Software Quality Journal (2022) 30:757–779
https://doi.org/10.1007/s11219-021-09579-6

Does PageRank apply to service ranking in microservice regression testing?

Lizhe Chen[1] · **Ji Wu**[1] · **Haiyan Yang**[1] · **Kui Zhang**[1]

Accepted: 17 November 2021 / Published online: 24 January 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract
Regression testing is required in each development iteration of microservice systems. Test case prioritization, which improves the fault detection rate by optimizing the execution order of test cases, is one of the main techniques to optimize regression testing. Existing test case prioritization techniques mainly rely on artifacts such as codes and system models, which are limited to microservice systems with service autonomy, development method diversity, and large service scale. This paper proposes a test case prioritization approach based on service ranking referred to as TCP-SR. TCP-SR ranks the services based on API gateway logs. The weights of test cases are calculated with the result of service ranking, which could be used to order test cases with single-objective and multi-objective strategies. To evaluate the effectiveness of TCP-SR, the empirical study based on four microservice systems is presented. The results show that the fault detection rate of TCP-SR is almost twice as high as that of the random prioritization technique, and almost the same as the prioritization technique based on WS-BPEL but requires much less prioritization time cost.

Keywords Microservice · Regression testing · Test case prioritization · PageRank · API gateway

1 Introduction

Microservice is a popular architecture pattern of current cloud applications, for which services are built in their own lifecycle around business domains, and cooperate with each other through lightweight communication mechanism (Fan & Ma, 2017; Larrucea et al., 2018; Lewis & Fowler, 2014). Microservice requires a high degree of autonomy of services (Newman, 2015), that is, each service can be independently constructed and deployed, so as to meet the requirements of rapid expansion and smooth upgrade. Faults may be introduced in every development iteration of microservice systems, so regression testing is

✉ Ji Wu
wuji@buaa.edu.cn

Lizhe Chen
by1506105@buaa.edu.cn

[1] School of Computer Science and Engineering, Beijing University of Aeronautics and Astronautics, Beijing 100191, China

needed to ensure the quality after changes (Yoo & Harman, 2012). The common strategy of regression testing is to rerunning all of the previously test cases, which also indicates high testing costs. For microservice systems, rapid iteration and large scale of services will lead to the sharp increase in costs of retest-all strategy. For example, the WeChat system has tens of thousands of services, and it may cost several months with retest-all strategy (Gao et al., 2019). So, lots of techniques such as test case prioritization, test suite minimization, and test case selection are proposed to optimize regression testing and reduce the testing costs (Roberto, 2020).

Test case prioritization (TCP) concerns ordering test cases for early maximization of some desirable properties, and the common objective is to increase the fault detection rate (Yoo & Harman, 2012). Aiming at this goal, test cases with high probability of detecting faults are run first to reveal as more faults as possible with the same scale. According to the existing research (Catal & Mishra, 2013; Mece et al., 2020; Mohd Shafie & Wan Kadir, 2018; de S. Campos Junior et al., 2017), most TCP techniques order test cases by establishing the coverage relationship between test cases and artifacts such as requirements, design models, and codes. Such TCP techniques are composed of two stages commonly: analysis and prioritization. In analysis stage, the artifacts are processed to retrieve information such as the coverage, critical ranks, and change impacts, etc. In prioritization stage, test cases are associated with such information and prioritized by the relationship data. Such TCP techniques assume that the artifacts are complete, consistent, and easy to obtain, and that the relationship data are well maintained to ensure the availability in continuous integration environments (Qiu et al., 2014; Zhongsheng, 2010).

However, when artifact-based TCP techniques applied in microservice regression testing, the following challenges are encountered: (1) TCP techniques based on codes are coupled with specific programming language and development framework (Qiu et al., 2014; Zhongsheng, 2010). When applied in the microservice systems with multiple implementation techniques, the technical costs will greatly increase. (2) TCP techniques based on specifications and design models are closely related to modeling perspective, granularity, method, and even security strategy (Qiu et al., 2014). When applied in microservice systems developed by different teams, it is difficult for testers to obtain complete specification and models to order test cases. (3) Most TCP techniques need to generate and maintain the relationship data between artifacts and test cases, which are used to transfer the priority of artifacts to test cases. When applied in the microservice systems composed of large-scale services, these data may become too large and bring unacceptable costs to maintain.

We notice that a large amount of API gateway layer logs can be collected in each iteration with the continuous delivery of microservice systems. API gateway layer is a key component for centralized distribution of service requests (Newman, 2015). It records every API request when microservice systems run, including information such as requester, responder, time, status code, etc. By statistical analysis of the logs, the frequency of requests from users to services and between services can be obtained. Based on this information, services can be ranked to build a kind of partial ordering relationships, according to which services receiving more requests and sending less requests have prior orders. The partial ordering relationships reflect the dependencies among services, which are also the targets of the artifacts based analysis in existing TCP techniques. If test cases can be prioritized based on the partial ordering relationships, the prioritization of test cases needs not rely on the artifacts such as codes and system models.

