.

In summary, we make the following main contributions:

(1) We propose a novel data-driven approach for efficiently building apps by integrating and automating the app planning, web APIs discovery and web APIs selection operations and relieving app developers of the detailed knowledge of potentially enormous candidate web APIs for building their target apps.

(2) We propose a web API correlation graph where web APIs are modeled as nodes and their compatibility as edges. This correlation graph is built based on extensive data about web APIs' compatibility obtained from mining online software repositories.

(3) Based on the API correlation graph, we model app developers' queries for web APIs as minimum group Steiner tree problem and solve it with dynamic programming technique to recommend diversified solutions.

(4) We conduct experiments on a dataset that contains the functional information about 18,478 real-world web APIs and 6,146 real-world apps crawled from programmableweb.com, to evaluate the usefulness and efficiency of WAR.

The rest of paper is organized as follows. Section 2 presents the API correlation graph. Section 3 formulates the research problem. Section 4 introduces how WAR answers keyword queries for app solutions based on the API correlation graph. Section 5 evaluates WAR with experimental results. Section 6 reviews the related work. Section 7 concludes the paper.

2 Web API Correlation Graph

Many web apps developed with web APIs have been published on many of the online software repositories. For example, a total number of 6,146 web apps have been developed and published on programmableweb.com. Such apps provide valuable information about their constituent web APIs' compatibility. For example, a web app $\{api_1, api_2\}$ published on programmableweb.com indicates that api_1 and api_2 are compatible because its developer has verified their compatibility and successfully integrated them into the web app. Such compatibility information allows a web API correlation graph to be built through connecting compatible web APIs mined from the online software repositories. Suppose another published web app $\{api_2, api_3\}$ in addition to $\{api_1, api_2\}$. In the correlation graph, api_1 is connected to api_2 and api_2 is connected to api_3. As more APIs are included, the correlation graph will grow larger and denser, offering a solid base for queries for building apps.

WAR does not stop other methods from being employed to evaluate and verify the web APIs to be included into the correlation graph. It runs any correlation graphs that fulfill the requirements specified by the following definitions:

DEFINITION 1. Nodes: For each web API in the repository, correlation graph G has a corresponding node v. Each node in G contains one or multiple keywords $k_{1, \ldots, k_n}$ that represent the functions offered by the corresponding API.

In the remainder of this paper, we will speak interchangeably of a web API and its corresponding node in G, both denoted as v. Please note that *voice recognition* has two terms, however is considered as one keyword, not two. Please also note that the issues of synonymy, word inflections and polysemy are handled with automatic query expansion techniques [7]. It is, however, out of the scope of this paper.

DEFINITION 2. Edges: For each pair of compatible v_1 and v_2, the correlation graph contains an edge $e(v_1, v_2)$ between v_1 and v_2. $e(v_1, v_2)$ is directed, pointing from v_1 to v_2 if v_2 can be the succeeding node of v_1 when they are integrated. An edge e can be bidirectional if v_1 can also be the succeeding node of v_2. Moreover, each edge $e(v_i, v_j)$ in E is assigned a weight $w_{i,j}$ ($w_{i,j} = w_{j,i}$ holds here) to indicate the total times that v_i and v_j have ever been invoked simultaneously before by all users.

DEFINITION 3. Correlation Graph: A correlation graph is represented by $G(V, E)$ where V and E denote its sets of nodes and edges, respectively.

According to Definition 2, relevant web APIs in the same domain are connected, either directly or indirectly, forming a connected correlation graph. However, a web API repository might contain APIs in different domains, e.g., *voice recognition* and *sunset times query*, which belong to different correlation graphs. It is possible that a API repository has multiple disconnected correlation graphs. However, an app developer usually will not enter entirely irrelevant keywords that belong to two domains. Thus, in this research, we do not consider such keyword queries.

To answer keyword queries for building apps (as will be detailed in Section 4), WAR prebuilds an inverted index for G. For each keyword in G, the nodes covering the keyword are stored in this index. For example, if nodes v_1, v_5 and v_8 cover keyword k_6, there is $V(k_6) = \{v_1, v_5, v_8\}$. This way, given an individual keyword k, WAR can easily find all the web APIs that can perform the task described by k.

3 Problem Formulation

Given a correlation graph G and a keyword query Q containing l ($l \geq 2$) keywords ($Q = \{k_1, \ldots, k_l\}$), the problem of answering the query over G consists of two steps: (1) to find an *answer tree*, denoted as $T(Q)$ in G, containing connected nodes that cover all the keywords in Q; (2) to induce the final answer based on the $T(Q)$.

Consider the example in Figure 2. The graph in Figure 2 is part of a correlation graph G. It contains 14 nodes (web APIs), i.e., $v_1, \ldots, v_{14}$ covering 14 keywords (functions), including $k_1, \ldots, k_{14}$. The notation $v_1\{k_1\}$ indicates that v_1 offers k_1. Note that $v_2\{k_1, k_4\}$ indicates that API v_2 is able to offer two functions, k_1 and k_4, which is allowed by WAR. Edge $e(v_1, v_{10})$ and weight $w_{1,10} = 2$ indicate that v_1 and v_{10} are compatible and have been integrated twice in the past. Note that other edges and weight values are omitted to ensure the concision of Figure 2.

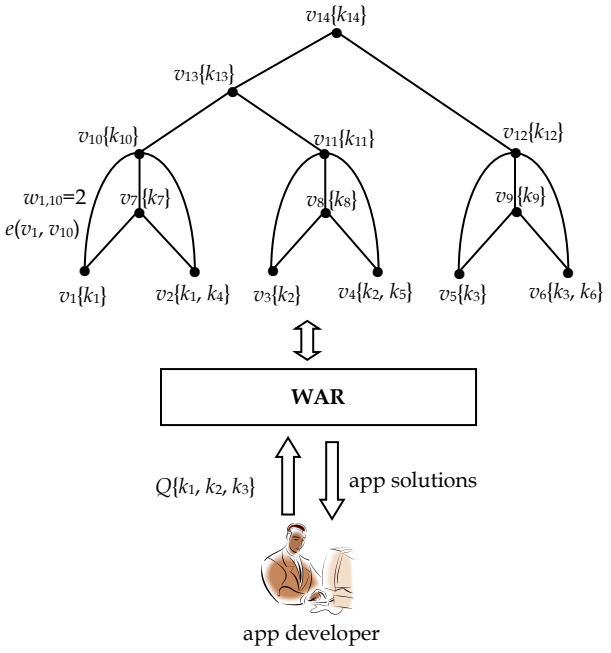


Figure 2. API correlation graph: an example

The app developer enters three keywords $\{k_1, k_2, k_3\}$ to query a set of web APIs as an app solution that performs tasks described by $\{k_1, k_2, k_3\}$. We can see that v_1 and v_2 contain k_1, v_3 and v_4 contain k_2, v_5 and v_6 contain k_3. Given query $Q\{k_1, k_2, k_3\}$, we are looking for an answer tree that connects one node from $\{v_1, v_2\}$, one node from $\{v_3, v_4\}$ and one node from $\{v_5, v_6\}$. This answer tree will need to connect nodes that do not cover any of k_1, k_2 and k_3, e.g., v_{10}, v_{11}, v_{12} and v_{13}. Thus, it is a Steiner tree [8], formally defined below:

DEFINITION 4. Steiner Tree. Given a graph $G = (V, E)$ and $V' \subseteq V$, T is a Steiner tree of V' in G if T is a connected subtree in G that covers all nodes in V'.

Using the inverted index introduced in Section 2, we can identify the groups of nodes in G corresponding to individual keywords in $Q = \{k_1, \ldots, k_l\}$, denoted as $V_1, \ldots, V_l$ where V_r $(1 \le r \le l)$ is the set of nodes in G that cover k_r $(1 \le r \le l)$. The problem is now to find a *group Steiner tree*, formally defined below:

DEFINITION 5. Group Steiner Tree: Given a graph $G = (V, E)$ and groups $V_1, \ldots, V_l \subseteq V$, where $V_i \cap V_j = \emptyset, \forall\ V_i, V_j$ $(0 \le i, j \le l$ and $i \ne j)$, T is an group Steiner tree of $V_1, \ldots, V_l$ in G if T is a Steiner tree that contains exactly one node from each group V_r $(1 \le r \le l)$.

