The Journal of Systems and Software 165 (2020) 110569

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Substructure similarity search for engineering service-based systems

Jintao Wu[a,b], Xing Guo[a,*], Guijun Yang[b,*], Shuhui Wu[c], Jianguo Wu[a]

[a] School of Computer Science and Technology, Anhui University, 111 Jiulong Road, Shushan District, Hefei 230601, China
[b] Key Laboratory of Quantitative Remote Sensing in Agriculture of Ministry of Agriculture, Beijing Research Center for Information Technology in Agriculture, Beijing 100097, China
[c] School of Management, Hefei University of Technology, 193 Tunxi Road, Shushan District, Hefei 230009, China

ARTICLE INFO

Article history:
Received 15 May 2019
Revised 15 February 2020
Accepted 4 March 2020
Available online 5 March 2020

Keywords:
Service-based system
Substructure similarity search
Service composition
Web service
Graph matching

ABSTRACT

With the broad application of service-oriented architecture in service-oriented software engineering, service-based systems (SBSs) are becoming ever more widely used. As a result, the selection of appropriate component services destined to fulfill the functional requirements becomes a critical challenge for successfully building SBSs, especially when the pre-specified SBS plan involves a complicated structure. Because building an exact SBS is often too restrictive, a similarity search for complex functional requirements becomes an essential operation that must be efficiently supported. We thus investigate in this work the substructure similarity search problem of building a SBS. To solve this new research problem, we propose a feature-based method, called the substructure similarity search for service-based systems (5S), to help users find similar SBS solutions by progressively relaxing a SBS plan. The 5S approach models each SBS as a set of features and transforms the task of relaxation of a SBS into the maximum allowed missing features, which can filter many SBSs directly without costly structure comparisons. 5S thus opens a new paradigm for efficient SBS engineering that shortens the build cycle. Finally, we discuss a series of experiments using real-world Web service datasets that demonstrate the effectiveness and efficiency of the proposed approach.

1. Introduction

Advances in service-oriented architecture (SOA) and microservices architecture (MSA) have provided a promising solution to engineer service-based systems (SBSs) by discovering and composing loosely coupled Web services provided by different platforms (Ardagna and Pernici, 2007; Liangzhao et al., 2004). In such a SBS, the component services are executed by a system engine (e.g., a BPEL engine (Baresi and Guinea, 2011)) and together offer the functionality of a SBS hosted on cloud platforms and delivered as software as a service (SaaS). Recently, a large number of service repository platforms have appeared, such as ProgrammableWeb[1] and Mashape,[2] has further fueled the rapid growth of web and cloud services. More and more companies such as Google, Facebook, and Amazon have encapsulated some of their capabilities as Web services and published them on service repository platforms. Users can search and invoke these services, and compose them to design SBSs according to their own requirements. The popularity of Web services and SOA enables various SBSs to be engineered to fulfill the increasingly sophisticated needs of various business organizations (Shang et al., 2010).

The service-composition process for engineering a SBS usually involves four phases, as shown in Fig. 1. These phases are *system planning, service discovery, service selection*, and *service delivery* (He et al., 2014; He et al., 2017). In the system planning phase, the user determines the tasks to be performed and the order in which the tasks are executed. Most techniques used in this phase are based on artificial intelligence (AI) techniques (Hoffmann et al., 2007; Pistore et al., 2005; Oh et al., 2008; Zou et al., 2014; Deng et al., 2013). Next, in the *service discovery* phase, the user identifies a set of candidate services that provide the required functionalities for each of the tasks based on the functional and semantic information about candidate services (Cassar et al., 2014; Wu et al., 2016; Blake and Nowlan, 2011; Klusch et al., 2009; Brogi et al., 2008). In the *service selection* phase, the user selects a particular service from each set of candidate services to satisfy the multi-dimensional quality constraints and achieve the given opti-

* Corresponding authors.
E-mail addresses: ah_wjt@foxmail.com (J. Wu), guoxingahu@qq.com (X. Guo), yanggj@nercita.org.cn (G. Yang), 2018010083@mail.hfut.edu.cn (S. Wu), wjg5408@163.com (J. Wu).

[1] http://www.programmableweb.com/.
[2] https://www.mashape.com/.

https://doi.org/10.1016/j.jss.2020.110569
0164-1212/

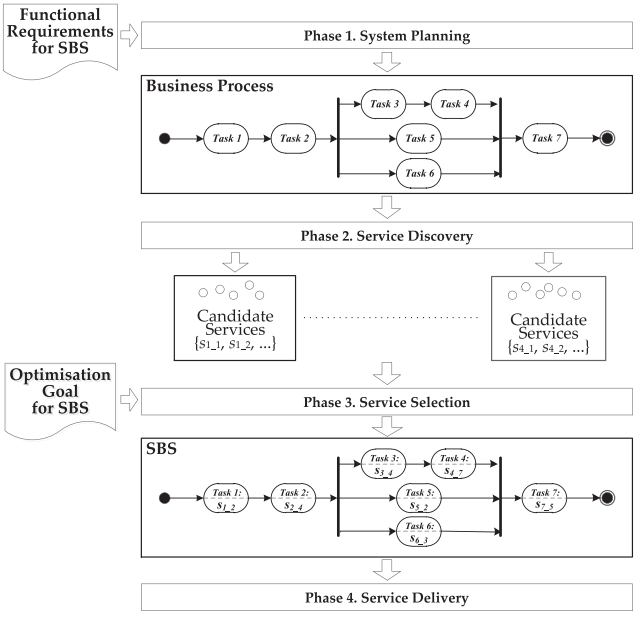


Fig. 1. Process in service-oriented software engineering.

mization goal. This problem is, in fact, an NP-hard quality-aware service-selection problem (He et al., 2014; Calinescu et al., 2011). Finally, in the *service delivery* phase, the selected services are executed in the desired order on a system engine to realize the SBS. In this phase, how to deploy the component services of a SBS becomes a critical issue, which involves load balancing (Bramson et al., 2010), resource management and allocation (Wang et al., 2012; Xu et al., 2014), and fault tolerance issues (Zheng and Lyu, 2015). Through the above phases, which combine several services, the SBS created can provide multiple functionalities to satisfy complex requests from users.

Although significant research has focused on solving the problems in each phase individually, building a SBS remains a highly complex task. In particular, for a pre-specified SBS plan with a complicated structure (multiple tasks, complex execution order), selecting appropriate Web services from the numerous candidates to build a SBS that meets the complex structural requirements of the user is often too restrictive and demands excessive effort, thereby becoming a significant obstacle to the further extensions of SOA applications. In extreme cases, no composed service can meet the exact needs of all component functions in a SBS plan. This situation calls for a subsequent query refinement process to find the SBSs of interest. Unfortunately, it is often too complicated and time-consuming for a non-expert to do manual refinements, which would excessively extend the build cycle in SBS engineering. Thus, an automated approach to fine tune the expected SBS and return possible SBS solutions is demanded by users for building SBSs.

A straightforward and promising approach for simplifying this problem is to ask the search system to find SBSs in the repository of previously created SBSs that almost match the expected SBS (i.e., a similarity search). Such an approach would be advantageous because it assists users to build SBSs rather than building them directly through system planning, service discovery, and service selection. This approach was inspired by the rapidly growing number of SBSs created, most of which follow some popular business models and usage patterns (Vollino and Becker, 2013). This type of similarity search is also beneficial within the development process. For example, in the process of engineering a SBS, the user may not know the exact composition desired for the full SBS, but requires that it contain a set of small functional fragments. In addition, this similarity-search strategy is appealing because users can first define the portion of the SBS plan for exact matching and let the system slightly alter the remaining portion of the SBS plan. In other words, if no matching SBS solution is available in the SBS repository, the user can delete or relabel a small number of tasks based on the SBS plan to satisfy approximately the user's functional requirements. Similarity search techniques have long been popularized by Web search engines such as Google and Bing to mine information from Web documents, and they have also been widely used in many other fields (Yan et al., 2005; Willett et al., 1998; Shasha et al., 2002). However, none of the existing similarity search techniques can be directly applied to effectively find multiple Web services for engineering SBSs.

In this paper, we propose to use a substructure similarity search for service-based systems (5S), which is a substructure similarity search mechanism that assists users engineering SBSs that meet or approximately meet user needs and require minimal computation time. The 5S search runs on a SBS repository, where each element in the SBS repository is actually a valid service composition implemented by a series of service invocations collected by data mining (Zhu et al., 2012; Subashini and Kavitha, 2011; Klein et al., 2014) or using semantics (Bonatti and Festa, 2005; Yu et al., 2007; Zeng et al., 2003). Given a pre-specified SBS plan that describes the functionality of the SBS, the 5S approach first searches Web services based on the complete structure of the SBS plan. If no match is found for the given SBS plan, then 5S relaxes the SBS plan by deleting or relabeling one or more tasks and then searching the Web services based on the relaxed SBS plan. Thus, 5S can find solutions that contain an SBS plan that approximates the plan desired by users. All of the operations in 5S are feature-based and, because of its efficiency, 5S requires relatively little time to build SBS or return a set of similar SBS solutions for users, which shortens the SBS build cycle.

To summarize, this research makes the following major contributions:

- This is the first attempt to model the critical problem of substructure similarity searching for engineering SBSs. We also propose 5S, which is an efficient approach to solve this problem.
- The 5S models each SBS as a set of features and transforms the task deletions into feature misses. With an upper bound on the maximum allowed feature misses, 5S can directly filter many SBSs without performing pairwise similarity computation.
- We show that using either too many or too few features does not improve the filtering performance because of the *high fragment difference* and *full fragment difference* phenomena identified herein. Therefore, 5S employs a multi-filter composition strategy in which the filters are constructed by a covering clustering algorithm that groups features with similar filtering power into a feature set. This clustering method does not require the number of clusters to be pre-specified or the initial centers to be manually selected.
- We conducted extensive experiments to evaluate the effectiveness and efficiency of the proposed approaches applied to a dataset that contains the functional information about 1496 real-world Web services and 2926 SBSs crawled from programmableweb.com.

The rest of the paper is organized as follows: The next section analyzes the requirements by using an example. Section 3 defines the preliminary concepts. Section 4 discusses the technical details of 5S. Section 5 discusses the evaluation experiments and analyzes the results. Section 6 discusses some parameter settings and comparison results. Section 7 reviews related work. Finally, Section 8 concludes the paper and gives the direction of future work.

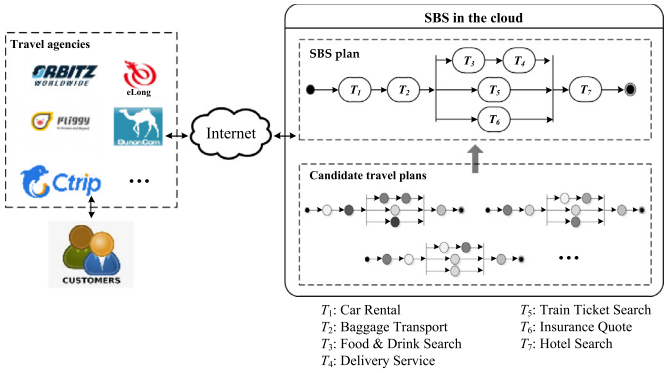


Fig. 2. Travel Booking SBS.

2. Motivating scenarios

This section presents an example to illustrate the motivation and requirements of this research. Fig. 2 shows an example of a cloud-based SBS that provides a travel booking service, which is represented as a business process that includes seven tasks: T_1, Car Rental; T_2, Baggage Transport; T_3, Food & Drink Search; T_4, Delivery Service; T_5, Train Ticket Search; T_6, Insurance Quote; T_7, Hotel Search. This SBS serves travel agencies by processing their user requests. The user enters their travel requirements (e.g., city of departure, destination, departure date, return date, preferred type of rental car, etc.), and the SBS returns a list of candidate travel plans (service composition solutions) for the user to book.

However, due to the numerous uncertain factors (e.g., techniques used to implement the service, network condition, consumer preferences, etc.), finding the exact SBS solution is often too restrictive. In other words, a service composition solution may not exist for a given SBS plan. For example, consider a user that is using the travel booking service shown in Fig. 2: He enters his travel requirements (i.e., city of departure, Beijing; destination, Shanghai; departure date, 22:30–23:30; requires food & drink service). However, trains with food & drink service may not be available on the departure date. Therefore, the cloud-based SBS has no solution to return to the user, which constitutes an anomaly. If this anomaly cannot be fixed in a timely manner, users with the same travel requirements will suffer from the inaccessibility of the travel service or may change their travel plans, which can easily lead to customer attrition and customer complaints. A possible solution is to rebuild the entire SBS plan and then execute a new search for the desired service; however, such an approach is potentially time consuming and therefore not practical.

A promising approach to address these issues is to ask the cloud-based SBS to find service composition solutions that nearly match the SBS plan. For example, if the consumer relaxes the SBS plan with one task miss, Fig. 3(a) would be a good solution (i.e., replace T_5: Train Ticket Search in the SBS plan with T_8: Airline Ticket Search). If the consumer further relaxes the SBS plan with two tasks misses, the SBSs in Figs. 3(a) and 3(b) are good solutions. For further relaxation, the SBS in Fig. 3(c) could also be a solution, although some tasks are omitted (T_3, T_4, T_6) and one task is bound (T_9). But in any case, this similarity-search strategy proposes various similar alternatives when an exact solution is unavailable.

Meanwhile, the similarity search mechanism can also bring the following benefits in the process of engineering a SBS.

(1) Little information is needed to formulate a reasonable query and provides more high-quality alternatives for users. To some extent, the failure of service composition (Vizcarrondo et al., 2017; Laleh et al., 2018) due to the unavailability of a single point of service is avoided (Deng et al., 2013).