This paper proposes a test case prioritization approach referred to as TCP-SR for microservice regression testing. First of all, in order to model the invocation relationships between services, the request directed graph (RDG) is defined, which takes services as nodes and

requests as directed edges with the probabilities as the weights. And then the algorithm for generating RDG from API gateway layer logs is proposed. Secondly, it is noticed that the transmission of service weight by invocations is similar to the transmission of web page importance by hyperlink jumps (Wu & Wei, 2007; Richardson & Domingos, 2002), an algorithm based on PageRank is designed to rank services based on RDG. Thirdly, since microservice test cases can be abstracted as service vectors (Canfora & Di Penta, 2009), a quantitative model is established to transfer service ranks to test case prioritizations. Furthermore, considering the multi-objective prioritization requirements and the influences of business hierarchical characteristics of some microservice systems on service ranking, four prioritization strategies are proposed to ensure the adaptability of TCP-SR to various microservice systems. In order to validate TCP-SR, an empirical study is conducted based on four microservice systems with different domains, scales, and business structures. The results show that the fault detection rate of TCP-SR is almost twice as high as that of the random prioritization technique, and almost the same as the prioritization technique based on WS-BPEL (Chen et al., 2010) but requires much less prioritization time cost.

The rest of this paper is structured as follows: Sect. 2 presents related work on microservice testing, test case prioritization, and PageRank. Section 3 explains the complete process of TCP-SR, Sects. 4 and 5 describe the empirical analysis, and Sect. 6 discusses conclusions and future work.

2 Related work

2.1 Microservice testing

Microservice testing includes unit testing, service testing, and end-to-end testing (Newman, 2015). Among them, unit testing is used to detect faults of functions or classes, which is usually processed with unit testing tools, such as xUnit (Meszaros, 2007), mockito (Kaczanowski, 2013). The purpose of service testing is to test corporations of services directly without user interfaces, while the end-to-end test focuses on the behavior of the entire system. Due to the challenges brought about by service autonomy, dynamic binding, and access restrictions (Qiu et al., 2014), service testing and end-to-end testing are difficult handled by traditional testing techniques, which are mainly concerned in research and practice (Newman, 2015). The scope of microservice regression testing concerned in this paper also includes service regression testing and end-to-end regression testing.

Meanwhile, both service testing and end-to-end testing involve multiple services and their invocations. For example, consumer-driven testing includes consumer services, target services, and stubbed services (Newman, 2015). Therefore, test cases of these two testing scope can be abstracted as service vectors (Canfora & Di Penta, 2009; Li et al., 2008; Khan & Heckel, 2011; Li et al., 2012; Liu et al., 2007), as the basis of test case prioritization.

2.2 Test case prioritization

The formal definition of test case prioritization problem is as follows (Horváth et al., 2019):

Definition 2.1. Given T, a test suite, PT, the set of all possible permutations of T, and f, a function that determines the performance of a given prioritization from PT to real numbers. Find $T' \in PT$:

$$s.t.(\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')] \tag{1}$$

As definition 2.1 shows, the key point of test case prioritization aims to quantitatively model the degree to which test cases meet the prioritization objectives, that is, how to design the function f. Then, based on f, T' can be found by global searching, heuristic searching, and other algorithms (Azizi & Do 2018).

Test case prioritization is a hot topic in regression testing research. The existing techniques can be divided into two types, code-based and model-based (Mece et al., 2020; Mohd & Wan, 2018). The code-based methods mainly concern the coverage information of test cases to codes (Li et al., 2012), such as the statement coverage, branch coverage, or path coverage (Horváth et al., 2019). It requires that the source codes of the system under testing must be available and depends on the specific programming languages, which is difficult to archive for microservice systems with service autonomy. The model-based method establishes the relationship data between the test cases and models of system structures or system behaviors, which is used to prioritize test cases based on the criticality, complexity, and defect density of the models (Korel et al., 2005, 2008; Mahdian et al., 2009). Because of the higher level of abstraction, the model-based method is more practical than the code-based method, but it depends on the integrity and consistency of the models, which is also difficult to guarantee in the microservice systems.

In addition, some studies concentrate on prioritization techniques based on historical information or for specific scenarios. In the research of Azizi and Do (2018), test cases are ordered by change historical information in dynamic environment, which is also difficult to maintain in the microservice systems. For embedded systems, a location-based prioritization technique is proposed by Wang et al. (2019), which uses gravitation law for high reliability after modification. In product line testing scenario, a similarity-based prioritization technique is proposed by Al-Hajjaji et al. (2019) with diverse feature interaction coverage. The study analyzed the effectiveness in both real and seeded fault detection. In the paper presented by Ouriques et al. (2018), different existing prioritization techniques are compared in the context of model-based testing using replicated study to investigate influence of test case size on the efficiency of fault detection rate ability. However, such techniques are only suitable for special scenarios and cannot be applied in microservice regression testing.