In response to a query, there are usually multiple group Steiner trees. WAR aims to find the *minimum group Steiner tree* with the minimum number of nodes, including *keyword nodes*, i.e., nodes containing the query keywords, and *bridging nodes*, i.e., nodes that do not contain the keywords but are necessary to connect the keyword nodes. This optimization objective is employed because the solution will require the fewest web APIs. This potentially simplifies the app developer's web app and reduces the cost for building the web app. A minimum group Steiner tree is defined as follows:

DEFINITION 6. Minimum Group Steiner Tree. Given a set of exact group Steiner trees in G, $T_1, \ldots, T_n$, T_i $(0 \le i \le n)$ is the minimum exact group Steiner tree if $|T_i| = \min(|T_1|, \ldots, |T_n|)$ where $|T_i|$ $(0 \le i \le n)$ represents the cardinality of T_i, i.e., the number of nodes in T_i.

The computation of a minimum group Steiner tree is already NP-complete [9]. WAR finds the minimum group Steiner tree with the dynamic programming (DP) technique, which is to be discussed in the next section.

4 WAR Mechanisms

This section discusses the DP-based search method WAR which is employed to answer keyword queries. In this section, we denote all the keywords entered by the app developer in query Q as K, e.g., $K = \{k_1, k_2, k_3\}$ in Figure 2, the minimum group Steiner tree rooted at node v and containing all keywords in K' as $T(v, K')$ where $K' \subseteq K$ and $K' \ne \Phi$. Given a query Q, WAR finds $T_{min}(v, K)$, the minimum group Steiner tree rooted at v and containing all keywords in K.

4.1 Finding Optimal Solution

Dynamic programming technique solves a given complex optimization problem by breaking it down into a collection of simpler subproblems. Each of the subproblems is solved only once and the corresponding result is stored. Through examining and combining previously solved subproblems, a dynamic programming algorithm can solve the given complex optimization problem exactly. In this research, a minimum group Steiner tree $T(v, K)$ of height h (the length of the longest downward path from the root of the group Steiner tree to any leaf) can be found by expanding the group Steiner trees of heights $h = 0, 1, \ldots$, that cover $K' \subseteq K$ [10]. Let $T(v, K')$ be a state in the dynamic programming model, and $w(T(v, K'))$ be the weight of $T(v, K')$, i.e., the total number of the nodes in $T(v, K')$, the state-transition equation in the dynamic programming model is:

$$w(T_{\min_{\forall v \in V_k}}(v, K')) = 1 \quad \text{IF } |K'| = |\{k\}| = 1 \tag{1}$$

$$w(T_{min}(v, K')) = \min(w(T_g(v, K')), w(T_m(v, K'))) \tag{2}$$

$$w(T_g(v, K')) = \min_{u \in N(v)} \{w(T_{min}(u, K') + v)\} \tag{3}$$

$$w(T_m(v, K')) = \min_{\substack{K' = K_1' \cup K_2' \\ K_1' \cap K_2' = \emptyset}} \{w(T_{min}(v, K_1') \oplus T_{min}(v, K_2'))\} \tag{4}$$

where $N(v)$ is the set of node v's neighbors in G, i.e., $v \in G(V, E)$ and $e(u, v) \in E$. Formula (1) indicates that the weight of any tree with only one node is 1. Formula (2) indicates that $T(v, K')$ can be obtained through either one of two operations, *tree growth*, formally represented by Formula (3), and *tree merging*, formally represented by Formula (4). The *tree growth* operation generates a new minimum group Steiner tree $T_{min}(v, K')$ by adding node v to $T_{min}(u, K')$ rooted at u (one of v's neighbors). The *tree merging* operation generates $T_m(v, K')$ by merging two trees both rooted at v, one covering K_1' and the other covering K_2' such that $K' = K_1' \cup K_2'$ and $K_1' \cap K_2' = \emptyset$.

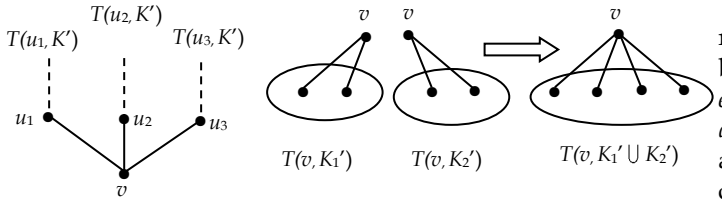


Figure 3. *Tree growth* and *tree merging* operations.

Figure 3 illustrates the *tree growth* and *tree merging* operations. In the left part of Figure 3, u_1, u_2 and u_3 are the neighboring nodes of node v. Tree $T(v, K')$ is generated by selecting the tree with the minimum weight from the three trees generated from adding v to $T(u_1, K')$, $T(u_2, K')$ and $T(u_3, K')$. The right part of Figure 3 shows two trees rooted at node v and containing K_1' and K_2', i.e., $T_{min}(v, K_1')$ and $T_{min}(v, K_2')$ can be merged into a larger tree that is rooted at node v and contains more keywords, i.e., $K_1' \cup K_2'$.

Through repeating the *tree growth* and *tree merging* operations, the trees are continuously expanded until $K' = K$, meaning the minimum group Steiner tree is found. Based on Formulas (1)-(4), WAR employs Algorithm 1 to answer a query Q. Here, "1 + T" means the number of nodes of tree T plus one.

Algorithm 1: *MST* (*G*, *K*)

Input:
$G(V, E)$: the correlation graph
$K = \{k_1, k_2, \ldots, k_l\}$: query keywords in Q
Output:
$T_{min}(v, K)$: a minimum group Steiner tree rooted at v and containing all keywords in K

```
1  Let Q_T be a queue in ascending order of number of tree nodes
2  Q_T = Φ
3  for each v ∈ V do
4     if v contains any nonempty keyword set K′ ⊆ K
5     then enqueue T_min(v, K′) into Q_T
6  Min_count = ∞  // number of nodes of minimal Steiner Trees
7  while Q_T ≠ Φ do
8     dequeue Q_T to T_min(v, K′)
9     if K′ = K
10    then if count (T_min(v, K′)) ≤ Min_count
11          then Min_count = count (T_min(v, K′))
12               return T_min(v, K′)
13          else break
14    else
15       for each u ∈ N(v) do
16          if 1 + T_min(u, K′) < T_min(v, K′)
17          then T_min(v, K′) = 1 + T_min(u, K′)
18               enqueue T_min(v, K′)
19               update Q_T
20       K1′ = K′
21       for each K2′ s.t. K1′ ∩ K2′ = Φ do
22          if T_min(v, K1′) ⊕ T_min(v, K2′) < T_min(v, K1′ ∪ K2′)
23          then T_min(v, K1′ ∪ K2′) = T_min(v, K1′) ⊕ T_min(v, K2′)
24             if K1′ ∪ K2′ = K
25             then if count (T_min(v, K)) ≤ Min_count
26                  then Min_count = count (T_min(v, K))
27                       return T_min(v, K)
28                  else brea
29             else enqueue T_min(v, K1′ ∪ K2′)
30                  update Q_T
```

Algorithm 1 maintains a queue Q_T that stores and ranks the generated trees in ascending order by the number of their constituent nodes. It uses three operations, the *enqueue* operation that inserts a tree into queue Q_T, the *dequeue* operation that removes the top tree in queue Q_T, and the *update* operation that ranks the trees in Q_T in ascending order by the total number of the nodes in the trees. Lines 3-5 locate nodes that contain individual keywords in K. For each node v in G, $v \in V$, if v contains any keywords K' in K, $K' \subseteq K$, the algorithm enqueues tree $T(v, K')$ into Q_T. At this stage, for each such tree in Q_T, there is $|T(v, K')| = 1$ because there is only one node in each of the trees in Q_T. Lines 15-19 implement the *tree growth* operation (Formula 3); Lines 20-30 denote the *tree merging* operation (Formula 4) of a tree. If a tree in Q_T contains all the keywords in K, the tree is outputted.