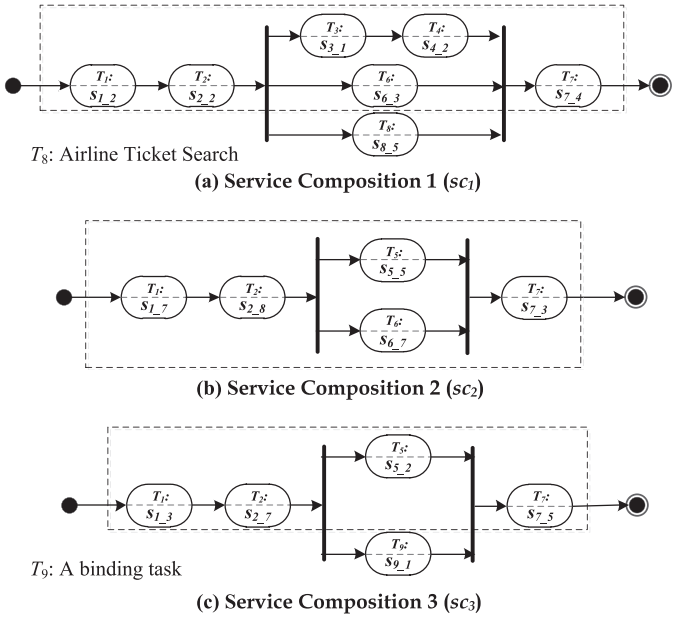


Fig. 3. SBS alternatives.

(2) Many implementations of similarity methods are computationally inexpensive, so searching a large SBS repository can be routinely performed. Compared with traditional service composition methods, such as (Zou et al., 2014; Jatoth et al., 2019), its efficiency will be greatly reduced with the increase of tasks.

(3) This reduces the conditions for building SBS, because even relatively simple tools, such as Oracle BPEL Process Manager and IBM Process Designer, still require system engineers to have a detailed knowledge of SOA techniques, which is often too much to ask (Ardagna and Pernici, 2007).

Based on the above analysis, we conclude that providing approximate match results rather than the exact match result can be beneficial. Unfortunately, solving the similarity-search problem in SBS engineering remains challenging because the computation of the pairwise similarity is very expensive for a large-scale SBS repository. A naive solution is to form a set of SBSs with one or more task deletions and then use the exact search. However, this approach does not work for more than one deletion. For example, if three tasks are deleted in a SBS plan with 15 tasks, it may generate 455 relaxed SBS plans, which would be too time-consuming to search. Therefore, a better solution is required.

3. Problem statement

The motivating scenario discussed above reveals the problem inherent in doing a substructure similarity search in SBS engineering, which is vital for SBS engineering. In this section, we explain the basic concepts and definitions required for the solution. Furthermore, based on the motivating example, we articulate the key concepts and problem description.

Definition 1 Web Service**.** A Web service is defined as a four-tuple $s = \langle i, f, p, QoS \rangle$, where

(1) i is the unique service identifier;
(2) f is the functional description of the service, including the input, output, pre-condition, and result of the service;
(3) p is the platform where the service is deployed;

(4) $QoS = \{q_1, q_2, ...\}$ is the set of quality values of s obtained from its service level agreements (SLAs); e.g., cost, response time, reliability, throughput, etc.

Definition 2 SBS Plan**.** The SBS plan can be formalized as a two-tuple $Q = \langle T, A\rangle$, where

(1) $T = \{T_1, T_2, ..., T_n\}$ is a set of tasks with each task T_i corresponding to a component functionality in the SBS engineering process and the integer n being the number of tasks required to build a specific SBS;
(2) $A = \{a(T_i, T_j)|T_i \in T, T_j \in T\}$ is a set of dependencies between tasks in T, where $a(T_i, T_j)$ means that the inputs of T_j depend on the outputs of T_i (i.e., T_j can be the succeeding task of T_i) and A is used to describe the structure of the SBS plan.

The task set of the SBS plan Q is denoted $T(Q)$ and the dependency set is denoted $A(Q)$. The size of a Q is defined by its number of tasks, denoted $|T(Q)|$.

Definition 3 SBS alternative**.** The SBS alternative is actually a service composition for a SBS that can be formalized as a triplet $sc = \langle T, S, A\rangle$, where

(1) $T = \{T_1, T_2, ..., T_n\}$ is a set of tasks, with each task T_i corresponding to a component functionality in the SBS engineering process and the integer n being the number of tasks required to build a specific SBS;
(2) $S = \{s_1, s_2, ... s_n\}$ is the set of Web services constituting the service composition; Each single service s_i corresponds to a component task T_i in the SBS engineering process;
(3) A provides the structural information of the SBS, which is the same as the SBS plan.

In fact, each SBS plan or alternative corresponds to a composite task or service graph, which reflects the order in which operations are invoked, including sequential and parallel tasks between two operations.

Definition 4 SBS Repository**.** A SBS repository, denoted $HL = \{sc_1, sc_2, ...\}$, where each $sc_i \in HL$ is a SBS alternative, which is a potential solution for a SBS.

Here, the SBSs in *HL* can be collected either by data mining (Zhu et al., 2012; Subashini and Kavitha, 2011; Klein et al., 2014) or by using semantics (Bonatti and Festa, 2005; Yu et al., 2007; Zeng et al., 2003). The data-mining-based approaches mine the available SBSs from the historical usage information of the SBS. The semantics-based approaches discover the available SBSs by mining semantic associations and interactions between services according to well-defined ontologies. Both types of approaches construct the SBS repository offline. Once completed, the SBS repository remains relatively stable and can be updated with minimum overhead for specific events (e.g., new SBSs joining or old ones leaving).

Given a SBS repository *HL* and a SBS plan *Q*, we may not find a SBS (or we may find only a few SBSs) in the *HL* that contains the entire *Q*. Thus, it would be of interest to find SBS solutions that almost match the SBS plan *Q*, which constitutes a SBS substructure similarity search problem.

Before solving this problem, an important issue to resolve is how to measure the SBS substructure similarity between a SBS solution *sc* and a SBS plan *Q*. A straightforward approach is to use the similarity between graphs to simulate the similarity between SBSs. Several similarity definitions of graphs are available in the literature; for example, Refs. (Shang et al., 2010; Zhu et al., 2012; Peng et al., 2015; Yuan et al., 2015) use the MCS-based graph similarity measure to calculate subgraph distances and thereby indirectly measure the similarity between graphs. Reference (Yan et al., 2005) uses the percentage of the maximum retained edges in the subgraph as a similarity measure. Inspired by this, we use herein the percentage of the maximum common SBS to measure the similarity between the SBSs, as defined below.

Definition 5 SBS Substructure Similarity**.** Given a SBS alternative *sc* and a SBS plan *Q*, if *CP* is the maximum common SBS of *sc* and *Q*, then the substructure similarity between *sc* and *Q* is defined by $|A(CP)|/|A(Q)|$.

For example, consider the candidate SBS solution in Fig. 3(a) and the SBS plan in Fig. 2. Their maximum common SBS (dashed box portion) has six dependencies (edges): $\{a(T_1, T_2), a(T_2, T_3), a(T_2, T_6), a(T_3, T_4), a(T_4, T_7), a(T_6, T_7)\}$. The SBS plan has 8 dependencies (edges): $\{a(T_1, T_2), a(T_2, T_3), a(T_2, T_5), a(T_2, T_6), a(T_3, T_4), a(T_4, T_7), a(T_5, T_7), a(T_6, T_7)\}$. Thus, the SBS substructure similarity between these two SBSs is about 75% with respect to the SBS plan *Q*. This also means that, if we relax the SBS plan *Q* with one task miss, Fig. 3(a) contains the relaxed *Q*. We can also compute the similarity of SBSs in Fig. 3(b) and 3(c) with the SBS plan *Q*; we obtain 62.5% and 37.5%, respectively.

Based on the above description, the SBS substructure similarity search problem is now to find SBSs as similar as possible in *HL* according to the SBS plan with user requirements. Note also that all SBSs discussed herein can be represented by directed, vertex-labeled graphs. Therefore, we translate the SBS substructure similarity search problem into a graph matching problem, which can be broadly classified into two groups: exact matching and approximate matching. The former is used to find the structurally identical subgraphs that exactly match a specific query (Berretti et al., 2001), which is often costly to compute. The latter is used to find subgraphs that approximately match a specific query within a degree of deviation (Yan et al., 2005; Raymond et al., 2002). These types of queries are beneficial within their own applications. For example, Grafil is an approximate match algorithm proposed in Ref. (Yan et al., 2005) to solve the substructure similarity search problem for large-scale graph databases. The successful transformation of the structure-based similarity measure into a feature-based measure renders the Grafil algorithm attractive in terms of accuracy and efficiency. In the present work, we exploit the advantages of Grafil algorithm and propose a novel approach 5S to solve the SBS substructure similarity search problem and help users to build SBSs. In the next section, we will present the technical details of our 5S approach.

4. Substructure similarity search in service-based system engineering

Our 5S approach is designed as a three-phase process, as shown in Fig. 4. Phase 1 extracts the features of SBSs in the *HL*. Next, Phase 2 constructs the *SBS-fragment table index* based on the features extracted in Phase 1. Finally, Phase 3, drops the SBS solutions that are not similar to *Q* based on the *SBS-fragment table index*. Sections 4.1–4.3 present the details of these phases, respectively.

4.1. Phase 1: feature extraction

To improve the efficiency of the substructure similarity search in SBS engineering, 5S adopts the feature-based measure to model each SBS plan as a set of features and thereby transforms task deletions into feature misses in the SBS plan. In other words, the similarity between a SBS plan and a SBS alternative can be measured by the number of features they have in common. Based on this, many SBS alternatives with low similarity can be directly filtered without performing pairwise similarity computation. The features discussed here could be task execution paths, discriminative frequent SBSs (i.e., frequent service composition solutions), or any SBS indexed in *HL* (i.e., SBS alternatives).

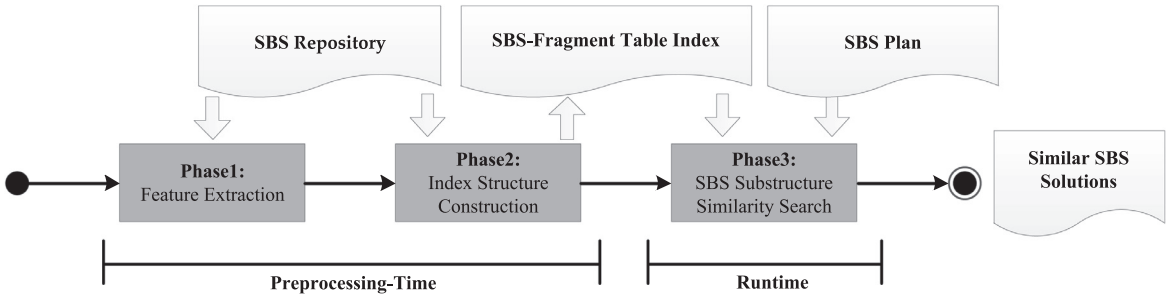


Fig. 4. Process for 5S.

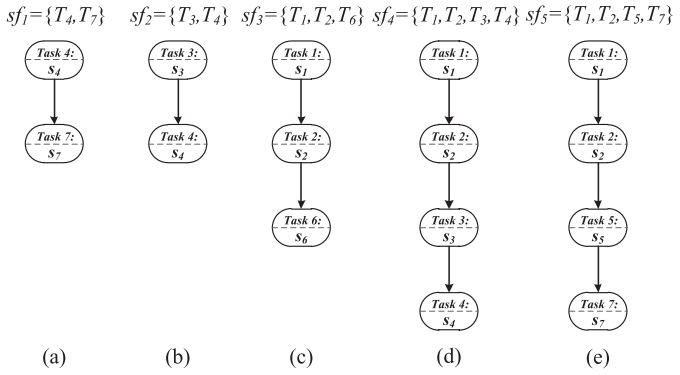


Fig. 5. SBS fragments.

Clearly, the extracted features should be as rich as possible, and these features should be uniformly representative for a large number of SBSs. On the one hand, if only a small number of features are extracted, some SBSs may not be represented by any feature; for example, if the total extracted features F contains only f_1–f_3. Therefore, it is highly probable that neither the SBS solution *sc* nor the SBS plan Q contains any features in F, in which case we cannot use the feature-based measure to determine whether *sc* is similar to Q. On the other hand, if the extracted feature is not representative, only a small number of SBSs contain this feature, so it is useless for most SBSs. For example, in a *HL*, the vast majority of SBSs contain fewer than five tasks. If the extracted feature contains ten tasks, then this feature is clearly meaningless for a large number of SBSs.

Without loss of generality, we use the task execution serial path as a feature in this research. In order for the number of features not to increase exponentially, the number of tasks in the path is at most equal to a predefined constant λ. We omit the case of a single task as a feature because of the lack of structural information. Therefore, the edge is the smallest unit of the feature (i.e., $\lambda = 2$). The size of λ can be set according to the size of the *HL* and the size of the SBS solution in the *HL*. We also study the impact of λ on the results in Section 6.1 from the perspective of similarity. In the present research, the task execution serial path of limited length is called a SBS fragment, and is defined as follows:

Definition 6 SBS Fragment (*sf*)**.** Given a SBS solution *sc*, the SBS fragment is actually a subgraph of the *sc*, which can also be denoted as a triplet $sf = \langle T, S, A\rangle$ and satisfies $T \subseteq sc.T$, $S \subseteq sc.S$, $A \subseteq sc.A$.

All *sfs* extracted from *HL* are denoted by F. For example, Fig. 5 depicts the F extracted from *HL* as shown in Fig. 3 when $\lambda = 4$. Note are other *sfs* are present in Fig. 3; however, we omit them and use only sf_1–sf_5 as F in the remainder of this paper.

The problem of extracting a SBS fragment from *HL* can be transformed into a subgraph isomorphism problem in a directed graph, which can be solved by numerous methods (Shang et al., 2008; He and Singh, 2008; Zhang et al., 2009; Zhao and Han, 2010), which we do not discuss in detail here because it is beyond the scope of this work.

Based on the discussion above, multiple *sfs* can be extracted from one SBS, in other words, the SBS can be represented by these *sfs*. Therefore, we can build a mapping from the SBS to the *sf* set. Details of this step are presented in the next section.

Table 1
SBS-Fragment Table Index.

Service composition	*sf* set
sc_1	$\{sf_1, sf_2, sf_3, sf_4\}$
sc_2	$\{sf_3, sf_5\}$
sc_3	$\{sf_5\}$

4.2. Phase 2: index structure construction

We now introduce the *SBS-fragment table*, which is an index structure designed to facilitate the SBS substructure similarity search. Each record of the table corresponds to a SBS in the *HL* and is represented by a set of *sfs*.