In recent years, with the development of machine learning and deep learning, many researchers apply such techniques to test case prioritization (Mece et al., 2020). The typical test case prioritization techniques are based on neural network, genetic algorithm, and Bayesian network. Among them, the techniques based on neural network mainly extract features from system specifications (Gökçe & Eminli, 2014) and historical testing records to prioritize test cases (Spieker et al., 2017). Genetic algorithm–based methods build some features such as module failure rate (Abele & Göhner, 2014), code coverage (Konsaard & Ramingwong, 2015) of test case prioritizations, and iteratively calculate the best prioritizations. The techniques based on Bayesian network mainly unify the white box information such as code changes and code coverage into a model to prioritize test cases (Zhao et al., 2015). Most of these technologies still rely on codes, specifications, and system models as inputs, so they are also not practical for the microservice systems.

Compared with TCP techniques discussed above, our approach does not rely on development artifacts as inputs, and proposes a full automated process instead of the tasks such

as artifacts collection, consistency checking, information extraction, and modeling, so as to avoid the challenges in microservice regression testing.

2.3 PageRank

PageRank is a machine learning algorithm to rank web pages based on their link structure (Wu & Wei, 2007; Richardson & Domingos, 2002), which has been applied to Google search. It is derived from the citation analysis, that is, articles citing other literature will pass on their own importance to the cited literature. Since the link structure of web pages can be abstracted as a directed graph, PageRank algorithm can be used to solve the problem of ranking nodes of directed graphs with weighted edges. PageRank algorithm framework can be defined as follows:

Definition 2.2. Given a directed graph $G = (N, E)$, N is the set of nodes which size is m, and E is the set of edges. $W(e_{ij})$ represents the normalized weight of an edge e_{ij} from node n_i to node n_j, which satisfies $\sum_{j=1}^{m} W(e_{ij}) = 1$. Let d represent the damping coefficient, the ranking weight value of node n_i can be calculated as follows:

$$PR(n_i) = (1 - d)/m + d \sum_{j=1}^{m} PR(n_j)W(e_{ji}) \tag{2}$$

The damping coefficient d is introduced to avoid "weight leakage" and "weight sinking" problems (Richardson & Domingos, 2002), and its empirical value is 0.85. When $W(e_{ij})$ represents the normalized degree of node n_i, formula 2 is the original form of PageRank. Since the iterative calculation process of formula 2 can be transformed into a Markov process, the ranking results will converge to stable values and have nothing to do with the initial values (Richardson & Domingos, 2002).

PageRank algorithm is mainly used in search engines at present, and related researches include two aspects: one is to optimize the calculation process of the algorithm itself and improve the performance of the algorithm, such as the power algorithm which uses the power method iteration to calculate the matrix eigenvector (Wu & Wei, 2007). On the other hand, the algorithm is expanded to adapt to more complex scenarios, such as solving user clustering problem (Naik et al., 2018) and entity ranking problem in knowledge graph (Diefenbach & Thalhammer, 2018). And some researchers have also applied PageRank to software engineering. Zhang et al. (2017) used PageRank in code debugging. They compute the spectrum information by considering the contributions of different testings and improve the effectiveness of spectrum-based fault localization. Kim et al. (2013) apply PageRank to root cause detection in service-oriented architectures. They propose MonitorRank to rank possible root causes for monitoring teams to investigate. Mirshokraie et al. (2015) concentrate on mutation testing and propose the notion of FunctionRank, a dynamic variant of PageRank, to rank functions in terms of their relative importance.

PageRank can derive the global web page ranking from the pairwise links between web pages, which is consistent with the global service ranking derived from pairwise requests between services. Therefore, this paper proposes a service ranking algorithm to generate service ranking weight values as the basis for test case prioritization. To the best of our knowledge, this work is the first research to apply the PageRank algorithm for microservice regression testing prioritization.

3 Methodology

The inputs of this method mainly contain API gateway layer logs and test cases, and the output is the ordered test cases. The steps include RDG generation, service ranking, and test case prioritization, as shown in Fig. 1. Each step is presented in detail as follows.

3.1 Request directed graph generation

This step builds RDG from API gateway layer logs. A request corresponding to a record in API gateway layer logs comes from a user or a service. Each record contains the IP address or domain name of the requester and the responder. Based on this information and service registries (such as zookeeper, Eureka), the service ID of the requester and the responder can be determined. Let s_0 represent the user, and $\{s_1,s_2,s_3...s_n\}$ represent services of the microservice system under testing. Given two services s_i and s_j $(i \neq j)$, r_{ij} represents the request from s_i to s_j, and $count(r_{ij})$ represents the frequency of r_{ij} in logs. According to the law of large numbers, the probability $p(r_{ij})$ of s_i depending on s_j can be calculated as follows:

$$p(r_{\mathrm{ij}}) \approx count(r_{\mathrm{ij}}) / \sum_{k-0}^{n} count(r_{ki}) \tag{3}$$

It is noticed that the sum of all $p(r_{ij})$ may be less than 1 (for example, with the data caching and circuit breaker (Azizi & Do, 2018b), some services may not process requests and directly return the cached data or the default data). The probability of self-dependence $p(r_{ii})$ is calculated as follows:

$$p\left(r_{\mathrm{ii}}\right) = 1 - \sum_{j-0}^{i} p\left(r_{ij}\right) - \sum_{j-i+1}^{n} p(r_{ij}) \tag{4}$$