Next, we utilize the example in Figure 4 (Figure 4(a) is the same as Figure 2) to demonstrate the execution process of Algorithm 1 in response to Q containing $K = \{k_1, k_2, k_3\}$, which are highlighted in bold. The trees rooted at the nodes containing k_1 or k_2 or k_3, i.e., v_1, v_2, v_3, v_4, v_5, v_6 are enqueued first, which are shown in Figure 4(b). These six trees cannot be merged, but the *tree growth* operation can be performed on them. Specifically, the edges connecting any one of $v_1, \ldots, v_6$ are added to the corresponding trees and the generated trees are enqueued, which are shown in Figure 4(c). The trees in Figure 4(c) will not be merged as the *tree merging* operation cannot increase the number of query keywords covered by the generated trees. For example, the first tree $\{v_{10}, v_1\}$ and third tree $\{v_{10}, v_2\}$ in Figure 4(c) can be merged into a new tree rooted at node v_{10}, i.e., $\{v_{10}, v_1, v_2\}$. However, this new tree does not cover more query keywords than trees $\{v_{10}, v_1\}$ and $\{v_{10}, v_2\}$, as they all contain k_1 only. Thus, the *tree growth* operation is performed on the trees in Figure 4(c). The results are shown in Figure 4(d). Here, note that some trees after the tree growth operation are of no use and hence are not shown in Figure 4(d). For example, the second tree $\{v_7, v_1\}$ in Figure 4(c) can grow to v_{10} to generate tree $\{v_{10}, v_7, v_1\}$. However, the new tree $\{v_{10}, v_7, v_1\}$ contains the same query keyword (i.e., k_1) as tree $\{v_{10}, v_1\}$ but has one more node. The trees in Figure 4(d) are then merged and the results are presented in Figure 4(e). After that, the trees in Figure 4(e) grow and the results are shown in Figure 4(f). Finally, after *tree merging* operations, eight minimal Steiner trees are found, as presented in Figure 4(g).

Theorem 1 ensures that the returned answer tree is the one with the minimum number of nodes covering K.

THEOREM 1. The tree returned by Algorithm 1, $T(v, K)$ is the group Steiner tree with the minimum number of nodes.

PROOF. This can be proven by contradiction. Let $T'(v', K)$ be a tree rooted at v' with a total number of nodes smaller than $T(v, K)$. There are $|T(v, K) - T'(v, K_1')| < |T(v', K) - T'(v', K_2')|$, where "–" is the inverse operation of "+". Line 8 in the second last iteration of Algorithm 1 would dequeue $T(v', K') - T'(v, K_1')$ from Q_T and merge $T(v', K')$ and $T'(v, K_1')$ to reach $T'(v', K')$ because Q_T always has the tree with the minimum number of nodes at the top to be dequeued. The tree dequeued from Q_T in the last iteration

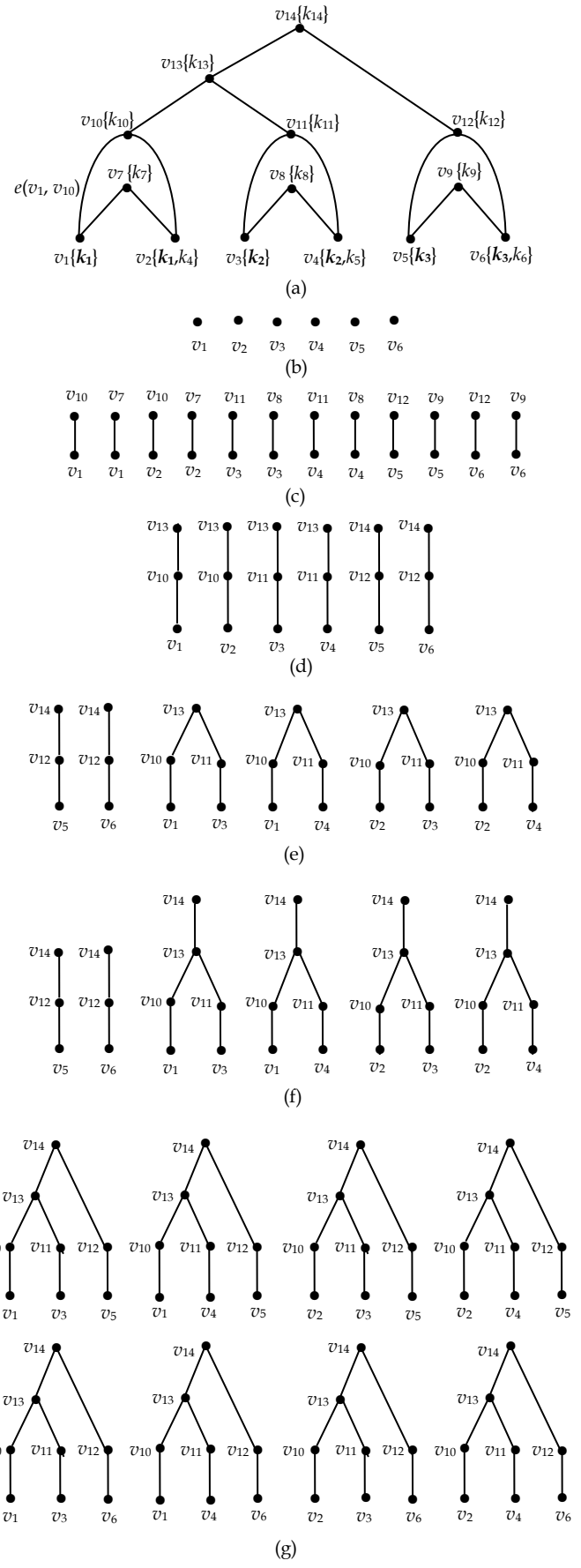


Figure 4. Minimal Steiner tree generation process.

of Algorithm 1 as the result of Algorithm 1 would be $T'(v', K)$. This contradicts with $T(v, K')$ being returned by Algorithm 1. □

We now analyze, in the worst-case scenario, the complexity of Algorithm 1 answering a query Q with a set of keywords $K = \{k_1, \ldots, k_l\}$ on a data graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. The complexity of finding all solutions is the same as finding the first solution, because the worst-case scenario for finding the first solution is to search all possible trees which is the same as finding all solutions. Thus, we analyze the complexity of Algorithm 1 for finding the first solution below.

Let $T(v, K')$ be the tree with the minimum number of nodes of all trees rooted at v containing a subset of keywords $K' \subseteq K$**.** There are 3 major components in Algorithm 1, i.e., queue maintenance, tree growth and tree merging.

Queue maintenance. In total, there are 2^l subsets of K. Thus, the maximum length of Q_T is $2^l n$, i.e., every tree rooted at any $v \in V$ containing any $K' \subseteq K$ is enqueued into Q_T. The complexity of enqueue/update operations and dequeue operations is dependent on the type of the queue. Here, we employ the Fibonacci Heap, which has the complexity of $O(1)$ for the enquene/update operations and $O(\log 2^l n)$ for dequeue operations respectively [11]. Because Algorithm 1 will enqueue or dequeue any $T(v, \mathbf{k})$ into/from Q_T at most once, the complexity of enqueuing and dequeuing all $2^l n$ trees in Q_T is $O(2^l n(l + \log n))$.

Tree growth. Lines 15-19 handle the *tree growth* operations implementing Eq. (3). The **for** loop iterates for $|N(v)|$ times, trying to find the $T(u, K')$ grown from $T(v, K') + u$ with the minimum number of nodes. Here, $|N(v)|$ is the total number of neighbors of v. Thus, the total time for Algorithm 1 to execute the comparison operations in lines 15-19 is $O(2^l \sum_{v \in V} |N(v)|) = O(2^l m)$.

Tree merging. Lines 20-30 handle the *tree merging* operations implementing Eq. (4). For each $T(v, K_1')$ dequeued in line 8, the **for** loop in lines 20-30 enumerates every K_2' that fulfils $K_1' \cap K_2' = \emptyset$, where $K_1', K_2' \subseteq K$. Given $|K| = l$, the total number of possible K_2' is $2^{l-|K_2'|}$. Thus, the total time for Algorithm 1 to execute the comparison operations in lines 16-25 is $n \sum_{i=1}^{l-1} C_{l,i} \times 2^{l-i} = O(3^l n)$.

Overall, the complexity of Algorithm 1 is $O(2^l n(l + \log n) + 2^l m + 3^l)$. This indicates that the efficiency of Algorithm 1 relies on the number of nodes and edges in the data graph, and exponentially on the number of query keywords. In real world problems where l is a small constant, the complexity of Algorithm 1 becomes $O(n \log n + m)$. We will evaluate it experimentally in Section 5.