Table 1 shows the *SBS-fragment table index* built for the *HL* in Fig. 3 and the total *sfs* shown in Fig. 5. For instance, sc_2 corresponds to the *sf* set $\{sf_3, sf_5\}$, which indicates that the SBS sc_2 can be represented by the two SBS fragments sf_3 and sf_5. The *SBS-fragment table index* is built offline and is easily maintained: adding a new SBS to the *HL* requires adding only a single additional record; that is, the mapping from SBS to the *sf* set.

Given the *SBS-fragment table*, we can apply the feature-based measure to any SBS plan Q against a candidate SBS solution in the *HL* by using any subset of the indexed *sfs*. Consider the SBS plan Q shown in Fig. 2 with one task relaxation. According to the *SBS-fragment table* in Table 1, even if we do not know the specific execution process of sc_3, we can discard sc_3 immediately based on the *sfs* included in sc_3, because sc_3 has only one of the SBS fragments sf_1–sf_5. This filtering process is not involved with any costly SBS structure similarity checking. The only computation required is to retrieve the *sf* set of the Q and calculate the number of missing *sfs*. Since the filtering mechanism is entirely built from the *SBS-fragment table index*, we need not access the physical SBS repository unless we want to accurately compute the similarity between SBSs.

4.3. Phase 3: SBS substructure similarity search

In this section, we first discuss how to filter SBS solutions that are not similar to a relaxed SBS plan Q using basic filtering strategy. We then describe how the size of the *sf* set of Q affects the filtering performance and propose an adaptive multi-filter composition strategy.

Table 2
Task-fragment matrix.

	sf_1	sf_2	sf_3	sf_4	sf_5
T_3	0	1	0	1	0
T_4	1	1	0	1	0
T_5	0	0	0	0	1
T_6	0	0	1	0	0

4.3.1. Basic filtering strategy

Let us first check an example. Fig. 2 shows a SBS plan Q and Fig. 5 depicts all *sf*s extracted from the *HL* shown in Fig. 3. Assume that a user finds no match for this Q in the *HL*. In this case, the user may relax one task (T_3 or T_4) via a deletion or relabeling operation. The user may deliberately retain the middle task (T_2 or T_5), the beginning task (T_1), and the ending task (T_7), because deleting the middle service may break the relaxed Q into pieces, and deleting the beginning task or the ending task may eliminate the user's functional requirements, which should certainly be avoided. Since the relaxation can take place among T_3 and T_4, we are not sure which *sf* will be affected by this. However, regardless of which task is deleted or relaxation relabeled, the relaxed Q should have at least two occurrences of these *sf*s. Equivalently, we say that the relaxed Q may miss at most three occurrences of these *sf*s in comparison with the original Q. By using this information, we can discard SBSs that contain fewer than two occurrences of these *sf*s. We name this similarity search method the "feature-based SBS substructure similarity search," which includes the two operations *sf* Miss-estimation and *fragment difference* Calculation, respectively. The details are given below.

A. *sf* miss-estimation

For a SBS plan Q, we use the *task-fragment matrix* to build a map between tasks and *sf*s (the task discussed here is the component task of Q). For convenience, we represent the set of *sf*s contained in Q as a *filter*. We first assume that only one *filter* exists, and all *sf*s contained in the Q are in this *filter* (we remove this assumption later in this section). Each row of the *task-fragment matrix* represents a task, and each column represents a *sf*. Table 2 shows the matrix built for the SBS plan Q in Fig. 2 and the *sf*s shown in Fig. 5. All *sf*s representing the Q are recorded in the *task-fragment matrix*, where each entry indicates whether a specific task is a meta-task for a specific *sf*. For example, sf_2 contains at least two tasks: T_3 and T_4. We say that task T_i is a *hit* with a SBS fragment sf_j if sf_j contains T_i. Building a *task-fragment matrix* is not expensive because each Q contains only a small number of *sf*s.

Consider a SBS plan Q: if k tasks can be deleted, what is the maximal number of *sf*s that can be missed? We call this problem the "*sf* miss-estimation problem." In fact, it is the maximum number of *sf*s that can be *hit* by k tasks in the *task-fragment matrix*. This is a classical maximal coverage problem, which has been shown to be NP-complete. The optimal solution that finds the maximal number of *sf* misses can be approximated by the greedy algorithm outlined in Algorithm 1.

Algorithm 1 GreedyCover.

Input: Task-fragment Matrix Ma, Maximum task relaxations k.
Output: The number of *sf* misses N_{greedy}.
1: let $N_{greedy} = 0$;
2: **for each** $l = 1 \ldots k$ **do**
3: select row r_l that maximizes $|Ma(r_l, \cdot)|$;
4: $N_{greedy} = N_{greedy} + |Ma(r_l, \cdot)|$;
5: **for each** column c **do**
6: **if** $Ma(r_l, c) = 1$ **do**
7: set $Ma(\cdot, c) = 0$;
8: **return** N_{greedy};

where $Ma(r, c)$ is the entry in row r_{th}, column c_{th} of the *task-fragment matrix* Ma. The r_{th} row vector and the c_{th} column vector of the matrix Ma are denoted $Ma(r,\cdot)$ and $Ma(\cdot,c)$, respectively, and $|Ma(r,\cdot)|$ represents the number of nonzero entries in $Ma(r,\cdot)$.

The greedy algorithm first selects a task that *hits* the largest number of *sf*s (line 3), and then the *sf*s covering it (lines 5–7). This selection is repeated until k tasks are considered (line 2). The number of *sf*s calculated by this greedy algorithm provides an estimate of the upper bound of *sf* misses.

Algorithm 1 contains two loops, one nested in the other. Let n be the number of columns in matrix Ma. The integer k denotes the maximum number of task relaxations. The time complexity of Algorithm 1 is $O(kn)$.

Theorem 1. *Let N_{greedy} and N_{opt} be the total sf misses computed by the greedy solution and by the optimal solution. We have*

$$N_{opt} \leq 1.6N_{greedy}. \tag{1}$$

Proof. Let k be the number of task relaxations. The first task picked by Algorithm 1 *hits* at least N_{opt}/k *sf*s. Therefore, the number of *sf*s of N_{opt} we still have to *hit* after the first task is picked is $N_1 \leq N_{opt} - N_{opt}/k$. □

We are now left with N_1 *sf*s that we have to *hit*. At least one of the remaining tasks T_i must *hit* at least $N_1/(k-1)$ of these *sf*s because the optimum solution would otherwise have less than N_{opt} *sf*s. After Algorithm 1 picks the task that *hits* the largest number of *sf*s that have not been *hit*, it is left with $N_2 \leq N_1 - N_1/(k-1)$ *sf*s that have not been *hit*. Notice that $N_2 \leq N_1 - N_1/(k-1) \leq N_1(1-1/k) \leq N_{opt}(1-1/k)^2$. In general, we then have

$$N_k \leq N_k(1-1/k) \leq N_{opt}(1-1/k)^k, \tag{2}$$

where $N_k = N_{opt} - N_{greedy}$ is the number of *sf*s that have not been *hit* after selecting k tasks by Algorithm 1. Thus, we also have

$$N_{opt} - N_{greedy} \leq N_{opt}(1-1/k)^k \leq N_{opt}/e. \tag{3}$$

We rewrite the inequality (3) as

$$\begin{aligned} N_{opt} - N_{greedy} &\leq N_{opt}/e \\ N_{opt} &\leq (1/(1-1/e))N_{greedy} \\ N_{opt} &\leq 1.6N_{greedy}. \end{aligned}$$

This theoretical result shows that the optimal solution cannot be approximated in polynomial time within a ratio of $e/(e-1)$–$O(1)$ unless $P = NP$ (Feige, 1998).

Based on Theorem 1, the number of columns removed by Algorithm 1 provides a way to estimate the upper bound of *sf* misses. In addition, N_{opt} should be less than or equal to the $k \times \max_r |Ma(r,\cdot)|$, as shown in Eq. (4), where $\max_r |Ma(r, \cdot)|$ is the maximum number of *sf*s of a service *hit*:

$$N_{opt} \leq k \times \max_r |Ma(r, \cdot)|. \tag{4}$$

Based on Eqs. (1) and (4) we conclude that

$$N_{opt} \leq \min\{k \times \max_r |Ma(r, \cdot)|, 1.6N_{greedy}\}. \tag{5}$$

For convenience, we use N_{apx} to denote $\min\{k \times \max_r |Ma(r, \cdot)|, 1.6N_{greedy}\}$.

A tighter bound often leads to a smaller set of SBS solutions, while a looser bound results in a larger number of results that contain many SBSs with low similarity. Thus, a tight bound of N_{opt} is critical to the filtering performance during Phase 3 of the 5S. To obtain a candidate solution set with a higher average similarity, the upper bound of *sf* misses obtained by Algorithm 1 needs to be further tightened.

Any optimal solution that leads to N_{opt} should encompass the following two cases: (Case 1) r_l is selected in this solution; or (Case

2) r_l is not selected. We call these two cases the *selection step* and *disqualifying step*, respectively. Let $N_{opt}(Ma, k)$ be the optimal value of the maximum *sf* misses for k task relaxations. Suppose row r_l maximizes $|Ma(r_l,\cdot)|$. Let Ma' be Ma except for $Ma'(r_l,\cdot) = 0$ and $Ma'(\cdot, c) = 0$ for any column c that is *hit* by r_l, and let Ma'' be Ma except for $Ma''(r_l,\cdot) = 0$. In the selection step, the optimal solution should also be the optimal solution of the remaining matrix Ma'. Therefore, $N_{opt}(Ma, k) = |Ma(r_l,\cdot)| + N_{opt}(Ma', k-1)$, where $k-1$ means that the remaining $k-1$ rows need to be selected from Ma' because row r_l is selected. In the disqualifying step, the optimal solution for Ma should be the optimal solution for Ma''. That is, $N_{opt}(Ma, k) = N_{opt}(Ma'', k)$, where k means that we still need to select k rows from Ma'' because row r_l is not selected. Since N_{opt} is the maximum number of columns *hit* by k tasks, N_{opt} should be equal to the maximum value returned by these two steps. Therefore, we have

$$N_{opt}(Ma, k) = \max \begin{cases} |Ma(r_l, \cdot)| + N_{opt}(Ma', k-1) \\ N_{opt}(Ma'', k). \end{cases} \tag{6}$$

Eq. (6) is a recursive solution for calculating N_{opt} and is equivalent to enumerating all the possible combinations of k rows in the *task-fragment matrix*, which may be very time-consuming. Each Ma' (or Ma'') derived from the original matrix Ma after several recursions is actually a matrix of interleaving multiple selection steps and multiple disqualifying steps. Thus we can control the depth of recursion by limiting the number of instances of interleaving. Suppose Ma' has h selected rows and b disqualified rows, then h is limited to less than H, b is limited to less than B, and $H + B$ should be less than m, where H and B are predefined constants, H is less than the maximum number k of task relaxations, and m is the number of rows in the *task-fragment matrix*. In this way, the recursion depth can be controlled to reduce the space-time overhead. This is a classical branch-and-bound method.

Based on the discussion above, we can estimate the upper bound of *sf* misses by the branch-and-bound algorithm and the greedy algorithm. The detailed process is shown in Algorithm 2.

Algorithm 2 is initialized by $N_{est}(Ma, k, 0, 0)$. The condition in line 1 will terminate the recursion when it selects H rows or when it disqualifies B rows. Line 3 selects row r_l, which maximizes $|Ma(r_l,\cdot)|$, where Ma is the *task-fragment matrix*. Lines 5 and 6 set the new matrix Ma' and matrix Ma''. Line 7 selects row r_l while line 8 disqualifies row r_l. Line 9 calculates the maximum value of the result returned by lines 7 and 8. Meanwhile, we can also use Algorithm 1 to get the upper bound of N_{opt} directly, as done by line 10. The bound obtained by Algorithm 2 is not greater than the bound derived by the greedy algorithm because we intentionally select the smaller bound in lines 10 and 11.

The computational complexity of Algorithm 2 depends on the recursive and non-recursive parts. In the recursive part, the recursion depth is controlled by constants B and H, so the computational complexity of the recursive part is $O(2^v)$ ($v = \min\{H, B\}$). The non-recursive part is a greedy algorithm (Algorithm 1) with a computational complexity of $O((k-H)n)$. Thus, the computational complexity of Algorithm 2 is $O(2^v + (k-H)n) = O(2^v)$.

The result of Algorithm 2 is only the recursive approximation in Eq. (6), but the result satisfies the inequality

$$N_{opt}(Ma, k) \le N_{est}(Ma, k, 0, 0) \le N_{apx}(Ma, k). \tag{7}$$

Given two non-negative integers H and B in Algorithm 2, if $H = k$ and $B = m - k$ then $N_{est}(Ma, k, 0, 0) = N_{opt}(Ma, k)$, and if $H << k$ then $N_{est}(Ma, k, 0, 0)$ is close to $N_{apx}(Ma, k)$.

Algorithm 2 Branch-and-Bound.