Based on formulas 3 and 4, RDG can be defined as follows:

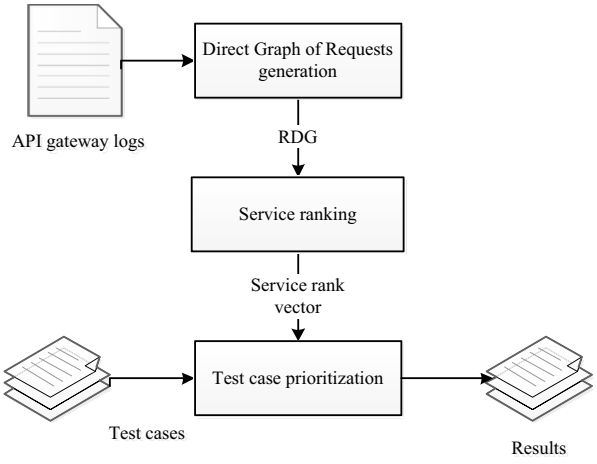

Fig. 1 Sketch map of the method

Definition 3.1. Request directed graph (RDG) is a tuple (N,E). $N=\{s_0\}\cup\{s_1,s_2,s_3\ldots s_n\}$ represents the set of nodes, and $E=\{r_{ij}|\ p(r_{ij})>0,\ 0\le i,j\le n\}$ represents the set of directed edges where r_{ij} represents an edge from node s_i to node s_j with the weight $p(r_{ij})$, which is calculated by formulas 3 and 4.

By definition 3.1, RDG can be generated from API gateway layer logs with the algorithm shown in Algorithm 1. This algorithm scans the records in logs one by one, and for each record, queries the service ID and updates the nodes and directed edges of RDG (line 5 to 11). Where, the function *query_services* retrieves the requester and the responder information from the service registries, and the function *find_or_new* finds the node or edge in the directed graph, or adds a new element while not found. At the end of scanning, the weight of edge *Probability* is updated based on formula 3 (line 14 to 17), and some self-pointing edges are created based on formula 4 (line 18 to 24). Figure 2 shows an example of RDG.

Algorithm 1: RDG generation algorithm

Declaration: *generateDG(logs, RDG)*.
Parameters: *logs* (**in**); *RDG* (**out**).

```
1  RDG.N←{ s0}
2  for each log ∈logs do
3     si, sj = query_services(log)
4     if si ≠ null then
5        RDG.N.find_or_new(si)
6     else
7        si ← s0
8     end if
9     RDG.N.find_or_new(sj)
10    sj.Count←sj.Count + 1
11    rij ←RDG.E.find_or_new(si, sj)
12    rij.Count←rij.Count + 1
13 end for
14 for each r ∈RDG.E do
15    si←r.Sender
16    r. Probability←r. Count / si.Count
17 end for
18 for each s ∈RDG.N do
19    if s.sumRequestsCount < s.Count then
20       r ←RDG.E.find_or_new(s, s)
21       r.Count←s.Count- s.sumRequestsCount
22       r. Probability←r. Count / s.Count
23    end if
24 end for
```

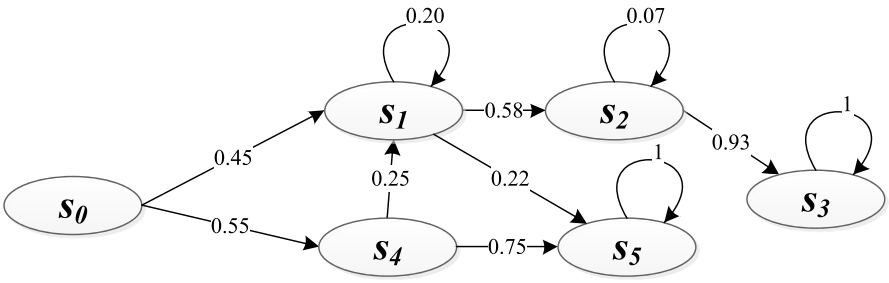

Fig. 2 An example of RDG

3.2 Service ranking

This step ranks the services as the nodes of RDG. The initial ranking weight value of each service can be set as the same (for example the integer 1). Then, the values of nodes will be transmitted from the start nodes to end nodes through the directed edges, which leads to changes in service ranking. Until the values of nodes become stable, the repeated transmission process is stopped and services can be ranked by the values of nodes.