4.2 Inducing Final Solution

The minimum Steiner tree $T(v, K)$ obtained by Algorithm 1 in response to keyword query is a subgraph of G. However, it is not the final solution for the query because some edges from G that connect $T(v, K)$'s constituent nodes in G might be missing from $T(v, K)$. Figure 5 presents an example, where $T(v, K) = \{v_1, v_2, v_3\}$. Suppose there is an edge in G that connects v_1 and v_3 in G. Algorithm

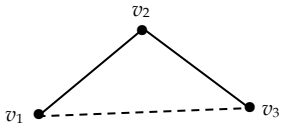


Figure 5. Inducing final solution

1 did not include this edge in $T(v, K)$ because $T(v, K)$ must not contain cycles. This missing edge might be useful for the app developer and thus needs to be included in the final solution. To induce the final solution based on $T(v, K)$, WAR inspects every pair of web APIs in $T(v, K)$ that are not connected. To increase its efficiency, WAR maintains a neighbor set for each node in G. A neighbor set $V_n(v)$ contains all v's neighbors in G.

5 Experiments

We have conducted a range of experiments on 18,478 real-world web APIs and 6,146 real-world web apps to evaluate the usefulness and efficiency of WAR.

5.1 Dataset and Deployment

We crawl from programmableweb.com a PW dataset containing information on 18,478 web APIs and 6,146 web apps. Based on the information, a correlation graph G is built. Keywords are used on programmableweb.com to describe the functions of the web APIs. Accordingly, we label the nodes in the correlation graph with those keywords. Two nodes are linked in G if they are both used by any of the 6,146 web apps with one of them succeeding or preceding the other directly. For example, suppose a web app in the dataset uses three consecutive web APIs, v_1, v_2 and v_3. Two directional edges will be included in G, one pointing from v_1 to v_2 and the other from v_2 to v_3, but none from v_1 to v_3. In total, there are 7006 edges in G. This way, the edges in G accurately describe the composability of the 18,478 real-world web APIs used by the 6,146 real-world web apps.

A total of 6,146 queries are created, one corresponding to each of the 6,146 web apps. For each query, the query keywords are extracted from the web APIs used by the corresponding web app. This way, we can evaluate the usefulness of WAR with its **success rate** in finding those 6,146 web apps from G. In real-world applications, an app developer rarely enters a large number of keywords to search for web APIs. This is evidenced by the web apps in the PW dataset. Of all the 6,146 web apps we crawled from programmableweb.com, only 223 (3.6% of all) have more than 6 keywords. Thus, we create queries with up to 6 query keywords. Three sets of experiments are conducted, i.e., set A, set B, set C. In experiment set A, the keywords in a query correspond to all the keywords of a web app. By testing 6,146 queries, we inspect whether WAR can find the web APIs needed for building each of the 6,146 web apps on programmableweb.com with all the keywords. In experiment set B, each query contains two keywords corresponding to the first and the last keywords of a web app. This way, we inspect the ability of WAR to find solutions exactly the same as or similar to the 6,146 web apps with only two keywords. In experiment set C, a number of keywords are selected randomly from G to generate each query. This allows us to evaluate the ability of WAR to find web app solutions more comprehensively with more keyword combinations. This way, we evaluate WAR's ability to find a complete solution through connecting keyword nodes and necessary bridging nodes.

As discussed in Section 3, WAR finds the minimum group Steiner tree in response to a query. Thus, WAR might be able to find solutions with fewer web APIs than the corresponding web apps in the dataset. Such solutions are considered new solutions. This ability allows WAR to find the simplest app solutions, which is usually preferable. Accordingly, we compare the **number of nodes** in the solutions with the corresponding web apps in the PW dataset. Besides, a feasible solution requires the mutual compatibility between the web APIs in the solution, i.e., the corresponding nodes are connected. In this regard, we compare the **connected rate** of the solutions returned by each of the approaches. It is the ratio of the solutions whose nodes are connected.

To evaluate the efficiency of WAR, we measure the **computation time** taken by WAR to answer queries. Fast responses will allow app developers to trial different keyword combinations to find suitable solutions for their apps.

WAR offers a new data-driven approach for finding multiple web APIs. Thus, we compare WAR with four baseline approaches:

- **Random**: This approach randomly selects a set of nodes from G that collectively cover the query keywords, and then finds a minimum spanning tree to connect the selected nodes with the fewest nodes among all spanning trees.
- **Greedy**: This approach randomly selects a set of nodes from G that collectively cover the query keywords. It then takes those nodes as the initial root nodes and continuously grows the trees until the selected nodes are interconnected. While the trees grow, a greedy heuristic is applied so that the neighbor containing the most query keywords is selected first.
- **SSR [12]:** Given the input text description and the required number of web APIs N for a web app, this approach first finds N sets of web APIs of N different categories that are most similar to the text description. Then, it selects all the solutions with the highest score calculated based on the web API similarity, popularity and correlation degree.
- **SPR_CR [13]:** This approach employs a methodology similar to SSR. There are three differences. The first is that SPR_CR mainly focuses on the popularity of web APIs. Second, it determines whether two web APIs would be recommended based on their co-occurances, which may lead to redundant APIs. Finally, it returns only one solution.

The experiments were conducted on a machine with 2.60 GHz CPU and 8.0 GB RAM. The software configuration environment is: Windows 10 and Python 3.6. Each experiment was repeated 50 times and their average experiment results were adopted. The source code and dataset used in the experiments are available at https://github.com/qlyseven/source-code.

5.2 Experiment Results

Concretely, six profiles are tested and compared to validate the feasibility of our proposal.

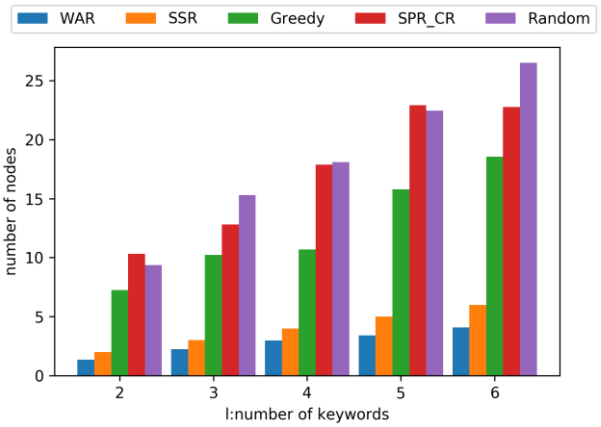
Profile-1: Number of web APIs returned by three approaches.

In this profile, we compare the average number of nodes (i.e., web APIs) in the solutions found by the five approaches for each set of query keywords. The number of query keywords, i.e., l, is varied from 2 to 6. The results are shown in Fig.6.

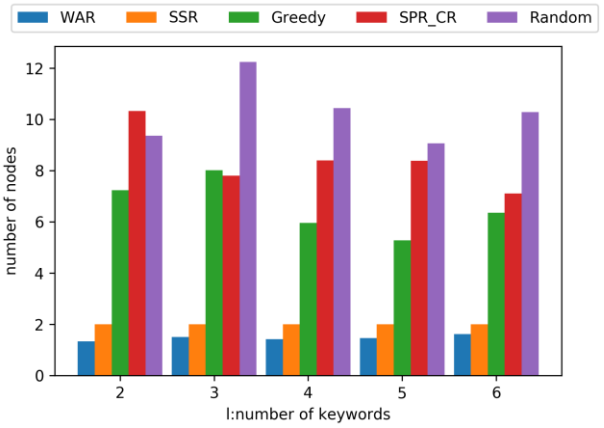
Fig.6(a) compares the number of web APIs in the solutions found by the five approaches in experiment set A. It shows that the solutions returned by the Random, Greedy and SPR_CR approaches consist of more nodes than those by the WAR and SSR approaches. This is because (1) when the Random and Greedy approaches grow a tree, a random or greedy strategy is applied so that they are often trapped within local optimua; (2) the SPR_CR approach may generate redundant web APIs. Therefore, the number of returned nodes is often large. WAR can find solutions with the fewest nodes, which is usually preferable. This indicates the ability of WAR to find the optimal app solutions.

Fig.6(b) shows the results in experiment set B where similar results can be observed as in Fig.6(a). Fig.6(b) shows the ability of WAR to recommend light-weight app solutions when app developers' requirements for their apps are uncertain. Overall, the results presented in Fig.6 show that WAR can find light-weight app solutions without having to include unnecessary nodes. This is very important because otherwise app developers need to spend a lot time on pruning unnecessary nodes in the app solutions. This may violate the connectedness of the remaining nodes in the app solution. In most cases, app developers will find app solutions that contain too many unnecessary nodes useless.