Input: *Task-fragment Matrix Ma*, Maximum task relaxations k, h selection steps and b disqualifying steps.
Output: The number of *sf* misses N_{est}.
1: **If** $b \ge B$ or $h \ge H$ **then**
2: **return** $N_{apx}(Ma,k)$;
3: select row r_l that maximizes $|Ma(r_l, \cdot)|$;
4: let $Ma' = Ma$ and $Ma'' = Ma$;
5: set $Ma'(r_l,) = 0$ and $Ma'(\cdot, c) = 0$ for any c if $Ma(r_l, c) = 1$;
6: set $Ma''(r_l, \cdot) = 0$;
7: $N_1 = |Ma(r_l, \cdot)| + Algorithm2(Ma', k, h+1, b)$;
8: $N_2 = Algorithm2\ (Ma'', k, h, b+1)$;
9: $N_a = \max(N_1, N_2)$;
10: $N_b = N_{apx}(Ma, k)$;
11: **return** $\min(N_a, N_b)$;

B. *fragment difference* calculation

Based on the description above, once the maximum number of *sfs* misses is obtained, we can use it to filter SBS solutions. Suppose F is the total *sfs* in *HL* that are extracted in Phase 1 in the 5S method. Consider further a SBS solution *sc* and a SBS plan Q. Let $FS(sc)$ and $FS(Q)$ denote the set of *sfs* contained in *sc* and Q, respectively. There is only one *filter*, so let $filter = FS(Q)$. Clearly, we have $FS(sc) \subset F$ and $filter \subset F$. Compared with *sc*, we want to know the number of extra *sfs* that appear in *filter*. We call the size of these extra *sfs* the *fragment difference*, denoted *FD*. Eq. (8) calculates the *fragment difference* between *sc* and Q in *filter*, where $|filter - FS(sc)|$ is the number of *sf* in the set of extra *sf*. Consider the SBS plan Q shown in Fig. 2. Assume that sf_1–sf_5 contained in Q (sf_1–sf_5 are in Fig. 5) form a *filter* used for filtering, and sc_1 in Table 1 is a candidate SBS solution. In this case, the *fragment difference* between sc_1 and Q is 1 calculated using Eq. (8).

$$FD = |filter - FS(sc_1)|. \tag{8}$$

Assume that the Q can be relaxed with k tasks and that $f_{\max}$ is the maximum allowed *sf* misses. If $FD > f_{\max}$, we conclude that *sc* is not similar to Q within k task relaxations. For example, Table 2 shows the *task-fragment matrix* built for the Q in Fig. 2 and the *sfs* shown in Fig. 5. Assume that the number of task relaxations $k = 1$, then the upper bound of the maximum *sf* misses calculated by using Algorithm 2 is 3 (denoted $f_{\max} = 3$). With this information, we can immediately discard sc_3 in Table 1 because the *frequency difference* between sc_3 and Q is 4 greater than $f_{\max}$. As mentioned above, we need not perform any complicated structure comparison between *sc* and Q. The corresponding *sf* set of a *sc* can be obtained from the *SBS-fragment table index* directly without scanning the *HL*, so the *fragment difference* calculation is actually very fast.

4.3.2. Adaptive multi-filter composition strategy

In the previous section, we explored the basic filtering strategy in the SBS substructure similarity search. However, the abovementioned research does not adequately consider how the size of the *filter* affects the filtering performance. An interesting question is whether, in a single search, the 5S method filters thoroughly if all the *sfs* of a Q are selected in a single *filter* or if only a small number of *sfs* are selected. In fact, our experiments give very different results: Using all the *sfs* together or only a small number of *sfs* in a single search deteriorates the filtering performance rather than improving it. In this section, we explain the mechanism behind this phenomenon and discuss how to solve this problem.

Let $FS(Q) = \{sf_1, sf_2, sf_3, sf_4, sf_5\}$ be all *sfs* of the SBS plan Q, where sf_1–sf_4 all contain a specific task T. In a single search, if we put all the *sfs* contained in Q into a single *filter*, and build the *task-fragment matrix*, we see that task T *hits* four *sfs* (sf_1 – sf_4). If the number of task relaxations $k = 1$, then the upper bound of the maximum *sf* misses calculated by using Algorithm 2 is four, denoted $f_{\max} = 4$. Therefore, the results returned by the 5S method contain these SBS solutions that have only one occurrence of all *sfs*.

In other words, the user may get a lot of results, but a large part of these results are not similar to *Q*. The origin of this phenomenon is that, if most *sfs* cover several common tasks, the relaxation of these tasks will make the maximum allowed *sf* misses too big. As one can see, the 5S method may fail to filter some SBS solutions that are not similar to *Q* if we use all the *sfs* together in a single *filter*. We call this phenomenon the *high fragment difference*.

Conversely, if we only put a small number of *sfs* into a single *filter* and build a *task-fragment matrix*, then the maximum allowed *sf* misses may become very close to the number of *sfs* contained in the *filter*. In that case, the filtering strategy loses its pruning power. Suppose that a *filter* contains only one *sf*. Let $k = 1$, then $f_{\max} = 1$ as calculated by using Algorithm 2. According to the *fragment difference* calculation in the previous section, we see that all SBSs in *HL* are not satisfied $|filter - FS(sc)| > f_{\max}$. That is, *sc* in the case of $FS(Q) \cap FS(sc) = \Phi$ cannot be filtered. In other words, the 5S method returns all SBS solutions in *HL* as results to the user. We call this phenomenon the *full fragment difference*.

The above example implies that deploying all the *sfs* or a small number of *sfs* in a single *filter* may weaken the filtering power. To address this issue, the 5S method uses the covering clustering algorithm that follows the minimum covering principal (Zhang and Zhang, 1999), which we call 5S-Covering. Once covering clustering is done on all *sfs* in *Q*, 5S-Covering adaptively partitions the closest *sfs* into clusters. A separate *filter* is constructed based on each cluster of *sfs*. Here, "closest" means that their filtering powers are the most similar. In this study, the filtering power of *sf* is related to two factors: one is the frequency of *sf* in the *SBS-fragment table* and the other is the size of *sf*. Generally speaking, the filtering power of *sf* is inversely proportional to the former and proportional to the latter. We use *selectivity* (defined below) to measure the filtering power of a *sf* for all SBSs in *HL*.

Definition 7 Selectivity**.** Given a SBS-*fragment table*, the *selectivity* of each SBS fragment *sf* in the table is defined by

$$sel_{sf} = w_1(1 - \frac{fre_{sf}}{|HL|}) + w_2\frac{|sf|}{\lambda} \tag{9}$$

where fre_{sf} is the frequency of occurrence of *sf* in the *SBS-fragment table*, $|Table|$ is the size of the *SBS-fragment table*, $|sf|$ is the SBS fragment size, λ is discussed in Phase 1 to limit the size of the extracted *sf*, and w_1 and w_2 are the influence factors of frequency and fragment size on *selectivity*, respectively In this paper, we set w_1 and w_2 to 0.5 for convenience.

In the description of the covering clustering algorithm, the newly formed cluster at each iteration is referred to as the *current cluster*, which is denoted C_{cr}. Its center, which is denoted c_{cr}, is referred to as the *center* of the *current cluster*, and its radius, which is denoted r_{cr}, is referred to as the *radius of the current cluster*. The set of *sfs* that are not yet covered by any cluster is denoted F_{uc} ($F_{uc} \subset F$), where *F* contains all the *sfs* of *Q*.

1. Identify the centroid of *F*, denoted by$cf = \sum_{sf \in F} sel_{sf}/|F|$.
2. Identify the *sf* in *F* that is closest to *cf* and use it as the center of the first cluster C_1, which is denoted c_{cr} ($c_{cr} = 1$) with $\min|cf - sel_{sf}|$, where $\forall sf \in F$.
3. Calculate r_{cr} by averaging the distances between c_{cr} and the *sfs* in F_{uc}, with $r_{cr} = \sum_{sf \in F_{uc}} |sel_c - sel_{sf}|/|F_{uc}|$.
4. Identify the *sf* in F_{uc} that is furthest from c_{cr} and use it as the new c_{cr} with $\max|sel_c - sel_{sf}|$, where $\forall\ sf \in F_{uc}$.
5. Repeat Steps 3 and 4 until no *sf* can be identified in Step 4, which means that all *sfs* in *F* are covered.

Fig. 6 presents an illustrative example to demonstrate the clustering process of the covering clustering algorithm. To group the *sfs*, the covering clustering algorithm goes through six iterations to identify the six clusters C_1, ..., C_6, with c_1, ..., c_6 as the centers, and r_1, ..., r_6 as the radii, respectively. This clustering algorithm does not require the number of clusters to be pre-specified or the initial centers to be manually selected. At each iteration, it merges the closest *sfs* into a single cluster.

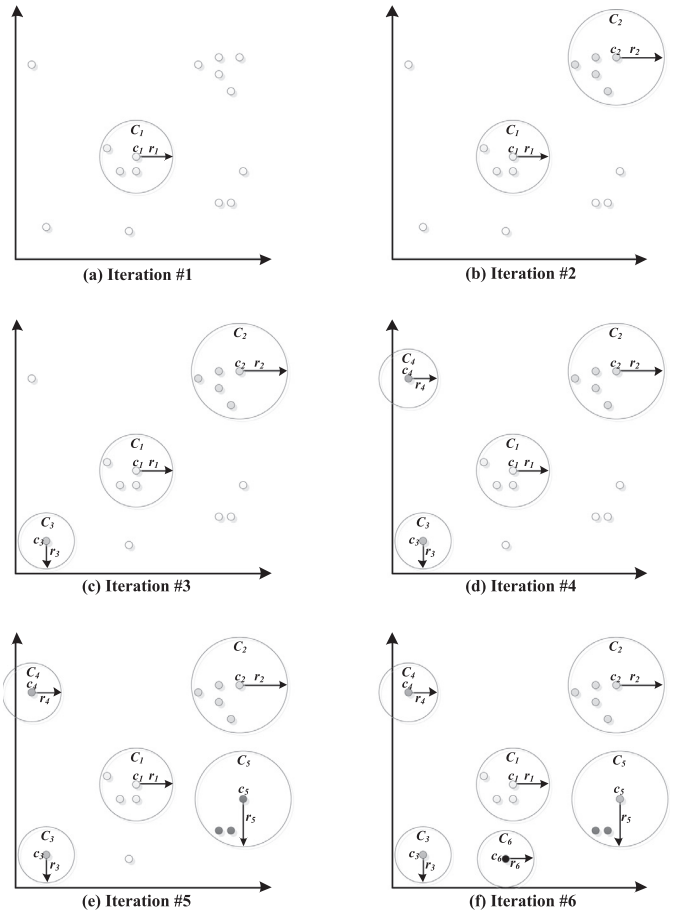


Fig. 6. Clustering example.

The coverage clustering algorithm merges the closest *sfs* into a single cluster that effectively reduces the probability of occurrence of *high fragment difference* but cannot avoid the occurrence of *full fragment difference*. For example, the clusters C_3, C_4, and C_6 in Fig. 6(f) contain only one *sf*. Therefore, we slightly modified the coverage clustering algorithm with the following two improvements:

1. **Merge:** We merge clusters that contain only one *sf* into a single cluster.
2. **Add:** Assume *k* task relaxations. For each $cluster_i$, if the upper bound of the maximum feature misses calculated by Algorithm 2 is $|cluster_i|$, then we add a virtual feature to $cluster_i$ to form a new *filter*. The virtual feature does not appear in the SBS plans *Q* nor in the *sc*.

Clearly, the **Merge** operation effectively reduces the probability of *full fragment difference*. The **Add** operation checks the filtering performance of each *filter*, and improves the filtering performance of the *filter* with no filtering power by adding the virtual feature. For example, suppose that $f_{max} = |cluster_i|$ in filter $cluster_i$, then, all SBSs in *HL* satisfy $|cluster_i - FS(sc)| \le f_{\max}$, so the *filter* is not capable of filtering. After the **Add** operation, the original $cluster_i$ is turned into $Ncluster_i$, and we have $1 \le |Ncluster_i - FS(sc)| \le |cluster_i| + 1$. If $FS(Q) \cap FS(sc) = \Phi$, then $|Ncluster_i - FS(sc)| = |Ncluster_i| > f_{\max}$, so the *sc* is filtered out.

The pseudocode for the improved clustering algorithm above is presented in Algorithm 3. This algorithm is executed on all *sf*s of the SBS plan *Q*.

Algorithm 3 Covering Clustering.

```
    Input: All fragments F of Q.
    Output: A set of clusters C1, C2, ...
 1: let cr = 1;
 2: identify cf
 3: do
 4:     identity c_cr //(1) when cr = 1 and (3) otherwise
 5:     Ccr.center = c_cr
 6:     calculate r_cr //(2)
 7:     C_cr.radius = r_cr
 8:     cr = cr+1
 9: while (F_uc ≠ Ø)
10: Merge
11: Add
12: return clusters;
```

The computational complexity of Algorithm 3 relies on the 5-step procedure and two operations discussed above. The computational complexity of Step 1 is $O(n)$ because there are n *sf*s in *Q*. Similarly, the computational complexity of Steps 2–4 is also $O(n)$. In the worst-case scenario, the computational complexity of the **Merge** operation is $O(n)$. The computational complexity of the **Add** operation is $O(kn)$. Overall, the computational complexity of Algorithm 3 is $O(kn)$.

4.4. Implementation of overall algorithm

Algorithm 4 gives a high-level description of the proposed 5S substructure similarity search algorithm.

Algorithm 4 5S.

```
    Input: SBS repository HL,
    SBS-Fragment Table Index TI,
    SBS plan Q,
    Maximum task relaxations k.
    Output: Candidate solution set R_Q
 1: F = FS(Q);
 2: compute the selectivity based on TI;
 3: C = Covering Clustering (F);
 4: R_Q= HL;
 5: for each filter in C do
 6:     build the task-fragment matrix Ma;
 7:     f_max = Branch-and-Bound (Ma, k, h, b)
 8:     R_Q = {sc|FD = |filter-FS(sc)| ≤ f_max, sc ∈ R_Q};
 9: Return R_Q
```

In line 1, *F* represents the set of SBS fragments extracted from the SBS plan *Q*. Line 2 calculates the selectivity of each SBS fragment in *F* based on the SBS-Fragment Table Index *TI* using Eq. (9). Line 3 uses Algorithm 3 to adaptively divide the nearest SBS fragments into clusters. After line 3, a separate *filter* is constructed based on each cluster of SBS fragments. Next, the basic filtering strategy for each filter is applied, (lines 5–8). The filtering result of the former filter serves as input for the latter filter. The input of the first filter is *HL*, as shown in line 4, where the similar SBS solutions are returned from the last *filter*.

If the user needs more similar SBS solutions than those returned from the 5S method, the SBS plan should be further relaxed and this process repeated. The *SBS-fragment table* in Phase 2 is built beforehand based on the features extracted in Phase 1 and can be used by any SBS plan. Thus, the 5S method is actually very fast.

5. Experimental evaluation

This section evaluates the effectiveness (measured by Success Rate, Number of Candidate Answers, and Average Similarity) and efficiency (measured by computational overhead) of the 5S method for doing a substructure similarity search in SBS engineering.