According to the definition of RDG, the weight $p(r_{ij})$ of an edge r_{ij} can be seen as the proportion of node n_i transfer its ranking weight to node n_j and $\sum_{j=0}^{n} p(r_{ij}) = 1$. So, based on definition 2.2, $p(r_{ij})$ can be used as $W(e_{ij})$ in formula 2. At the same time, it is noticed that some nodes have not any edges starting from them (for example s_3 and s_5 in Fig. 2), which will lead to zero values for other nodes' ranking weight. Therefore, it is necessary to introduce a damping factor d to indicate that any node in RDG may randomly transfer its value to other nodes with a fixed probability, which ensures the correct convergence of ranking weight values. To sum up, the service ranking algorithm is defined as follows:

Definition 3.2. Given an instance of RDG and a damping factor $d=0.85$, the ranking weight value of node n_i can be calculated as follows:

$$SR(s_i) = (1-d)/(n+1) + d\sum_{j=0}^{n}\left(SR(s_j)p(r_{ji})\right) \tag{5}$$

Let a vector $X=[SR(s_0),SR(s_1)\ldots\ SR(s_n)]^T$ represent all of the ranking weight values, and an $n+1$-dimensional full-1 vector e represent the initial values of all nodes. Then, based on the power method iteration, X can be calculated as follows:

$$X = \frac{limA^n e}{n \to \infty} \tag{6}$$

where matrix $A=dP+ee^T(1-d)/(n+1)$, and matrix P is an $n+1$-order square matrix and the element of P in row i, column j is $p(r_{ji})$.

Algorithm 2 shows the service ranking algorithm implemented based on formulas 5 and 6. Firstly, X is initialized and matrix A is calculated (line 1 to 2). Then, the loop of matrix power operation is processed (line 3 to 11), which is break out by the condition that the difference between X calculated before and after in one iteration is less than the convergence threshold. At last, the head value corresponding to s_0 is removed, and X is

returned. For the RDG in Fig. 2, given $\varepsilon = 0.0001$, nine iterations were processed to run the service ranking algorithm. The service ranking vector is $X = [0.3062, 0.3200, 2.6863, 0.2201, 2.3172]$. Obviously, s_3 has the maximal ranking weight value, and s_4 has the minimal value.

Algorithm 2: Service ranking based on power method

Declaration: *caculateServiceRank*(*RDG*, ε , *X*).
Parameters: *RDG* (**in**); ε (**in**); *X* (**out**).

```
1   X_pre ← e
2   A ← RDG. getA()
3   do
4      X ← A X_pre
5      if |X - X_pre | < ε then
6         X.removeHead()
7         return
8      else
9         X_pre ← X
10     end if
11  while(True)
```

3.3 Test case prioritization based on service ranking vector

In microservice regression testing, a test case of service testing or end-to-end testing can be abstracted as a service vector v. To order service vectors, a quantitative model is needed to calculate the ordered weight value based on the service ranking vector X. The quantitative model should meet the following two requirements:

1. The model needs to highlight the extreme value of service ranking weights in v, which can be the maximal value or the minimal value. Otherwise, the services corresponding to the extreme value, which are more likely to be a major concern in regression testing than other services, may be ignored.
2. The model should be independent of the v's size. Otherwise, the test cases corresponding to short vectors with more important services may be masked by the ones corresponding to long vectors with less important services.

Above requirements can be met by two stages of ordering. In the first stage, service vectors are divided into different levels based on the extreme value of service ranking weights, which meets requirement 1. In the second stage, for each level, service vectors are ordered by the average value of service ranking weights, which meets requirement 2. Therefore, the quantitative model can be defined as follows:

Definition 3.3. Given a service *s and* a service vector v, let $L(s) \in \{1,\ldots,n\}$ represent the order assigned to s, $M(v)$ represent the service with extreme value of service ranking weights in v, $\mu(v)$ represent the mean value of service ranking weights in v, and $||X||$ is the 1-norm form of X. The ordered weight value of v can be calculated as follows:

$$PR(v) = (L(M(v)) - 1)/n + \mu(v)/(n\|X\|) \tag{7}$$

Formula 7 shows that v is divided into an interval of $1/n$ in length according to $M(v)$, where the ordered weight value is calculated proportionally based on $\mu(v)$. For example, based on the service ranking vector in Sect. 3.2, given $v=[s_1,s_2,s_3]$, if the extreme value is considered as the maximal value, $M(v)=s_3$, $L(s_3)=5$, $\mu(v)=1.1042$, and $PR(v)=0.8378$; if the extreme value is considered as the minimal value, $M(v)=s_1$, $L(s_1)=2$, $\mu(v)=1.1042$, and $PR(v)=0.2378$.

Since early maximization of structural coverage will also increase the chance of early maximization of fault detection, prioritization techniques actually aim to maximize early coverage (Yoo & Harman, 2012). Different from treating all services equally, service ranking builds priority to the degree of service dependence and users' concentration. Early coverage of high priority services can make testing more targeted and improve the fault detection rate. Thus, $PR(v)$ can be used as the basis for single-objective or multi-objective test case prioritization, proposed as the four test case strategies shown in Table 1. The first two are single-objective strategies, while the latter two multi-objective strategies also consider the diversity between test cases.