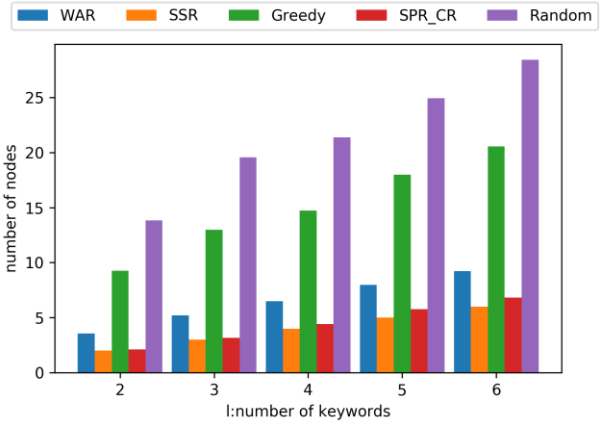
In Fig.6(c), as l increases, the sizes of the solutions found by the WAR, Greedy and Random approaches increase compared to Fig.6(a) and Fig.6(b). The reason is that the keywords are randomly selected from G. Thus, a large number of bridging nodes are needed to construct a connected tree. In addition, the SPR_CR and SSR approaches return smaller solutions than the WAR, Greedy and Random approaches. The reason is that when the keywords are randomly selected from G, the probability of finding a set of co-occurrent APIs that collectively cover the keywords declines. This decreases the number of returned nodes as well as the success rate and connected rate (see Fig.7 and Fig.8).



(a) experiment set A



(b) experiment set B

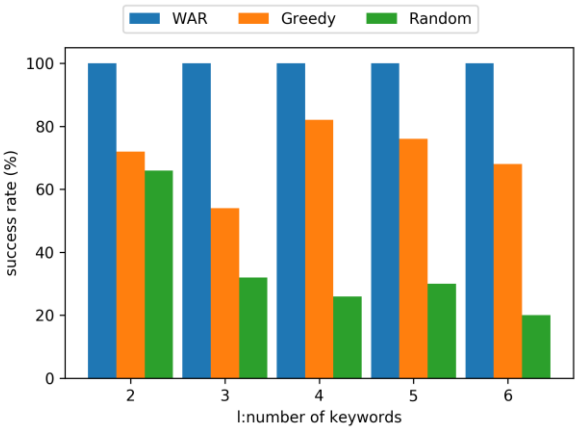


(c) experiment set C

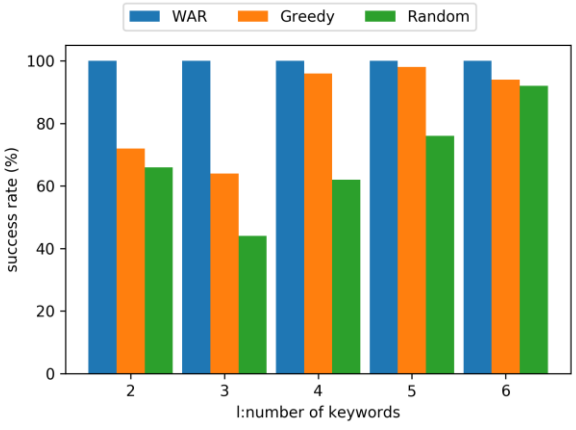
Figure 6. Number of web APIs in the solutions found by five approaches

Profile-2: Success rate of three approaches.

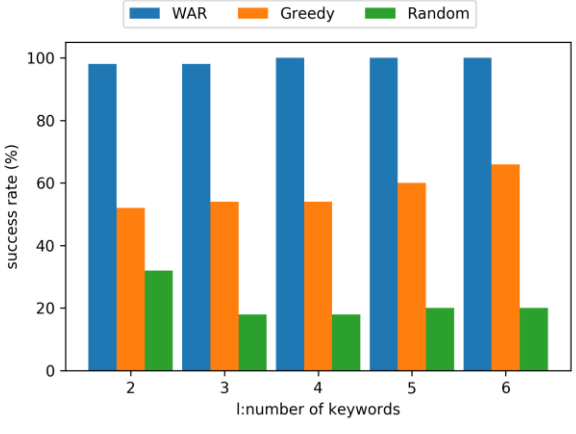
In this profile, we compare the success rates of three recommendation approaches. Here, a web API recommendation solution is successful iff 1) the web APIs in the returned solution are compatible; 2) the number of APIs in the returned solution is not larger than twice the number of query keywords, i.e., $2l$. The value of l is varied from 2 to 6 here. The success rate is defined in this manner because an app developer usually does not want an app solution that is overly sophisticated with too many



(a) experiment set A

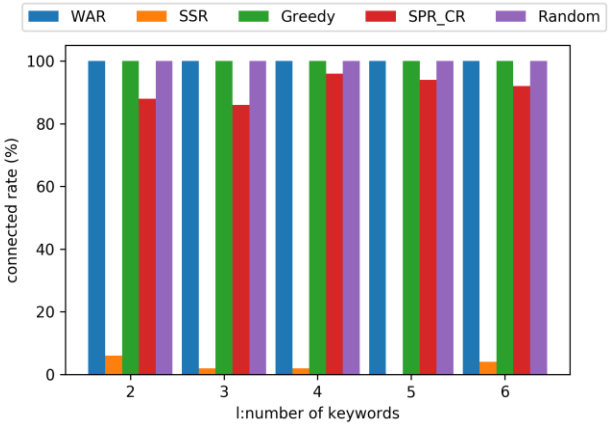


(b) experiment set B



(c) experiment set C

Figure 7. Success rate of finding solutions



(a) experiment set A

(b) experiment set B

(c) experiment set C

Figure 8. Connected rate of finding solutions.

web APIs. This might not be exactly what every app developer wants. However, it allows us to compare the abilities of WAR with the Greedy and Random approaches to find an app solution in a reasonable manner. Please note that according to the definition of success rate in this profile, the success rates of the SPR_CR and SSR approaches are zero because 1) they do not consider the web APIs compatibility; and 2) their solutions often contain more than 2l nodes. Thus, their success rates are not included in Fig. 7.

Fig.7(a) shows the success rates of the three approaches in finding solutions for the 6,146 keyword queries in experiment set A. We can see that WAR can answer all the queries regardless of the number of query keywords with a consistent success rate of 100%. Compared with WAR, it is much more difficult for the Random and Greedy approaches to find a successful solution, especially when l is large. This is indicated by their significantly lower success rates in the experiments, 26%, 30% and 20% for Random and 82%, 76% and 68% for Greedy in the ex-

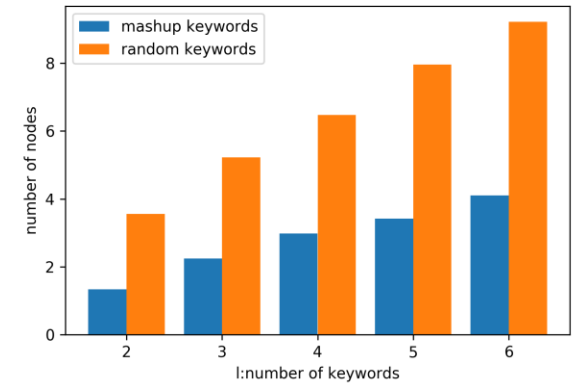
periments where l = 4, 5 and 6 respectively. Fig.7(b) shows the results in experiment set B. Again, WAR manages to obtain a 100% success rate consistently across all different cases in the experiments, versus 77% and 96% obtained by the Random and Greedy approaches on average. Similar results are shown in Fig.7(c). This demonstrates the ability of our proposed WAR to identify necessary bridging web APIs to find a complete app solution. The results presented in this profile demonstrate the usefulness of WAR - it can be used for app developers to find web APIs from G for building any of the 6,146 web apps in the PW dataset.

Profile-3: Connected rate of five approaches.