5.1. Compared approaches

We implemented the three approaches to SBS substructure similarity search mentioned in Section 4:

- **5S-Basic:** This approach uses all the sfs together in a single filter. It gives the baseline for the comparison.
- **5S-Hierarchical:** This approach constructs a multi-filter by using the hierarchical clustering algorithm. In the multi-filter composition phase, 5S-Hierarchical combines the sfs whose sizes differ at most by 1, and groups them by their selectivity. The results of each clustering operation are divided into three categories. See Ref. (Yan et al., 2005) for details on this comparison approach.
- **5S-Covering:** This approach constructs a multi-filter by using the covering clustering algorithm. This clustering method does not require the number of clusters to be pre-specified or the initial centers to be manually selected. In the multi-filter composition phase, 5S-Covering groups by selectivity all the sfs contained in the SBS plan Q.

5.2. Metrics

This section describes the metrics used in the evaluation.

5.2.1. Effectiveness

Given a SBS plan *Q*, the proposed approach aims to find SBS solutions in the SBS repository that meet or approximately meet user needs. Accordingly, we evaluate the effectiveness of the comparison approaches, which are measured by three metrics, Success Rate (SR), Number of Candidate Answers (NCA) and Average Similarity (aveSim).

SR is defined as

$$\mathrm{SR} = 1 - n/N, \tag{10}$$

where N is the number of SBS plans. For each SBS plan *Q*, if the 5S method does not return a solution, then the 5S searching has failed. The integer n is the number of times the 5S searching fails. The quantity SR is the probability that the 5S method finds SBS solutions. *A higher SR indicates a higher probability of finding suitable solutions.*

NCA is defined as

$$\mathrm{NCA} = |R_Q|, \tag{11}$$

where $|R_Q|$ is the number of solutions returned by each approach. Due to the phenomena of *high fragment difference* and *full fragment difference*, the user may get a lot of candidate solutions, but a large part of these results are not similar to *Q* and need to be filtered. NCA measures the filtering performance of each approach. *A low NCA indicates strong filtering power.* aveSim is defined as

$$\mathrm{aveSim} = \frac{\sum_{r \in R_Q} similarity(r, Q)}{|R_Q|}, \tag{12}$$

where the function *similarity* is defined in Definition 5. The quantity aveSim measures the average similarity between the solutions returned by each approach and the *Q*. *A higher aveSim indicates a high similarity with the original Q.*

5.2.2. Efficiency

To evaluate the efficiency of the proposed approaches, we measured the computational overheads of all approaches.

Table 3
PW dataset.

Number of Services Used by SBS	2	3	4	5	6	7	8	9	10	11–15	16–20	20+
Number of SBSs	1490	623	313	195	85	57	34	40	21	49	12	7

5.3. Experiment setup

The experiments were conducted on the PW dataset, which contains the functional information about real-world Web services and SBSs crawled from programmableweb.com, a service portal that has been accumulating a variety of Web services and SBSs since 2005 (Al-Masri and Mahmoud, 2008; Barros and Dumas, 2006). This dataset has also been widely used in many other studies on service-oriented software engineering (He et al., 2017). The PW dataset contains the information about which of the 1496 Web services are used by each of the 2926 SBSs. Table 3 presents the relevant statistics of the PW dataset. We generate a SBS repository *HL* based on the information retrieved from the PW dataset. First, we choose some frequent service sets (frequently used to build SBS) which derives from the PW dataset. Next, each frequent service set is assigned a random structure; we call these structured frequent service sets the "seed pool". Finally, 2926 candidate SBSs in the *HL* are constructed on the basis of the generated seed pool. See Ref. (Kuramochi and Karypis, 2001) for details about the graph (candidate SBS) generator.

To evaluate the proposed approaches we conducted two series of experiments, called series A and series B. In series A, a total of 2926 SBS plans were generated directly from the *HL*, each corresponding to one of the 2926 SBSs in the PW dataset. We first evaluated the exact search ability of the 5S method, which is to find a SBS solution that meets the exact needs of all component functions in a SBS plan. We then evaluated the similarity search ability of the 5S method, which is to find similar solutions that approximately meet user needs. Finally, to evaluate the efficiency of 5S, we measured the computational overhead (i.e., the time required by 5S to process the given SBS plans). Series A demonstrates the practicality and efficiency of 5S for a real-world SBS repository and real user queries.

In series B, the total SBS plans were randomly combined from the generated seed pool, and the size of each SBS plan was determined from a Poisson distribution with mean m. We denote the total SBS plans by Q_m, where m is the average size of the SBS plan in Q_m. In series B we take the value of m to be 5, 10, and 15, and 1000 SBS plans were used for all three sizes. In this series of experiments, we changed the number of task relaxations from 0 to 5 in steps of 1 to compare the SR, NCA, aveSim, and the computational overhead of the results obtained by the different approaches. The random SBS plan used in series B enabled us to evaluate 5S more comprehensively.

All comparison approaches were implemented in Scala by using JDK 1.7. All experiments were conducted on a machine running Ubuntu 14.04 LTS with an Intel i7-4790 CPU running at 3.60 GHz and with 4 GB RAM.

5.4. Evaluation of effectiveness

Fig. 7 demonstrates the SR of 5S for answering the 2926 SBS plans in experiment series A. Note that each SBS plan in this series of experiments corresponds to a real SBS in the PW dataset. The SR discussed here refers to the probability that the results returned by 5S contain an exact SBS solution (i.e., covers all tasks and dependencies in the SBS plan). It shows that 5S (three variants, i.e., 5S-Basic, 5S-Hierarchical, 5S-Covering) can answer all the SBS plans regardless of the size of the SBS plan, which demonstrates

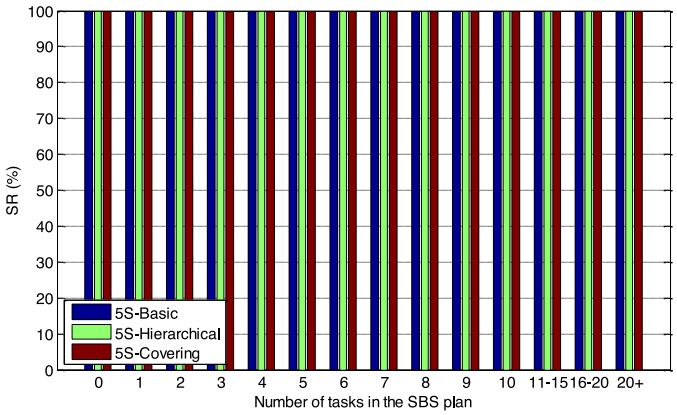


Fig. 7. Success rate of answering SBS plans (experiment series A).

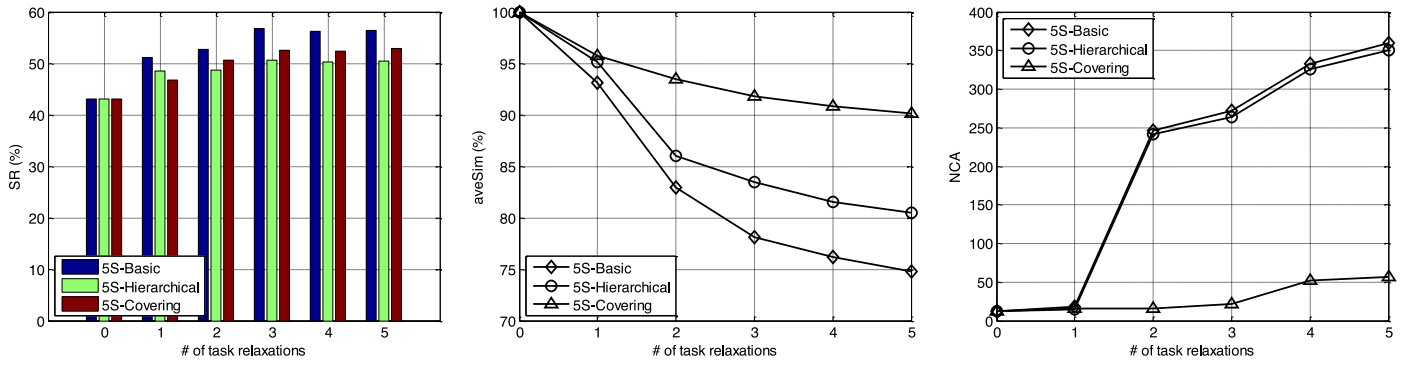
that a user can indeed use 5S to identify the services needed for building any of the SBSs in the PW dataset by entering the SBS plans that represent the tasks of the SBS.

Fig. 8 shows the similarity search performance of the 5S method in experiment series A. Here, for each SBS plan, we manually remove the corresponding SBS alternative in *HL* before performing 5S. Therefore, 5S can hardly find an exact SBS solution (the NCA is almost 0) for zero task relaxations. Then 5 s helps users find similar SBS solutions by progressively relaxing the SBS plan. To consider the substructure similarity goal, the following discussion assumes 1–5 task relaxations.

Fig. 8(a) shows that the SRs obtained by 5S-Basic, 5S-Hierarchical, and 5S-Covering increases with increasing number of task relaxations. This indicates that the more a user relaxes requirements, the more likely he/she is to get a SBS solution similar to the relaxed SBS plan. The results also show that, in most cases, 5S-Covering maintains its slight advantage over 5S-Hierarchical, and 5S-Basic has higher SR values than 5S-Covering and 5S-Hierarchical. However, this does not mean that 5S-Basic performs better than 5S-Covering and 5S-Hierarchical. Note that the SR value only represents the probability of returning a possible solution and does not represent the quality of the returned solution. With the following experiment results, we analyze the performance of the three approaches for the aveSim and NCA values.

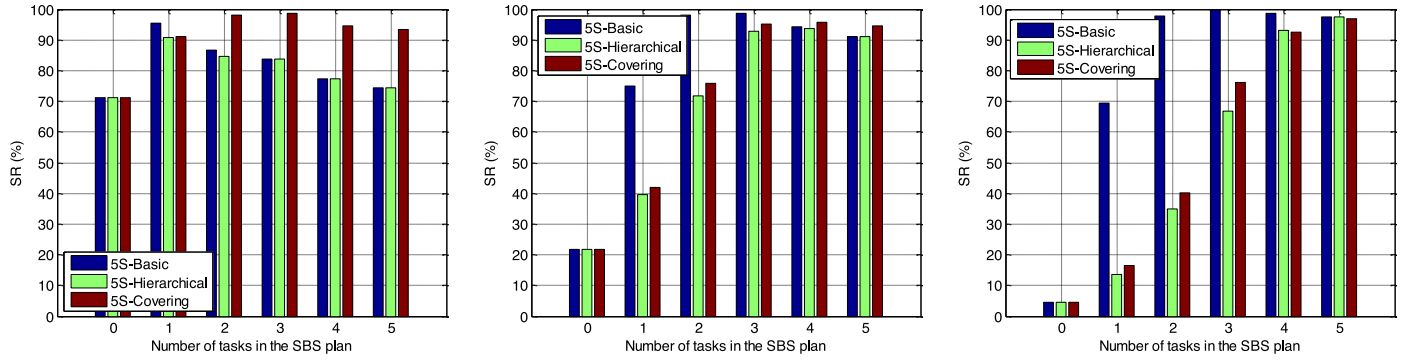
Fig. 8(b) shows that 5S-Covering consistently gives highly representative search results. If we allow two tasks to be relaxed for Q, the 5S-Basic (5S-Hierarchical) approach can return a candidate answer set with an average similarity of 82.89% (86.01%) with the SBS plan Q, whereas our 5S-Covering approach can reach 93.46%. In addition, as k increases, the gap between 5S-Covering and the other approaches also increases. For example, when $k = 5$, the average similarity of the results returned by the 5S-Basic (5S-Hierarchical) approach is only 74.8% (80.5%), whereas the 5S-Covering approach still delivers 90.1% similarity. The increase in k mainly affects the calculation of the maximum number of allowed *sf* misses. However, the **Merge** and **Add** operations in 5S-Covering prevent $f_{\max}$ from getting too large, so the search results are closer to the user's requirements.

Fig. 8(c) shows that 5S-Covering provides the best filtering, outperforming both 5S-Hierarchical and 5S-Basic. This result is attributed to the fact that, in each *filter*, the 5S-Covering approach filters out the *sc* in *HL* that satisfies $FS(Q) \cap FS(sc) = \Phi$, so the 5S-



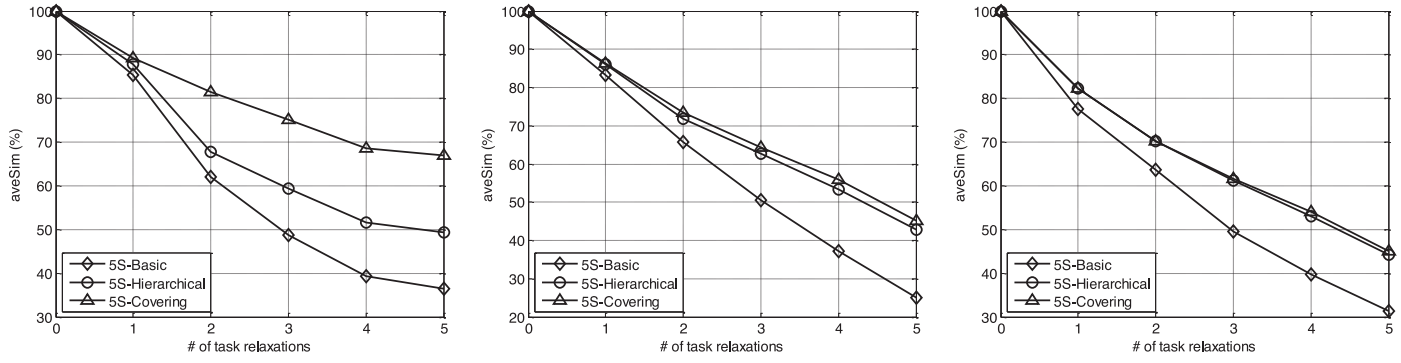
(a) SR (b) aveSim (c) NCA

Fig. 8. Effectiveness of different approaches (experiment series A).



(a) Q_5 (b) Q_{10} (c) Q_{15}

Fig. 9. Impact of number of task relaxations (k) on SR (experiment series B).