In Table 1, for the single-objective strategies MaxR and MinR, $PR(v)s$ of test cases are considered respectively maximizing and minimizing. This is because some microservice systems may have business hierarchical characteristics, for example, in Fig. 2, service s_1, s_2, and s_4 emphasizing on application functions can be called as application layer services, while service s_3 and s_5 emphasizing on resource management can be called as facility layer services. In such microservice systems, requests between different levels of services present a relatively fixed distribution law. That is, most of the application layer services send requests to the facility layer services, but in turn, there will be few or no related requests. After service ranking, the weight values of application layer services are always less than those of facility layer services. When some versions of the microservice system under testing emphasize on application layer services and MaxR is applied, test cases for facility layer services will be given high priority, which may lead to reduction of fault detection rate. Therefore, it is better to choose suitable prioritization strategy for versions of microservice systems with business hierarchical characteristics.

Table 1 Test case prioritization strategies

Type	Name	Description
Single-objective	Maximal service ranking (MaxR)	Test case is given higher priority when its $PR(v)$ is greater
	Minimal service ranking (MinR)	Test case is given higher priority when its PR(v) is less
Multiple-objective	Maximal service ranking with diversity (MaxR&D)	Test case is given higher priority when its PR(v) the diversity between it and the previous ordered test case is greater
	Minimal service rank with diversity (MinR&D)	Test case is given higher priority when its PR(v) and the diversity between it and the previous ordered test case is greater

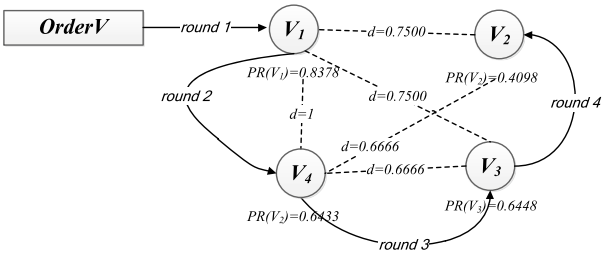

Fig. 3 An example of heuristic search process

For the multiple-objective strategies MaxR&D and MinR&D, the diversity between test cases is considered with MaxR and MinR to cover all services as soon as possible. Meanwhile, by adding the diversity, the regular distribution of MaxR and MinR will also be disturbed, and the correlation between ranking results and business hierarchical characteristics will be weakened. Therefore, MaxR&D and MinR&D should be more suitable than single-objective strategy.

The diversity between two test cases can be measured by the Jaccard distance (Prado et al., 2020) of two corresponding service vectors:

Definition 3.4. Given service vector v_1 and v_2, let $Set(v)$ represent the set of services in v, the diversity between v_1 and v_2 can be calculated as follows:

$$d(v_1, v_2) = \frac{\left|Set(v_1) \cup Set(v_2)\right| - \left|Set(v_1) \cap Set(v_2)\right|}{\left|Set(v_1) \cup Set(v_2)\right|} \tag{8}$$

Based on formula 7 and 8, MaxR&D and MinR&D can be implemented using heuristic search methods, and the pseudo codes of the algorithm for MaxR&D is given in Algorithm 3. The algorithm searches for the next test case (line 3 to 4) heuristically based on the previously ordered test case. The search goal is to maximize the sum of $PR(v)$ and the diversity (line 5 to 18), and the searching loop ends when the size of ordered test cases reaches the specified scale (line 2) to meet the actual testing resource limitation.

Following the example in Sect. 3.2, suppose there are four test paths need to be ordered. The corresponding list of service vectors is $V=[v_1,v_2,v_3,v_4]$, where $v_1=[s_1,s_2,s_3]$, $v_2=[s_4,s_1]$, $v_3=[s_1,s_5]$, $v_4=[s_4,s_5]$. Let *Scale* be 100%, and run the algorithm *heuristicPrioritization* to output *OrderV*. The whole heuristic search process can be showed in Fig. 3. In the first round of the cycle, *OrderV* is empty and choose vector v_1 which has the maximal *PR* value. In the rest rounds, the tail vector of *OrderV* is set as the start node to find the vector which has the maximal sum of *PR* value and the diversity value until all vectors have been ordered. Then, the result $OrderV=[v_1,v_4,v_3,v_2]$ can be used to order corresponding test paths.

4 Empirical study

To validate the technique proposed in this paper, we performed an empirical study based on four microservice systems. The study investigates three research questions as follows:

RQ1: Is TCP-SR significantly better than random prioritization to improve fault detection rate?

Algorithm 3: Heuristic search algorithm with MaxR&D

Declaration: *heuristicPrioritization*(*V*, *Scale*, *OrderV*).
Parameters: *V*, list of service vectors (**in**); *Scale,* size threshold (**in**); *OrderV*, list of ordered service vectors (**out**).

```
1  originalScale ← V.length
2  while OrderV.length / originalScale < Scale do
3     pre_v←OrderV.getTail()
4     ComppareList←∅
5     for each v ∈ V do
6        comp←PR(v)
7        if pre_v≠ null then
8           comp←comp + d(v, pre_v)
9        end if
10       ComppareList.append(comp)
11    end for
12    index ← -1
13    index←ComppareList.getMaxIndex()
14    if index > 0 then
15       OrderV.append(V.getAt(index))
16       V.removeAt(index)
17    end if
18 end while
```

Random prioritization is a benchmark method in the study of test case prioritization. Many test case prioritization techniques are compared to this method to assess its effects (Qiu et al., 2014). So, the fault detection rate of TCP-SR and that of randomly prioritization are compared to show whether TCP-SR can improve test efficiency.