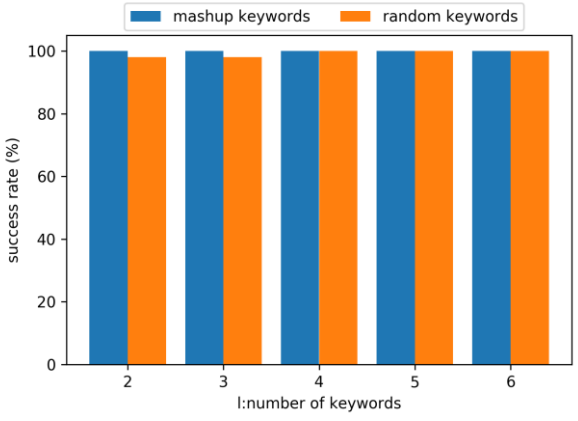
The connected rate (%) of different approaches are presented in Fig.8 with l increasing from 2 to 6. As Fig.8(a) shows, the connected rates of the WAR, Greedy and Random approaches are always 100% as these three approaches grow trees which are always connected. SPR_CR considers web APIs' functionalities, popularity and co-occurrences simultaneougly. As a result, its connected rate is lower than 100% in most cases. SSR takes APIs correlation degree as its major consideration and does not consider web APIs' compatibility directly. Thus, its connected rate is very low. Similar results are shown in Fig.8(b). In Fig.8(c), the keywords are randomly selected from G. Therefore, for the SPR_CR and SSR approaches, the probability of finding a set of co-occurrent APIs collectively covering the keywords declines. As a result, the connected rates decrease accordingly. The results presented in this profile demonstrate that the WAR approach can always guarantee the compability of the returned web APIs.

Profile-4: Number of web APIs in solutions found by WAR and corresponding success rate.

The usefulness of WAR is not limited to only those 6,146 web applications. To demonstrate this, we have also selected random keywords to generate keyword queries to find out whether WAR can handle other queries. The results are presented in Fig.9 with l increasing from 2 to 6. As demonstrated in Fig.9(a), the app solutions to queries with random keywords contain more web APIs than those to queries generated based on the web applications in the PW dataset. The web applications in the PW datasets are real-world web applications. The web APIs used in each of those web applications are closely related to each other. They belong to the same domain or similar domains. Thus, they are close to each other in G. On the contrary, when random keywords are selected to generate queries, their relevance are not ensured. Thus, the web APIs that contain those keywords can be far away from each other in G. As a result, more bridging nodes are needed to connect the corresponding keyword nodes in G. This leads to more web APIs in the app solutions compared to the solutions to the queries generated based on the web applications in the PW dataset, as demonstrated in Fig.9(a). Fig.9(b) shows the success rate achieved by WAR in response to those two different types of queries. Not surprisingly, WAR achieves a consistent success rate



(a) number of nodes



(b) success rate

Figure 9. Number of web APIs in solutions found by WAR in response to queries with random keywords and corresponding success rate.

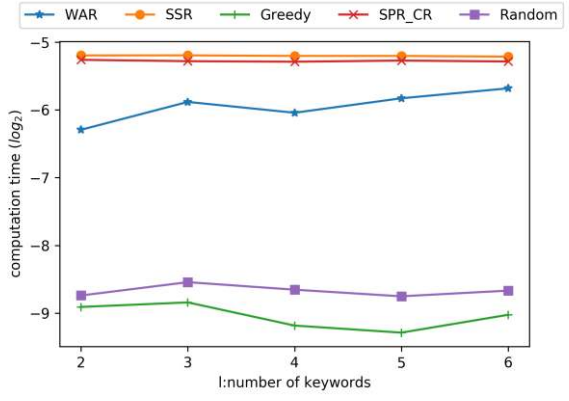
of 100%. This indicates that WAR can always connect all the keyword nodes in G to find an app solution despite the number of necessary bridging nodes, zero or many.

Profile-5: Computation time of five approaches.

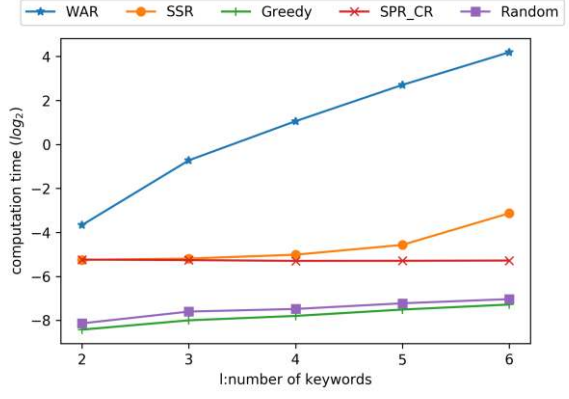
In this profile, we compare the efficiency of the three approaches, measured by the computation times taken to find app solutions in response to app developers' keyword queries.

Fig.10(a) shows the results of experiment set A. As demonstrated, the time cost of SPR_CR stays approximately stable with the growth in l as SPR_CR is not correlated with l. The time cost of SSR increases linearly with l as the candidate web APIs need to be divided into l categories according to their functionalities in the first step of SSR; the Random and Greedy approaches are much faster than WAR, taking significantly less time to find an app solution. When l increases, all WAR, Greedy and Random approaches take more time to find an app solution. The increase in the computation of WAR is more significant than the Random and Greedy approaches. This confirms with the complexity analysis at the end of Section 4.1. When l increases, the number of possible solutions increases exponentially. Finding the optimal solution with the fewest web APIs from such tremendous web APIs is extremely time consuming. Thus, WAR takes a lot of time to respond to app developers' queries when l is large.

(a) experiment set A

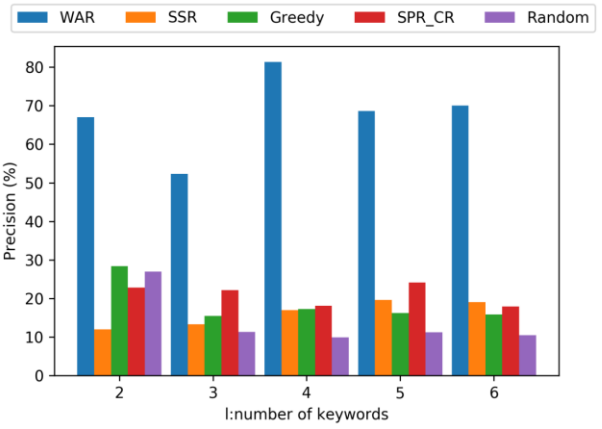


(b) experiment set B

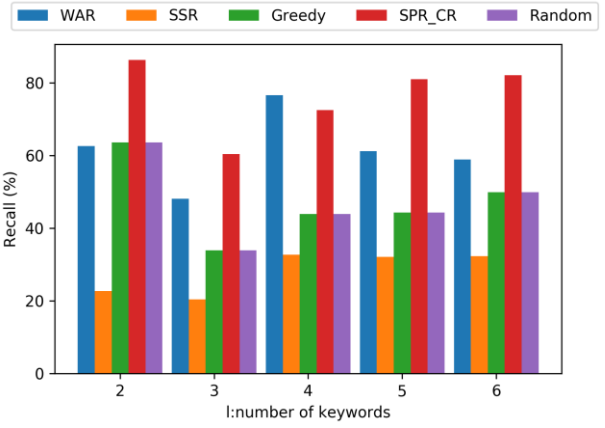


(c) experiment set C

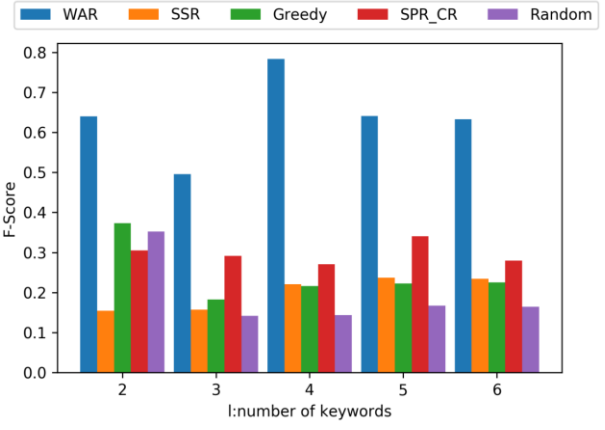
Figure 10. Computation time of five approaches



(a) Precision



(b) Recall



(c) F-score

Figure 11. Other performances of five approaches

In Fig.10(b), only two keywords (i.e., the first keyword and last keyword) are entered regardless of l. Therefore, the time costs of the Greedy, Random, SSR and SPR_CR approaches stay stable as l increases. The computational time of WAR increases when l increases as more bridging nodes are needed to connect the keyword nodes.

The results in Fig.10(c) show that WAR takes more time to find a solution when the query keywords are randomly selected. The reason is the same as why WAR needs to find more web APIs to constitute an app solution, which is discussed above with respect to Fig.10(a) and Fig.10(b). Compared to the query keywords in experiment sets A and B, randomly-selected keywords are far away from each other in G. Thus, more bridging nodes are needed to connect the keyword nodes. Accordingly, the number of candidate solutions increases, which requires WAR to evaluate more candidate solutions to find the optimal one. This is not a significant issue because in real-world cases, app developers would not need to build apps that require completely irrelevant web APIs, e.g., a hotel booking web API and an image processing web API.