(a) Q_5 (b) Q_{10} (c) Q_{15}

Fig. 10. Impact of number of task relaxations (k) on aveSim (experiment series B).

Covering approach returns a smaller set of candidate answers. For example, if we allow five tasks to be relaxed for the SBS plans Q, the 5S-Basic (5S-Hierarchical) approach prunes off 87.7% (88.03%) of the dataset, whereas 5S-Covering prunes off 98.3%. The results obtained from Fig. 8(b) and Fig. 8(c) demonstrate that, given a relaxed SBS plan, 5S-Covering can find similar SBS solutions with higher quality, which is more suitable for the user's similarity search needs.

Figs. 9–11 present the results obtained from experiment series B, which comprehensively compared the performance of the 5S method under three scaled SBS plans (Q_5, Q_{10}, Q_{15}).

As shown in Fig. 9, increasing the number of task relaxations (denoted by k) does not necessarily increase the SR value. Overall, as k increases, the SR increases at the beginning before reaching its peak value, and then starts to decrease as k continues to increase. The result is attributed to the fact that, as k increases, the maximum number of allowed *sf* misses may approach the total number of *sfs* in each *filter*, which increases the occurrence of the *high fragment difference* and *full fragment difference* phenomena, thus lowering the SR. This phenomenon is more obvious in the case of smaller SBS plans. Consider 5S-Covering in Fig. 9(a) as an example. The SR of 5S-Covering increases from its lowest point at 71.3% to its highest point at 98.7% as k increases from 0 to 3, and decreases from 98.7% to 93.5% as k increases from 3 to 5. This tells us that, by increasing the number of task relaxations, the probability of finding suitable solutions will not always increase. The results also show that the SR of the 5S-Basic, 5S-Hierarchical, and 5S-Covering methods decrease as the SBS plan scales up. The reason for the decrease in SR is that the increased size of the SBS plan means more complex query structures, making it more difficult to

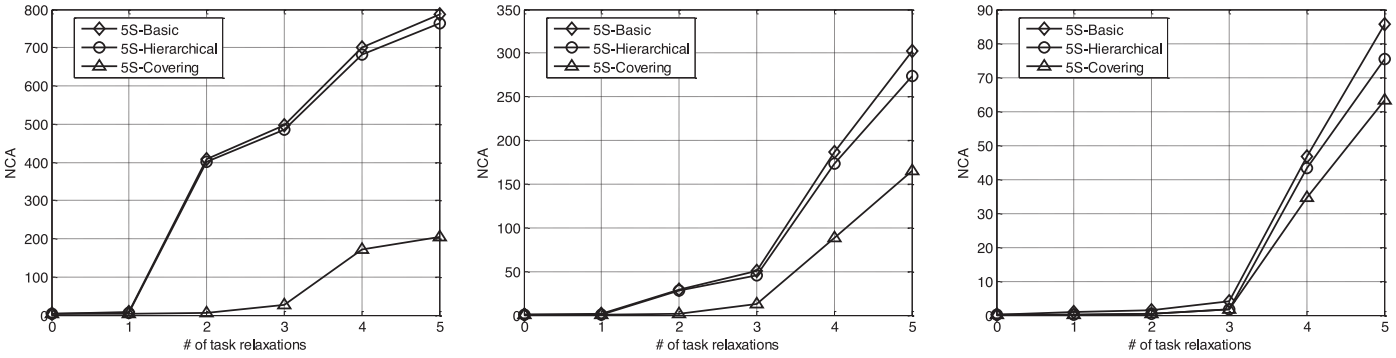


Fig. 11. Impact of number of task relaxations (k) on NCA (experiment series B).

find a SBS solution that meets or approximately meets user needs. In addition, similar to Fig. 8(a), 5S-Basic has higher SR values than 5S-Covering and 5S-Hierarchical in most case. In the following, we further discuss the performance of 5S method based on the results of the experimental series B.

The results presented in Fig. 10 illustrate that the value of aveSim decreases as the number of task relaxations increases. Compared with the size of Q, the increase in the number of task relaxations more strongly affects aveSim. In the Q_5 experiments [Fig. 10(a)], the 5S-Covering approach provides the highest aveSim, outperforming the 5S-Hierarchical approach, which provides the second-highest aveSim, and the 5S-Basic approach, which provides the lowest aveSim. If we allow one task to be relaxed for the queries in Q_5, aveSim for 5S-Covering, 5S-Hierarchical, and 5S-Basic is 89.33%, 87.9%, and 85.3%, respectively. However, the advantages of 5S-Covering start to become manifest as the number of task relaxations increases. For example, if we allow five tasks to be relaxed, aveSim for 5S-Covering, 5S-Hierarchical, and 5S-Basic is 67.05%, 49.34%, and 36.5%, respectively. The results for the Q_{10} experiments [Fig. 10(b)] again show that 5S-Covering provides the highest aveSim, with 5S-Hierarchical coming in second. Even for $k = 5$, the aveSim for 5S-Covering and 5S-Hierarchical can exceed 40%, whereas aveSim for 5S-Basic is only about 25%. Fig. 10(c) presents the results obtained from the Q_{15} experiments. As k increases, 5S-Covering maintains its slight advantage over 5S-Hierarchical in most cases, and the similarity of the 5S-Basic approach remains the lowest. Fig. 10 also shows that 5S-Covering significantly outperforms 5S-Hierarchical and 5S-Basic when the number of task relaxations increases from 2 to 5. However, when the size of Q increases, the performance of 5S-Covering is close to that of 5S-Hierarchical. This result is attributed to the increasing number of *sf*s included in the Q as Q scales up. The 5S-Covering approach adaptively groups the *sf* into several complementary subsets according to the *selectivity* of *sf*. However, the hierarchical clustering operation in 5S-Hierarchical strictly groups the *sf*s into three clusters with high *selectivity*, medium *selectivity*, and low *selectivity*, which increases the occurrence of *high fragment difference* and *full fragment difference*. This phenomenon is more obvious for small scale Q (i.e., Q_5). Based on the results shown in Fig. 10, we thus conclude that a user can use 5S-Covering to find SBS solutions with higher similarity. In other words, 5S-Covering is much more effective for finding a solution than 5S-Hierarchical or 5S-Basic.

The results presented in Fig. 11 illustrate that the NCA for all approaches increases as the number of task relaxations (denoted by k) increases (gradually at the beginning and rapidly afterward). Compared with the other approaches, 5S-Covering provides a significantly lower NCA in most cases. When $k = 0$, the NCA returned by all approaches is close to zero [see Figs. 11(a)–11(c)], indicating that the *HL* contains no *sc* that exactly matches the SBS plan Q. However, when $k > 1$, the value of NCA sharply increases in all experiments shown in Fig. 11. Despite all this, the 5S-Covering approach still provides the best filtering. In addition, the NCA of all approaches decreases as Q scales up. When we fix k to 5, the average NCA of 5S-Covering, 5S-Hierarchical, and 5S-Basic is respectively 205.7, 763.43, and 787.04 [see Figs. 11(a)]; When $k = 10$, the average NCA is 165.16, 274.06, and 302.44, respectively [see Figs. 11(b)]; When $k = 15$, the average NCA is 63.38, 75.42, and 85.69, respectively [see Figs. 11(c)]. These results clearly show that the scale of the SBS plans Q significantly affects the NCA for all approaches. These results have a twofold explanation: First, the *fragment difference* (denoted by *FD*) between Q and *sc* increases as Q scales up, and if *FD* exceeds $f_{\max}$, we can discard *sc*. Therefore, compared with Q_5 and Q_{10}, it is more difficult to find similar SBSs for Q_{15}. Second, the number of SBS tasks in the PW dataset is generally less than ten, so SBS plans with more than ten tasks return fewer solutions.

The results obtained from this series of experiments demonstrate that, given only a SBS plan, 5S can find SBS solutions as specified in the PW dataset, as well as similar alternatives when an exact solution is unavailable. This is reasonable because for a SBS plan with complex function requirements, building an SBS that can meet all its requirements is often too restrictive.

5.5. Evaluation of efficiency

Fig. 12 shows the computation times for 5S-Covering, 5S-Hierarchical, and 5S-Basic in experiment series A for different numbers of task relaxations (denoted by k). In general, the computation time increases as k increases. However, 5S-Covering performs very well, with only a slight increase in computation time in response to the increase in the number of task relaxations in Q. For 5S-Hierarchical, the clustering component combines features that differ in size by one at most and divides them into three clusters, which may generate too many *filter*s. For example, when the feature size of Q is in the range of 2–6, 5S-Hierarchical will generate 12 *filter*s, which is not required when using 5S-Covering. Moreover, these empirical studies show that the number of *filter*s generated by 5S-Covering is less than the number of *filter*s generated by 5S-Hierarchical. Thus, 5S-Hierarchical requires a much longer computation time than 5S-Covering. For 5S-Basic, building a *task-fragment matrix* based on all the *sf*s in Q is time-consuming, and the large matrix retards the *sf* miss estimate. Thus, 5S-Basic requires significantly more computation time than 5S-Covering. In addition, the computation times are roughly linear in k, which indicates high scalability. We thus conclude that the efficiency is acceptable in most, if not all, real-world applications.

Fig. 13 shows how the number of task relaxations affects the average computation time required by 5S-Covering, 5S-

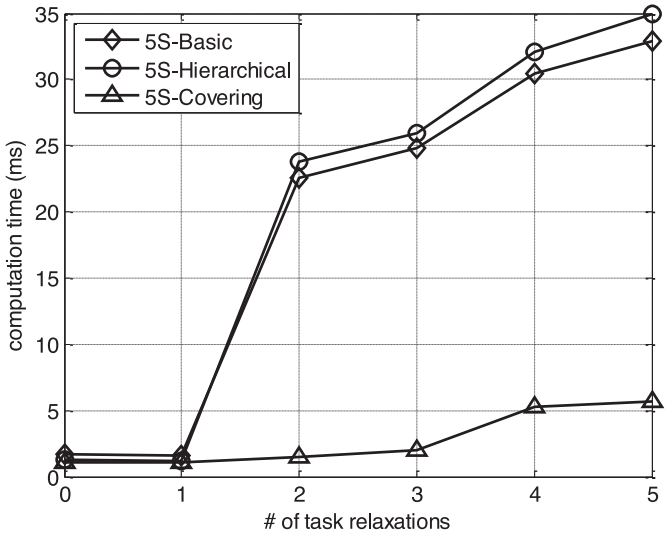


Fig. 12. Impact of number of task relaxations (k) on efficiency (experiment series A).

Hierarchical, and 5S-Basic in experiment series B. As shown in Figs. 13(a)–13(c), the differences in their computation times are similar to that shown in Fig. 12. 5S-Covering remains the fastest approach. Note also that the computation times gradually decrease as the SBS plans Q scale up. This result may be understood by combining the results of Fig. 11, which shows that SBS plans with simpler structure lead to the more candidate SBS solutions. Thus, in each *filter*, the search space for Q_5 is larger than that for Q_{10} and Q_{15}, which results in more computational overhead. This interesting phenomenon is particularly evident for large k. Overall, Figs. 12 and 13 demonstrate that the proposed 5S method is sufficiently fast for real-world applications.

Based on these experimental results for computation time, we conclude that the proposed 5S approach is highly efficient and can scale with the number of tasks in the SBS plan. Furthermore, because this approach can provide users with multiple alternative solutions, it should be preferred by users concerned about multiple solutions.

6. Discussion

6.1. Evaluations on λ

The parameter λ determines the maximum number of tasks in the feature, i.e., in order for the number of features not to increase exponentially. To evaluate how λ affects the experimental results, in this experiment, we changed λ from 2 to 6 in steps of 1, and the number of service relaxations was changed from 0 to 5 in steps of 1 to compare the average similarity of the results obtained by the different approaches. The experiment setup is the same as experiment series A.

The experimental results presented in Fig. 14 have demonstrated that, higher λ yields a better aveSim. The reason for this result very simple: In the feature-extraction section, a lower λ leads to fewer *sfs* that contain a lot of simple *sf* (small size *sf*). A set of simple *sf* sets does not adequately represent a SBS.

Fig. 14(a) presents the results of the 5S-Basic approach for answering the 2926 SBS plans. The results illustrate that increasing λ does not directly influence aveSim, which is because the single *filter* approach using all *sfs* together does not perform well due to the *high fragment difference* identified in Section 4.3.2. It is difficult to improve the results of aveSim by increasing the value of λ. In this set of experiments, the number of service relaxation constraints is the only factor that affects aveSim. The results of Fig. 14(b) show that increasingλ leads to an increase in aveSim for the 5S-Hierarchical approach. The reason is that a larger λ increases the number of features contained in the filter. Fig. 14(c) presents the results of the 5S-Covering approach for answering the 2926 SBS plans, where the various values of λ give similar results except for $\lambda = 2$. This is because a single edge as a *sf* makes it challenging to describe the complete structure of the SBS. However, even for small λ, the 5S-Covering results remain superior to all the results of the other two approaches. This shows that the 5S-Covering method can still find similar solutions even with a small feature quantity and a simple feature structure.

The results also show that, as the number of task relaxations increases, the solution deviates from the original Q, as indicated by the decrease in aveSim. This phenomenon is more obvious for a smaller λ. Because a user may not like the Q to deviate too far from her or his requirement, we use $\lambda = 6$ in this article.

6.2. Comparison with other approaches

To highlight the benefits of substructure similarity searching in the SBS engineering process, we compare 5S with the following relevant state-of-the-art techniques:

ASCP (Deng et al., 2013) (**A**utomatic **S**ervice **C**omposition based on **P**lanning-graph Algorithm): This approach employs the planning-graph technique to find the top k solutions in a large-scale service repository.

KS3 (He et al., 2017) (**K**eyword **S**earch for **S**ervice-based **S**ystems): This approach allows users to search for component services to build SBSs by typing keywords that represent the desired SBS tasks.