RQ2: Is TCP-SR more efficient than the typical system model–based TCP techniques in microservice regression testing?

Theoretically, TCP-SR does not rely on artifacts such as specifications, design models, and source codes. It is completely decoupled from the design and implement techniques of the system under testing, which leads to the scalability is obviously better than TCP techniques based on artifacts. In our experiments, if the fault detection rates of TCP techniques are close, the efficiency can be compared by the time costs of the prioritization process. The technique which costs less time is more efficient. Considering the wide application of Web Service Business Process Execution Language (WS-BPEL), TCP-SR is compared with a WS-BPEL-based technique (Chen et al., 2010), which prioritizes test cases based on the coverage of test cases to WS-BPEL models of systems under testing.

RQ3: When applied to microservice systems with business hierarchical characteristics, will the four prioritization strategies proposed above have different effects.

In Sect. 3.3, it is discussed that business hierarchical characteristics of some microservice systems will affect service ranking and four prioritization strategies are proposed to fit different scenarios. To support the above analysis, experiments should be processed to compare the effects of these strategies in microservice systems with and without business hierarchical characteristics.

4.1 Case introduction

Considering technology architecture, scale, and data availability, we carried out the experiments based on cases as follows:

1. m-Ticket: a multi-end ticket system based on SpringBlade (an open source microservice framework available at https://github.com/chillzhuang/SpringBlade) provides ticket services in various fields such as transportations, accommodations, tourist attractions

and movies, supporting service management, monitoring, and tracing. The changes of different versions are mainly about the horizontal expansion of different businesses, for example, the first version release only supports the transportations and lodging ticket business, the second version increases the tourist attractions ticket business, and so on. There are no obvious business hierarchical characteristics among services.

2. z-Shop: a mobile-oriented mall system based on Zheng (an open source microservice framework available at https://github.com/shuzheng/zheng) provides one-stop management services for goods, stores, content promotion, orders, logistics, etc. The version changes are mainly about the vertical expansion of businesses, that is, the early versions emphasize on user management, commodity management, order management, and other supporting services, while the later versions emphasize on application services such as store customization, marketing activities, intelligent recommendation, etc.
3. JOA: an OA system based on Spring Cloud provides comprehensive information display, document circulation, process approval, plan management, organization personnel management, contract management, fund management, and material management services for the organization with multiple departments and secret levels. Due to the domain rules and data security, JOA is jointly developed by several teams and has the characteristics of business hierarchy. However, due to the parallel development at different business levels, each version does not emphasize on a certain level.
4. DIS: a domain-specific information intelligence service system provides data crawling, data governance, intelligence analysis, and intelligence application services. In DIS, various businesses are developed continuously as services running on the intelligence information infrastructures. Like JOA, DIS is jointly developed by several teams and has the characteristics of business hierarchy.

Table 2 shows the total numbers of services, logs, versions, test cases, and faults of each case (relevant data are available at https://gitee.com/maple_clz/cases-for-microservice-regression-testing). Faults of each version are collected from the test reports of service testing and end-to-end testing. Based on these data, a posteriori method is used to setup experiments, that is, when the execution results of all test cases are known, different test case prioritization techniques are evaluated and compared.

4.2 Evaluation metrics

Based on the literature (Yoo & Harman, 2012), the average percentage of fault detection (APFD) is used to measure the efficiency of test cases ordered. Let n represent the size of test case set T and m represent the size of fault set F. TF_i represents the order value of the test case by which the ith fault is found. Then, APFD is calculated as shown in formula 9. Obviously, the greater the APFD value, the earlier the fault is detected, and the more efficient the test case sequence is.

Table 2 Case application properties

Object	Services	Logs	No. of versions	No. of test cases	No. of faults
m-Ticket	61	40,000	5	1182	339
z-Shop	43	20,000	4	913	227
JOA	605	50,000,000	9	13,356	2783
DIS	1004	420,000,000	15	54,268	15,208

$$APFD = 1 - \sum_{i=1}^{m} TF_i/(mn) + 1/(2n) \tag{9}$$

If T_s, a subset of T, is used for testing, then the number of faults found by T_s is $m_s \leq m$, and if the TF value of undetected faults is n, then the APFD calculation formula is adjusted to:

$$APFD = 1 - \sum_{i=1}^{m_s} TF_i/(mn) - (m - m_s)/m + 1/(2n) \tag{10}$$

In addition, the time of the prioritization process is also recorded to measure the costs of a TCP technique. Since the intention of TCP is to reduce testing cost, the less prioritization time cost, the more practical the TCP technique is.

4.3 Experiment setup

We designed three experiments to answer the above three RQs. In the experiments for RQ1 and RQ2, the strategy MaxR&D with TCP-SR is applied. At the same time, because JOA and DIS involve multiple team development, the iteration of versions is relatively complex and the scale is relatively large; the system models are difficult to generate with WS-BPEL, so the experimental validation of RQ2 is mainly based on m-Ticket and z-Shop.