Profile-6: Other performance comparisons of five approaches.

In this profile, we evaluate and compare the five ap-

proaches by the metrics of Precision, Recall and F-score. The experimental results are obtained from experiment set A, where l varies from 2 to 6 and 50 real-world web apps are randomly selected from the PW dataset for experiment. The results are shown in Fig. 11.

Fig.11(a) indicates that WAR outperforms the other four approaches in precision. This is because WAR can guarantee the minimum number of nodes in the solution ("true positive" nodes and "false positive" nodes) when a query is executed, which leads to a higher recommendation precision. In Fig. 11(b), the recall value of WAR is worse than that of SPR_CR because SPR_CR often returns more nodes than WAR and hence often a higher recall value. However, the recall value of WAR is higher than those of SSR, Greedy and Random. Given the precision and recall values, the F-score values achieved by the five approaches are presented in Fig. 11(c). It shows that WAR outperforms other four approaches in F-score. This indicates that WAR achievces the highest overall performance.

6 Related Work

Some recent work has advanced the research on web API correlation and laid the groundwork for WAR. Zhong et al. proposed a method in [14] that constructs profiles for web APIs based on historical information about how web APIs are used in web apps. Gu et al. proposed in [15] a method that discovers the correlation among web APIs. Through discourse analysis of web apps' functional specifications, the semantic correlation between web APIs can be learned. A method named Targeted Reconstructing Service Descriptions (TRSD) is proposed in [16] that discovers information about possible application scenarios for web APIs for enriching the description of web APIs. In [17], Cao et al. clustered web APIs into different categories to discover the correlations. Based on the clustering results, similar web APIs can be recommended with collaborative filtering techniques. Liang et al. [18] also attempted to enrich the description of web APIs. They applied a Random Walk with Restart model to find keywords from structural and semantic information on web APIs for describing them.

In recent years, many researchers have investigated the recommendation of web APIs based on the correlation between web APIs. To name a few, Huang et al. proposed a network model that describes the interactions among web APIs [19]. They employed a link prediction method based on rank aggregation to predict the evolution of the interactions over time. This way, web APIs are recommended in pairs and their compatibility is ensured. The major limitation of this method is that web APIs can only be recommended in pairs. The authors later improved their method so that web APIs can be recommended in groups if they constitute a community [20]. Each web API clique is considered a group of web APIs that are potentially useful together. However, this recommendation method is static because it is entirely based on the topological features of the web API network. App developers cannot search for multiple web APIs that specifically and collectively fulfill their objectives. Zhong et al. proposed in [21] a method that exploits latent Dirichlet allocation to extract web API evolution patterns over time. Then, following a simple "the more popular, the better" rule, popular web APIs in popular domains are recommended. To find web APIs that are not popular but potentially useful for a given partial web app, Rahman et al. proposed a recommendation method based on matrix factorization in [22]. This method as well as the one proposed in [21] suffers from the same major limitation as the one proposed in [20]. App developers can only receive recommendations passively. They cannot search for web APIs that they need for building their own web apps.

The authors of [23] also proposed a correlation graph for describing web APIs and their correlations, where the compatibility between two web APIs is evaluated through matching the tag-based semantics of their input and output. Then, given a set of web APIs preselected by an app developer, web APIs that might be of the app developer's interest are recommended to the app developer. The major limitations of this approach are twofold. First, their correlation graph is not entirely reliable because tag-based semantics can be easily misinterpreted. Suppose two web APIs api_1 and api_2, api_1 for booking drive test appointments and api_2 for querying sunset time. When a drive test appointment is booked, api_1 generates a location and a time as output. To query the sunset time, api_2 requires a location and a time. Accordingly, it can be concluded that api_1 and api_2 are compatible and must be linked in the correlation graph. However, in real-world applications, the edge between api_1 and api_2 is useless because it is meaningless to integrate api_1 and api_2. A correlation graph built based on such semantic information will include many such useless edges that significantly impact the usefulness of the answers to keyword queries. The second major limitation of their method is that app developers are navigated through web APIs based on their preselected web APIs step by step. It focuses on the prediction of web APIs that might be of app developers' interest rather than finding web APIs that app developers actually need.

In [24], the authors assume there are a group of functional-qualified but quality-varied candidate services for each task requested by a user. A quality-optimal service composition solution is then produced when a user requires multiple tasks to execute his/her business application while guaranteeing the privacy-preservation of service quality. This work focuses more on composite service quality optimization with privacy-preservation instead of functional-qualified and compatible web APIs recommendations. In [25], the authors propose a novel approach called combinatorial auction for service selection (CASS) to support effective and efficient service composition by considering both the complementarities between services and the competition among service providers. However, this approach does not consider the compatibility between different web services or APIs. In [26], the composability of two services is evaluated by the matching degree between the inputs and outputs of the two services. Concretely, if service A's outputs match the inputs of service B, the two services are regarded as composable and compatible. However, similar to [23], this I/O matching-based service compatibility measurement approach may include many useless edges in the service

correlation graph and impact the usefulness of the answers to keyword queries.

In [12], an approach named SSR is proposed. Given the input text description and the required number of web APIs N for a web app, this approach first finds N sets of web APIs of N different categories that are most similar to the text description. Then, it selects all the solutions with the highest score calculated based on the web API similarity, popularity and correlation degree. In [13], an approach named SPR_CR is proposed. This approach employs a methodology similar to SSR. There are three differences. The first is that SPR_CR mainly focuses on the popularity of web APIs. Second, it determines whether two web APIs would be recommended based on their co-occurances. Finally, it returns only one solution. However, APIs compatibility is not the major focus of these two approaches; as a result, the success rates and connected rates are not high, which have already been validated by the experiment results in Profile 2 and Profile 3.

Mining valuable and useful information from online software repositories has become a popular approach for increasing the productivity of software development [27]. In recent years, it has been a very active and attractive field of research on software engineering since the appearance of centralized online software repositories, such as app stores [28], SourceForge, GitHub and Google Code [29]. Online repositories of web APIs, such as programmable.com and mashape.com, contain a large amount of data on SBSs and the web APIs used by those SBSs. Those data can be mined to uncover useful and helpful information about the interactivity between web APIs [30]. Based on information mined from online software repositories, we proposed a web API correlation graph where web APIs are modeled as nodes and their compatibility as edges. Based on this correlation model, WAR employs the keyword search technique to allow app developers to search for web APIs for their apps by entering only a few keywords that represent the required app tasks. WAR addresses the abovementioned issues and offers a novel data-driven approach for efficiently building apps.

7 Conclusions

In this paper, we propose WAR, a novel approach that integrates and automates the app planning, web API discovery and web API selection operations for building mobile and web apps based on extensive data mined from online software repositories. It assists app developers without detailed knowledge of web APIs in finding app solutions with only a few keywords that describe the required app tasks. WAR offers a new data-driven approach for building apps and can significantly save the time and effort during the process for building apps. The results of experiments on 18,478 real-world web APIs and 6,146 real-world web apps demonstrate the usefulness and efficiency of WAR.

In our future work, we will enhance WAR to increase the diversity of the multiple solutions to offer app developers with higher flexibility in building their apps. Keyword search is still an open research topic. We will follow up the advances in keyword search techniques and enhance WAR accordingly. In real-world applications, app developers might have other objectives, e.g., quality-of-service [31] and privacy [32, 33], other than minimizing the number of web APIs in the solution. We will investigate the approaches for answer queries with such optimization objectives. We will also enhance WAR with automatic query expansion techniques [7] to handle synonymy, word inflections and polysemy.

Acknowledgment

This paper is partially supported by the National Key Research and Development Program of China (No. 2017YFB1400600), Natural Science Foundation of China (No. 61872219, 61672276), the Natural Science Foundation of Shandong Province (ZR2019MF001), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University.

References

[1] M. B. Blake, and M. E. Nowlan. Knowledge discovery in services (kds): Aggregating software services to discover enterprise mashups. *IEEE Transactions on Knowledge and Data Engineering*, 23(6): 889-901, 2011.

[2] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas. Finding authorities and hubs from link structures on the world wide web. *Proc. of the 10th International Conference on World Wide Web (WWW 2001)*, Hong Kong, China, pp. 415-429, 2001.