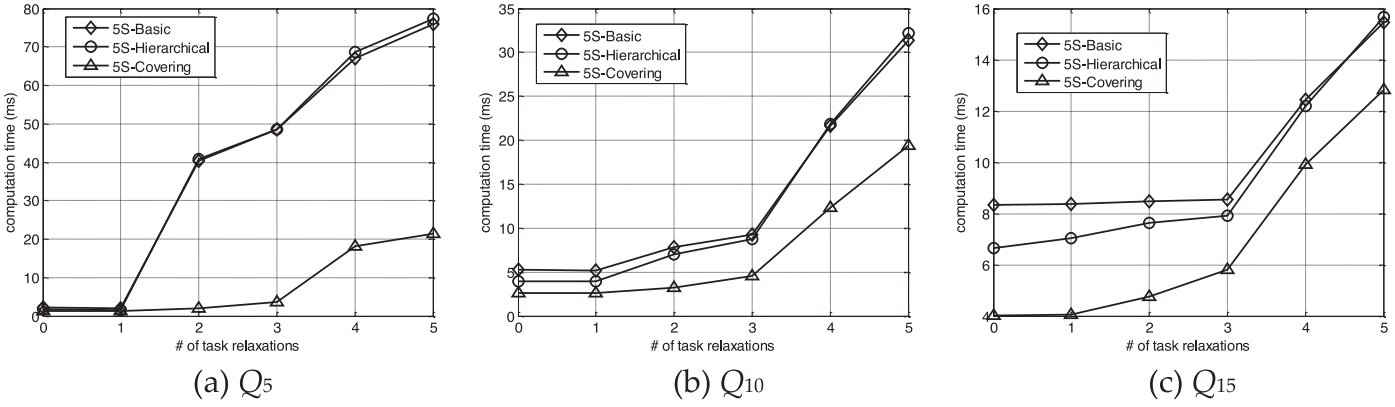


Fig. 13. Impact of number of task relaxations (k) on efficiency (experiment series B).

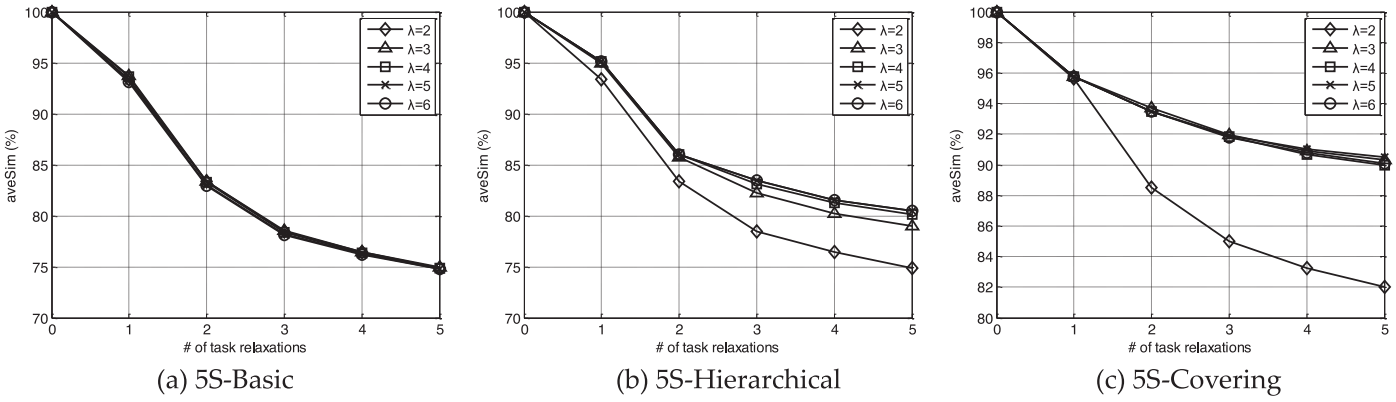


Fig. 14. Impact of parameter λ on aveSim.

In the comparative experiments, we assume that each SBS in Q_m (m = 5, 10, 15) in Section 5.3 is the one that the user wants to build. We use these comparison methods to search for SBS solutions in the SBS repository (this is the same as the *HL* discussed in Section 5.3) and evaluate their search performance (measured by Success Rate, Number of Candidate Answers, Average Similarity, and computational overhead).

To compare these methods in a fair and objective manner, we adjusted them somewhat as follows: (1) The 5S first does an exact search; if no SBS solution is found, the SBS plan is relaxed progressively until a relaxation threshold is reached (at most 5 tasks may be relaxed) or a similar SBS solution is found. (2) A user request in the ASCP is denoted (I, O), where I is the set of requested input parameters and O is the set of requested output parameters. We simplify I and O to the first and last tasks of the corresponding SBS plan, respectively, and we only focus on functional requirements. (3) In KS3, we consider two cases: In the first case, each query contains three keywords, which are obtained from the first, the last, and a random task used by the corresponding SBS plan. In the second case, each query contains all the component tasks in the corresponding SBS plan (denoted "KS3-all"); we only focus on functional requirements in KS3 and KS3-all.

Table 4 presents the SR, aveSim, NCA, and computational overhead resulting from the use of different methods in the comparative experiments involving SBS plans of differing size.

As shown in Table 4, ASCP returns better SRs than the three existing methods for SBS plans of different scale and provides users with more alternatives, as demonstrated by the greater NCA. However, for the SBS solution, ASCP returns a low value for aveSim, which means that the SBS solution returned by ASCP may deviate from the user requirements. This is because ASCP only considers the user input and output, and does not consider which services the user wants to use to engineer their own SBS. The aveSim value of KS3 is slightly greater than that of ASCP because KS3 uses user-supplied keywords for searching, so it makes the search more stringent and thereby returns a lower value for SR. KS3-all is based on a large number of keywords and thereby requires more computation time and returns lower values for SR, aveSim, and NCA than other methods. Because finding SBS solutions that cover all of the component functions in the SBS plan is usually too strict. Take Q_{15} for example: the SR of KS3-all is only 4.5%, and NCA is almost zero, which means that in the case of large SBS plans, KS3-all has difficulty finding a SBS solution to meet the user requirements. Our substructure similarity-search method significantly outperforms existing methods. Of all methods, 5S-Covering obtains the best value for aveSim. For Q_5, Q_{10}, and Q_{15}, the aveSim values are 93.6%, 77.5%, and 68.9%, respectively, which are significantly greater than other methods. In addition, it guarantees high SR (average SR above 99%) and low computational overhead (average computation time is less than 20 ms). Thus, 5S-Covering is the best approach for the service composition for an SBS in most real-world applications.

To summarize these evaluations, we recommend applying the 5S method in the following service-based scenarios: (1) In the process of building a SBS, the user may not know the exact composition desired for the full SBS but require that it contain a set of small functional fragments. (2) In case some component functions are invalid in the optimal SBS, 5S can help users select another SBS as an alternative from among the similar SBSs. This can improve the feasibility of the result set. (3) In extreme cases, no composed service can meet the exact needs of all component functions in a SBS plan. In a nutshell, in scenarios for which no SBS solution is available or where an extremely fast SBS is required, 5S is a more a suitable method because of its outstanding advantage in efficiency over traditional methods.

Table 4
Performance comparison of different approaches under different SBS plan scales.

SBS plan size	Method	Metrics			
		SR	aveSim	NCA	Computational overheads
Q_5	ASCP	100.0%	28.0%	14.5	65.2 ms
	KS3	88.6%	35.7%	2.9	71.4 ms
	KS3-all	71.3%	34.1%	2.7	117.2 ms
	5S-covering	**100.0%**	**93.6%**	**2.8**	**11.3 ms**
Q_{10}	ASCP	100.0%	17.8%	8.2	103.2 ms
	KS3	70.9%	16.9%	1.5	123.3 ms
	KS3-all	21.8%	7.8%	0.3	342.8 ms
	5S-covering	**99.7%**	**77.5%**	**1.3**	**16.1 ms**
Q_{15}	ASCP	100.0%	13.9%	8.2	165.1 ms
	KS3	59.7%	9.9%	1.2	188.3 ms
	KS3-all	4.5%	1.4%	0.1	570.6 ms
	5S-covering	**98.9%**	**68.9%**	**1.2**	**19.5 ms**

7. Related work

Service composition for the SBS has been an active research field over the past few years. The rapid growth in the number of available web-delivered services, including web services, APIs, and data feeds, makes service composition for the SBS more popular. The service composition process for engineering the SBS mainly consists of four phases: *system planning, service discovery, service selection*, and *service delivery*. Numerous studies have worked on solving the various problems in these phases.

In the *system planning* phase, the user determines the tasks to be performed and the order in which they should be executed and then implements the functional requirements of the SBS by invoking its component services one after another. In this phase, many studies have used AI techniques (Pistore et al., 2005;

Oh et al., 2008; Zou et al., 2014; Deng et al., 2013; Küster et al., 2007) to identify the tasks needed to implement a SBS. The main idea is to translate the service composition into a planning problem that can be fed into an efficient AI planner. For example, in Ref. (Deng et al., 2013), the authors proposed an approach based on planning graphs to solve the top-k QoS-aware automatic service composition problem. The results show that their approach can provide more alternative SBSs with the optimal QoS for users. After the *system planning* phase, the tasks of a SBS are determined.

In the *service discovery* phase, the system engineer identifies a set of suitable candidate services for each task based on the functional and semantic information of candidate services. As one of the core techniques in SOA, the *service discovery* phase has been studied extensively in the past two decades. Existing approaches for Web-service discovery typically use keyword-matching technologies to find published Web services (Cassar et al., 2014; Wu et al., 2016; Blake and Nowlan, 2011). Although such approaches can benefit from information-retrieval techniques and have incorporated several enhancement methods (e.g., structural matching of WSDL (Wang and Stroulia, 2003) and clustering algorithms (Dong et al., 2004; Cong et al., 2015)), they still suffer from low precision and recall (Plebani and Pernici, 2009). To improve the accuracy of service matching, various semantic Web-service languages based on ontology technology have been proposed, such as DSD (Küster et al., 2007) and OWLS-MX (Klusch et al., 2009). The ontology contains the definitions of predefined task-based queries which enrich the service description and SBS specification. Existing semantics-aware approaches can be further divided into logic-based (Roman et al., 2015; Chen et al., 2015) and non-logic-based (Wang et al., 2017; Chen et al., 2017) subcategories. Both subcategories have made significant contributions to the efficiency and practicality of service discovery. After the service-discovery phase, a set of functionally equivalent candidate services are determined for each task of a SBS.

In the *service selection* phase, the users selects a particular service from each set of candidate services to compose the target SBS, which must fulfill quality constraints and achieve the SBS owners' optimization goals. Moreover, the work in Ref. (Bonatti and Festa, 2005) proves that the service selection with multiple quality constraints for service composition is NP-complete. To address this issue, significant research has focused on solving the SBS quality-aware service selection problem (Ardagna and Pernici, 2007; Liangzhao et al., 2004; He et al., 2014; Calinescu et al., 2011; Trummer et al., 2014). The general idea is to model service selection as a multi-choice 0–1 knapsack problem or a multi-constraint optimal path problem, which can be solved by using integer programming (IP) or heuristic algorithms. For example, Ref. (Yu et al., 2007) models quality-aware web service selection as a 0–1 knapsack problem and present heuristic algorithms to find near-optimal solutions in polynomial time. References (Liangzhao et al., 2004; Zeng et al., 2003) present AgFlow, a QoS-aware middleware platform that supports quality-driven Web service compositions, which adopt a method based on IP for selecting an optimal execution plan. After the *service selection* phase, a SBS is determined.

Finally, in the *service delivery* phase, the selected services are executed by a system engine in a certain order to realize the SBS. In this phase, how to deploy the component services of a SBS becomes a critical issue. Numerous studies have looked at such issues (e.g., load balancing (Bramson et al., 2010), resource management and allocation (Wang et al., 2012; Xu et al., 2014), and fault tolerance issues (Zheng and Lyu, 2015)). In addition, some studies have explored the issue of service deployment for efficient execution of SBSs. For example, Ref. (Huang and Shen, 2015) models the service-deployment problem as a minimum k-cut problem and proposes an integrated approach, which takes into consideration not only inter-service communication costs but also the potential parallelism among services. At the end of this phase, the SBS is executed by a system engine to meet the user needs.

These phases can involve many complex techniques and methods to solve the various problems, requiring users to spend significant time and effort to select, learn, and apply these techniques to develop a SBS solution. This bottleneck has been a significant obstacle to extending the use of SOA. Some assistive approaches have been proposed. The authors of (Liu et al., 2015) first propose a technique to extract useful information from multiple sources to abstract service capabilities with a set tags. This supports intuitive expression of user's desired composition goals by simple queries, without having to know underlying technical details. A planning technique is then proposed to explore SBS solutions by looking up services whose tags match the tags describing the SBS. An efficient approach is proposed in (Huang et al., 2015) for helping users navigate from the entry service to the exit service through multiple queries. However, this planning technique does not support building SBS with more than two tags. The work in (He et al., 2017) presents a novel approach that integrates and automates the system planning, service discovery and service selection operations for building SBSs. It assists users without detailed knowledge of SOA techniques in identifying SBS solutions with only a few keywords that describe the tasks of the SBSs, which can significantly save the time and effort during the SBS engineering process.

Although significant research has been devoted to solving the problems in the SBS engineering process, building a SBS remains a highly complex task. In particular, when the pre-specified SBS plan involves a complicated structure, it has become increasingly difficult to select the appropriate Web services from the massive number of candidates to build an SBS that exact meets the complex structural requirements of the user. Therefore, a similarity search becomes an essential operation that must be efficiently supported. However, most of the existing methods for engineering SBSs do not consider this operation. A pressing need therefore exists for an assistive approach that can automatically fine-tune the SBS plan and quickly provide users with approximate SBS solutions for the case in which the expected SBS cannot be built.

The 5S method is a substructure similarity search mechanism that helps users search for Web services to build SBS that meets or approximately meets user needs and requires only a small amount of computation time. This approach models the SBS engineering process as a substructure similarity search problem, which has been studied in various fields (Yan et al., 2005; Willett et al., 1998; Shasha et al., 2002). By progressively relaxing the SBS plan, the 5S method returns similar SBSs that meet the user's needs, thereby significantly saving time and effort in the SBS engineering process.

8. Conclusions and future work

This study investigates the problem of substructure similarity search in the SBS engineering process and proposes the 5S method, which is an approximate matching approach for building service-based systems (SBSs). It assists users in identifying similar SBS solutions by progressively relaxing a pre-specified SBS plan. The 5S method offers a new paradigm for efficient SBSs engineering by shortening the build cycle in SBS engineering. The comprehensive analysis of experiments shows the effectiveness and efficiency of the 5S method.

In future work, we will investigate how to handle multiple non-functional properties and take more consideration of users' preferences to obtain similar SBSs. And we will further consider how different clustering operations affect the effectiveness and efficiency of the 5S method. In addition, we will also investigate the statistical relationship between the structure of the extracted features and the results of the 5S method.

Declaration of Competing Interest

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

CRediT authorship contribution statement

Jintao Wu: Conceptualization, Methodology, Software, Writing - original draft. **Xing Guo:** Visualization, Data curation. **Guijun Yang:** Validation, Formal analysis, Writing - review & editing, Funding acquisition. **Shuhui Wu:** Investigation, Resources. **Jianguo Wu:** Supervision, Project administration.

Acknowledgments

This study was supported by the National Key Research and Development Program of China (no. 2017YFE0122500 and no. 2016YFD020060306), the National Science and Technology Program of China (no. 2015BAK24B00), the Nature Science Program of Anhui Province (no. 1908085MF181), and the Special Funds for Technology innovation platform building sponsored by the Beijing Academy of Agriculture and Forestry Sciences (no. PT2019-29).

References

Al-Masri, E., Mahmoud, Q.H., 2008. Investigating web services on the world wide web. In: Proc of 17th International Conference on World Wide Web (WWW 2008), Beijing, China, pp. 795–804.
Ardagna, D., Pernici, B., 2007. Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. 33 (6), 369–384.
Baresi, L., Guinea, S., 2011. Self-supervising BPEL processes. IEEE Trans. Softw. Eng. 37 (2), 247–263.
Barros, A.P., Dumas, M., 2006. The rise of web service ecosystems. IT Prof. Mag. 8 (5), 31–37.
Berretti, S., Del Bimbo, A., Vicario, E., 2001. Efficient matching and indexing of graph models in content-based retrieval. IEEE Trans. Pattern Anal. Mach. Intell. 23 (10), 1089–1105.
Blake, M.B., Nowlan, M.E., 2011. Knowledge discovery in services (kds): aggregating software services to discover enterprise mashups. IEEE Trans. Knowl. Data Eng. 23 (6), 889–901.
Bonatti, P.A., Festa, P., 2005. On optimal service selection. In: Proc of 14th International Conference on World Wide Web (WWW 2005), Chiba, Japan, pp. 530–538.
Bramson, M., Lu, Y., Prabhakar, B., 2010. Randomized load balancing with general service time distributions. ACM SIGMETRICS Perform. Eval. Rev. 38 (1), 275–286.
Brogi, A., Corfini, S., Popescu, R., 2008. Semantics-based composition-oriented discovery of web services. ACM Trans. Internet Technol. 8 (4), 1–39.
Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G., 2011. Dynamic QoS management and optimization in service-based systems. IEEE Trans. Softw. Eng. 37 (3), 387–409.
Cassar, G., Barnaghi, P., Moessner, K., 2014. Probabilistic matchmaking methods for automated service discovery. IEEE Trans. Serv. Comput. 7 (4), 654–666.
Chen, F., Li, M., Wu, H., Xie, L., 2015. Web service discovery among large service pools utilising semantic similarity and clustering. Enterp. Inf. Syst. 11 (3), 452–469.
Chen, F., Lu, C., Wu, H., Li, M., 2017. A semantic similarity measure integrating multiple conceptual relationships for web service discovery. Expert. Syst. Appl. 67, 19–31. doi:10.1016/j.eswa.2016.09.028.
Cong, Z.J., Fernandez, A., Billhardt, H., Lujak, M., 2015. Service discovery acceleration with hierarchical clustering. Inf. Syst. Front. 17 (4), 799–808.
Deng, S., Wu, B., Yin, J., Wu, Z., 2013. Efficient planning for top-K web service composition. Knowl. Inf. Syst. 36 (3), 579–605.
Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J., 2004. Similarity search for web services. In: Proc of 30th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada, pp. 372–383.
Feige, U., 1998. A threshold of ln n for approximating set cover. J. ACM 45 (4), 634–652.
He, H., Singh, A.K., 2008. Query language and access methods for graph databases. In: Proc of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008), 40, Vancouver, Bc , Canada, pp. 405–418.
He, Q., Yan, J., Jin, H., Yang, Y., 2014. Quality-Aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. IEEE Trans. Softw. Eng. 40 (2), 192–215.
He, Q., Zhou, R., Zhang, X., Wang, Y., Ye, D., Chen, F., Grundy, J., Yang, Y., 2017. Keyword search for building service-based systems. IEEE Trans. Softw. Eng. 43 (7), 658–674.
Hoffmann, J., Bertoli, P., Pistore, M., 2007. Web service composition as planning, revisited: in between background theories and initial state uncertainty. In: Proc of 22nd AAAI Conference on Artificial Intelligence (AAAI 2007), Vancouver, British Columbia, Canada, pp. 1013–1018.
Huang, G., Ma, Y., Liu, X.Z., et al., 2015. Model-based automated navigation and composition of complex service mashups. IEEE Trans. Serv. Comput. 8 (3), 494–506.
Huang, K.-.C., Shen, B.-.J., 2015. Service deployment strategies for efficient execution of composite SaaS applications on cloud platform. J. Syst. Softw. 107, 127–141. doi:10.1016/j.jss.2015.05.050.
Jatoth, C., Gangadharan, G., Buyya, R., 2019. Optimal fitness aware cloud service composition using an adaptive genotypes evolution based genetic algorithm. Future Gener. Comput. Syst. 94, 185–198.
Klein, A., Ishikawa, F., Honiden, S., 2014. SanGA: a self-adaptive network-aware approach to service composition. IEEE Trans. Serv. Comput. 7 (3), 452–464.
Klusch, M., Fries, B., Sycara, K., 2009. OWLS-MX: a hybrid semantic web service matchmaker for owl-s services. J. Web Semant. 7 (2), 121–133.
Kuramochi, M., Karypis, G., 2001. Frequent subgraph discovery. In: Proc of 1th in International Conference on Data Mining (ICDM 2001), San Jose, California, USA, pp. 313–320.
Küster, U., König-Ries, B., Stern, M., Klein, M., 2007. DIANE: an integrated approach to automated service discovery, matchmaking and composition. In: Proc of 16th International Conference on World Wide Web (WWW 2007), Banff, Alberta, Canada, pp. 1033–1042.
Laleh, T., Paquet, J., Mokhov, S., et al., 2018. Constraint verification failure recovery in web service composition. Future Gener. Comput. Syst. 89, 387–401.
Liangzhao, Z., Benatallah, B., H., A.H., Dumas, M., Kalagnanam, J., Chang, H., 2004. QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng 30 (5), 311–327.
Liu, X.Z., Ma, Y., Huang, G., et al., 2015. Data-driven composition for service-oriented situational web applications. IEEE Trans. Serv. Comput. 8 (1), 2–16.
Oh, S.C., Lee, D., Kumara, S.R.T., 2008. Effective web service composition in diverse and large-scale service networks. IEEE Trans. Serv. Comput. 1 (1), 15–32.
Peng, Y., Fan, Z., Xu, J., Choi, B., Bhowmick, S., 2015. Authenticated subgraph similarity searchin outsourced graph databases. IEEE Trans Knowl Data Eng 27 (7), 1838–1860.
Pistore, M., Marconi, A., Bertoli, P., Traverso, P., 2005. Automated composition of web services by planning at the knowledge level. In: Proc of 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, UK, pp. 1252–1259.
Plebani, P., Pernici, B., 2009. URBE: web service retrieval based on similarity evaluation. IEEE Trans. Knowl. Data Eng. 21 (11), 1629–1642.
Raymond, J.W., Gardiner, E.J., Willett, P., 2002. Rascal: calculation of graph similarity using maximum common edge subgraphs. Comput. J. 45 (6), 631–644.
Roman, D., Kopecky, J., Vitvar J. Domingue, T., Fensel, D., 2015. WSMO-Lite and hRESTS: lightweight semantic annotations for web services and RESTful APIs. J. Web Semant. 31, 39–58. doi:10.1016/j.websem.2014.11.006.
Shang, H., Zhang, Y., Lin, X., Yu, J.X., 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In: Proceedings of the VLDB Endowment (PVLDB), 1, pp. 364–375.
Shang, H., Zhu, K., Lin, X., Zhang, Y., Ichise, R., 2010. Similarity search on supergraph containment. In: Proc of 26th International Conference on Data Engineering (ICDE 2010), Long Beach, CA, USA, pp. 637–648.
Shasha, D., Wang, J.T., Giugno, R., 2002. Algorithmics and applications of tree and graph searching. In: Proc of 21th ACM SIGMOD Conference on Principles of Database Systems (PODS 2002), Madison, Wisconsin, USA, pp. 39–52.
Subashini, S., Kavitha, V., 2011. A survey on security issues in service delivery models of cloud computing. J. Netw. Comput. Appl. 34 (1), 1–11.
Trummer, I., Faltings, B., Binder, W., 2014. Multi-objective quality-driven service selection-a fully polynomial time approximation scheme. IEEE Trans. Softw. Eng. 40 (2), 167–191.
Vizcarrondo, J., Aguilar, J., Expósito, E., et al., 2017. ARMISCOM: self-healing service composition. Serv. Oriented Comput. Appl. 11 (3), 345–365.
Vollino, B., Becker, K., 2013. Usage profiles: a process for discovering usage patterns over web services and its application to service evolution. Int. J. Web Serv. Res. 10 (1), 1–28.
Wang, J., Gao, P., Ma, Y., He, K., Hung, P.C.K., 2017. A web service discovery approach based on common topic groups extraction. IEEE Access 5, 10193–10208. doi:10.1109/ACCESS.2017.2712744.
Wang, X., Du, Z., Chen, Y., 2012. An adaptive model-free resource and power management approach for multi-tier cloud environments. J. Syst. Softw. 85 (5), 1135–1146.
Wang, Y.W.Y., Stroulia, E., 2003. Flexible interface matching for web-service discovery. In: Proc of 4th International Conference on Web Information Systems Engineering (WISE 2003), Rome, Italy, pp. 147–156.
Willett, P., Barnard, J.M., Downs, G.M., 1998. Chemical similarity searching. J. Chem. Inf. Comput. Sci. 38 (6), 983–996.
Wu, Y., Yan, C., Ding, Z., Liu, G., Wang, P., Jiang, C., Zhou, M., 2016. A multilevel index model to expedite web service discovery and composition in large-scale service repositories. IEEE Trans. Serv. Comput. 9 (3), 330–342.
Xu, Q., Li, X., Ji, H., Du, X., 2014. Energy-efficient resource allocation for heterogeneous services in OFDMA downlink networks: systematic perspective. IEEE Trans. Veh. Technol. 63 (5), 2071–2082.
Yan, X., Yu, P.S., Han, J., 2005. Substructure similarity search in graph databases. In: Proc of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, pp. 766–777.
Yu, T., Zhang, Y., Lin, K.-J., 2007. Efficient algorithms for web services selection with end-to-end QoS constraints. ACM Trans. Web 1 (1) 6-es.

Yuan, Y., Wang, G., Chen, L., Wang, H., 2015. Graph similarity search on large uncertain graph databases. Int. J. Very Larg. Data Bases 24 (2), 271–296.
Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z., 2003. Quality driven web services composition. In: Proc of 12th International Conference on World Wide Web (WWW 2003), New York, NY, USA, pp. 411–421.
Zhang, L., Zhang, B., 1999. A geometrical representation of mcculloch-pitts neural model and its applications. IEEE Trans. Neural Netw. 10 (4), 925–929.
Zhang, S., Li, S., Yang, J., 2009. GADDI: distance index based subgraph matching in biological networks. In: Proc of 12th International Conference on Extending Database Technology (EDBT 2009), Saint Petersburg, Russia, pp. 192–203.
Zhao, P., Han, J., 2010. On graph query optimization in large networks. In: Proceedings of the VLDB Endowment (PVLDB), 3, pp. 340–351.
Zheng, Z., Lyu, M.R., 2015. Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints. IEEE Trans. Comput. 64 (1), 219–232.
Zhu, Y., Qin, L., Yu, J.X., Cheng, H., 2012. Finding top-k similar graphs in graph databases. In: Proc of 15th International Conference on Extending Database Technology (EDBT 2012), Berlin, Germany, pp. 456–467.
Zou, G., Lu, Q., Chen, Y., Huang, R., Xu, Y., Xiang, Y., 2014. QoS-Aware dynamic composition of web services using numerical temporal planning. IEEE Trans. Serv. Comput. 7 (1), 2–15.

Jintao Wu is a Ph.D. student at the School of Computer Science and Technology of Anhui University. His-current research interests include service computing, business process management, cloud computing, computer vision, image processing, and deep learning.

Xing Guo received his Master in Computer Science in 2009 and Ph.D. in 2013 both from the Anhui University, China. He is a Lecturer in the School of Computer Science and Technology at Anhui University. His-research interests include computer vision, image processing, service computing, and cloud computing.

Guijun Yang received the Ph.D. degree in cartography and geographic information system from State Key Laboratory of Remote Sensing Science, Institute of Remote Sensing Applications (IRSA), Chinese Academy of Sciences (CAS), Beijing, China, in 2008. Currently, he is a Research Associate with the National Engineering Research Center for Information Technology in Agriculture (NERCITA), Beijing, China. His-research interests include radiative transfer modeling, imagery simulation, atmospheric correction, quantitative inversion, big data, and cloud computing.

Shuhui Wu received the master's degree in computer technology from the School of Computer Science and Technology, Anhui University, in 2018, where she is currently pursuing the Ph.D. degree at Hefei University of Technology. Her current research interests include computer vision, image processing, robotics, and cloud computing.

Jianguo Wu received the Ph.D. degree in Computer Science and Technology from the Beijing Institute of Technology, Beijing, China, in 1998. He is currently a Professor with the School of Computer Science and Technology at Anhui University, Anhui, China. He has authored or co-authored numerous technical articles in well-known international journals and conferences. His-research interests include intelligent CAD / EDA, identification technology, embedded system, big data, and cloud computing.