[3] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B. J. P. Hsu, and K. Wang. An overview of microsoft academic service (mas) and applications. *Proc. of the 24th International Conference on World Wide Web (WWW 2015)*, Florence, Italy, pp. 243-246, 2015.

[4] S. Bergamaschi, E. Domnori, F. Guerra, R. T. Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. *Proc. of 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 2011)*, Athens, Greece, pp. 565-576, 2011.

[5] V. Hristidis and Y. Papakonstantinou. DISCOVERY: Keyword search in relational databases. *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, China, pp. 670-681, 2002.

[6] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong. Exact top-k nearest keyword search in large networks. *Proc. of 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*, Melbourne, Australia, pp. 393-404, 2015.

[7] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1): 1-50, 2012.

[8] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner tree problem*. vol. 53, Elsevier, 1992.

[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in data-

bases using BANKS. *Proc. of the 18th International Conference on Data Engineering (ICDE 2002)*, San Jose, CA, USA, pp. 431-440, 2002.

[10] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. *Proc. of IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, Istanbul, Turkey, pp. 836-845, 2007.

[11] Cormen, T.H., Introduction to Algorithms. MIT Press, 2009.

[12] W. Gao, and J. Wu. A novel framework for service set recommendation in mashup creation. *Proc. of IEEE International Conference on Web Services (ICWS 2017)*, Honolulu, USA, pp. 65-72, 2017.

[13] Q. Gu, J. Cao, and Q. Peng. Service package recommendation for mashup creation via mashup textual description mining. *Proc. of IEEE International Conference on Web Services (ICWS 2016)*, Francisco, USA, pp. 452-459, 2016.

[14] Y. Zhong, Y. Fan, W. Tan, and J. Zhang. Web service recommendation with reconstructed profile from mashup descriptions. *IEEE Transactions on Automation Science and Engineering*, 2016. DOI: 10.1109/TASE.2016.2624310.

[15] Q. Gu, J. Cao, and Q. Peng. Service package recommendation for mashup creation via mashup textual description mining. *Proc. of IEEE 23rd International Conference on Web Services (ICWS 2016)*, San Francisco, USA, pp. 452-459, 2016.

[16] Y. Hao, Y. Fan, W. Tan, and J. Zhang. Service recommendation based on targeted reconstruction of service descriptions. *Proc. of IEEE 24th International Conference on Web Services (ICWS 2017)*, Hawaii, USA, pp. 285-292, 2017.

[17] B. Cao, X. Liu, M. M. Rahman, B. Li, J. Liu, M. Tang. Integrated content and network-based service clustering and web APIs recommendation for mashup development. *IEEE Transactions on Services Computing*, 2017. DOI: 10.1109/TSC.2017.2686390.

[18] T. Liang, L, Chen, J. Wu, A, Bouguettaya, Exploiting Heterogeneous Information for Tag Recommendation in API Management, *Proc. of 23rd International Conference on Web Services (ICWS 2016)*, San Francisco, USA, pp. 436-443, 2016.

[19] K. Huang, Y. Fan, W. Tan, and X. Li. Service recommendation in an evolving ecosystem: a link prediction approach. *Proc. of 20th International Conference on Web Services (ICWS 2013)*, Santa Clara, USA, pp. 507-514, 2013.

[20] K. Huang, Y. Fan, and W. Tan. Recommendation in an evolving service ecosystem based on network prediction. *IEEE Transactions on Automation Science and Engineering*, 11(3): 906-920, 2014.

[21] Y. Zhong, Y. Fan, K. Huang, W. Tan, and J. Zhang. Time-aware service recommendation for mashup creation. *IEEE Transactions on Services Computing*, 8(3): 356-368, 2015.

[22] M. M. Rahman, X. Liu, and B. Cao. Web API recommendation for mashup development using matrix factorization on integrated content and network-based service clustering. *Proc. of IEEE International Conference on Services Computing (SCC 2017)*, Hawaii, USA, pp. 225-232, 2017.

[23] G. Huang, Y. Ma, X. Liu, Y. Luo, X. Lu, and M. Brian Blake. Model-based automated navigation and composition of complex service mashups. *IEEE Transactions on Services Computing*, 8(3): 494-506, 2015.

[24] W. Dou, X. Zhang, J. Liu, and J. Chen. HireSome-II: towards privacy-aware cross-cloud service composition for big data applications. *IEEE Transactions on Parallel and Distributed Systems*, 26(2): 455-466, 2015.

[25] Q. He, J. Yan, H. Jin, and Y. Yang. Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. *IEEE Transactions on Software Engineering*, 40 (2): 192-215, 2014.

[26] N. Chen, N. Cardozo, and S. Clarke. Goal-driven service composition in mobile and pervasive computing. *IEEE Transactions on Services Computing*, 11(1): 49-62, 2018.

[27] H. Kagdi, M.L. Collard, J.I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2): 77-131, 2007.

[28] W. Martin, F. Sarro, Y. Jia, Y. Zhang, M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 43(9): 817-847, 2017.

[29] R. Dyer, H.A. Nguyen, H. Rajan, T.N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology*, 25(1): 7, 2015.

[30] L. Qi, Q. He, F. Chen, W. Dou, S. Wan, X. Zhang, X. Xu. Finding all you need: web apis recommendation in web of things through keywords search. *IEEE Transactions on Computational Social Systems*, 2019. (DOI: 10.1109/TCSS.2019.2906925).

[31] Q. He, G. Cui, X. Zhang, F. Chen, S. Deng, H. Jin, Y. Yang. A game-theoretical approach for user allocation in edge computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 2019. DOI: 10.1109/TPDS.2019.2938944, 2019.

[32] W. Gong, L. Qi, Y. Xu. Privacy-aware multidimensional mobile service quality prediction and recommendation in distributed fog environment. *Wireless Communications and Mobile Computing*, vol. 2018, Article ID 3075849, 8 pages, 2018.

[33] L. Qi, X. Zhang, W. Dou, Q. Ni. A distributed locality-sensitive hashing-based approach for cloud service recommendation from multi-source data. *IEEE Journal on Selected Areas in Communications*, 35(11): 2616-2624, 2017.

Lianyong Qi received his PhD degree in Department of Computer Science and Technology from Nanjing University, China, in 2011. In 2010, he visited the Department of Information and Communication Technology, Swinburne University of Technology, Australia. He is currently a full professor of Qufu Normal University of China. His research interests include recommender systems and services computing.

Qiang He received his first Ph. D. degree from Swinburne University of Technology (SUT), Australia, in 2009 and his second Ph. D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2010. He is a senior lecturer at Swinburne University of Technology.

Feifei Chen received her PhD degree from Swinburne University of Technology, Australia in 2015. She is currently a lecturer at Deakin University. Her research interests include software engineering, cloud computing and green computing.

Xuyun Zhang is a senior lecturer in the Department of Computing at Macquarie University, Australia. Prior to his current appointment, he worked as a lecturer at the University of Auckland, New Zealand, and a postdoctoral fellow in the Machine Learning Research Group of NICTA (currently Data61, CSIRO) in Australia. He received his PhD degree from University of Technology, Sydney (UTS, Australia) in 2014, as well as his ME and BS degrees in Computer Science from Nanjing University (China) in 2011 and 2008, respectively. His primary research interests include big data, cloud computing, data privacy & security, and Web service technology.

Wanchun Dou received his PhD degree in Mechanical and Electronic Engineering from Nanjing University of Science and Technology, China, in 2001. From Apr. 2001 to Dec. 2002, he did his postdoctoral research in the Department of Computer Science and Technology, Nanjing University, China. Now, he is a full professor of the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include workflow, cloud computing, big data and service computing.

Qiang Ni (M'04–SM'08) received the B.Sc., M.Sc., and Ph.D. degrees from the Huazhong University of Science and Technology, China, all in engineering. He is a Professor and the Head of Communication Systems Research Group, School of Computing and Communications, Lancaster University, InfoLab21, Lancaster, U.K. His research interests include future generation communications and networking systems, including green communications and networking, cloud systems, cognitive radio network systems, heterogeneous networks, 5G, SDN, IoTs, big data analytics and vehicular networks in which areas he had already published more than 180 papers. He is a Voting Member of IEEE 1932.1 standard. He was an IEEE 802.11 Wireless Standard Working Group Voting member and a Contributor to the IEEE Wireless Standards.