Set *TR* as the randomly ordered test cases, *TW* as the test cases ordered by WS-BPEL-based technique, and *TS* as the test cases ordered by TCP-SR. The experimental steps are designed as follows:

• Experiment 1: Comparison of TCP-SR and random prioritization

1. For each case, from the k-th ($k \geq 2$) version, with the threshold of prioritization scale 25%, 50%, 75% and 100%, generate *TR*, respectively
2. For each case, from the k-th ($k \geq 2$) version, based on the logs of k-1-th version and strategy MaxR&D, with the threshold of prioritization scale 25%, 50%, 75%, and 100%, generate *TS*, respectively
3. For each test case set, calculate its APFD values based on formula 9 and 10 with the fault records
4. To answer RQ1, compare the APFD values of *TR* and *TS*

• Experiment 2: Comparison of TCP-SR and the prioritization technique based on WS-BPEL

1. For m-Ticket and z-Shop, from the k-th ($k \geq 2$) version, with the threshold of prioritization scale 25%, 50%, 75%, and 100%, generate *TW* respectively, and record the time of the prioritization process
2. For m-Ticket and z-Shop, apply the step 2 of experiment 1, and record the time of the prioritization process
3. For each test case set, calculate its APFD values based on formula 9 and 10 with the fault records
4. To answer RQ2, compare APFD values and the prioritization time cost of *TW* and *TS*

• Experiment 3: Comparison of four prioritization strategies of TCP-SR

1. For each case, from the k-th ($k \geq 2$) version, based on the logs of k-1-th version and four strategies, with the threshold of prioritization scale 100%, generate *TS-MaxR*, *TS-MinR*, *TS-MaxR&D*, and *TS-MinR&D* respectively
2. For each test case set, the corresponding APFD values are calculated based on formula 9 with the fault records
3. Compare the trends of APFD values changing with versions of four strategies to answer RQ3

5 Results and discussion

5.1 Data and analysis

The data of experiment 1 are shown in Table 3. As shown in the table, the APFD values of *TR* ranges from 0.22 to 0.56, while the APFD values of *TS* ranges from 0.58 to 0.93.

Table 3 The APFD values of *TS* and *TR* in experiment 1

Subjects/versions		Prioritization Scale Ratio/APFD							
		25%		50%		75%		100%	
		TS	*TR*	*TS*	*TR*	*TS*	*TR*	*TS*	*TR*
m-Ticket	v2	0.63	0.34	0.70	0.39	0.73	0.45	0.73	0.45
	v3	0.65	0.33	0.76	0.38	0.80	0.38	0.80	0.38
	v4	0.65	0.36	0.75	0.46	0.83	0.49	0.83	0.49
	v5	0.73	0.41	0.82	0.45	0.82	0.46	0.82	0.46
z-Shop	v2	0.60	0.32	0.65	0.45	0.76	0.47	0.76	0.47
	v3	0.62	0.30	0.68	0.38	0.70	0.43	0.74	0.43
	v4	0.59	0.29	0.62	0.37	0.69	0.41	0.69	0.41
JOA	v2	0.66	0.34	0.76	0.46	0.81	0.51	0.81	0.51
	v3	0.65	0.43	0.76	0.52	0.79	0.53	0.79	0.53
	v4	0.62	0.39	0.75	0.42	0.78	0.42	0.78	0.42
	v5	0.60	0.35	0.73	0.41	0.75	0.41	0.75	0.41
	v6	0.58	0.30	0.70	0.45	0.73	0.45	0.73	0.45
	v7	0.71	0.47	0.77	0.48	0.81	0.48	0.81	0.48
	v8	0.60	0.45	0.73	0.49	0.78	0.50	0.78	0.50
	v9	0.62	0.38	0.75	0.38	0.79	0.43	0.79	0.43
DIS	v2	0.70	0.29	0.75	0.39	0.82	0.45	0.86	0.49
	v3	0.68	0.40	0.69	0.42	0.79	0.46	0.84	0.55
	v4	0.59	0.31	0.72	0.51	0.80	0.53	0.87	0.56
	v5	0.65	0.27	0.68	0.47	0.85	0.49	0.89	0.53
	v6	0.67	0.28	0.69	0.38	0.89	0.35	0.93	0.46
	v7	0.67	0.33	0.73	0.43	0.79	0.49	0.83	0.52
	v8	0.63	0.44	0.72	0.48	0.80	0.52	0.84	0.55
	v9	0.65	0.35	0.67	0.50	0.77	0.54	0.83	0.56
	v10	0.66	0.22	0.77	0.35	0.83	0.43	0.85	0.50
	v11	0.72	0.33	0.78	0.40	0.85	0.43	0.88	0.49
	v12	0.68	0.31	0.70	0.42	0.82	0.45	0.87	0.51
	v13	0.67	0.41	0.79	0.45	0.86	0.49	0.88	0.53
	v14	0.64	0.38	0.75	0.40	0.81	0.44	0.86	0.48
	v15	0.68	0.27	0.73	0.37	0.83	0.42	0.85	0.47

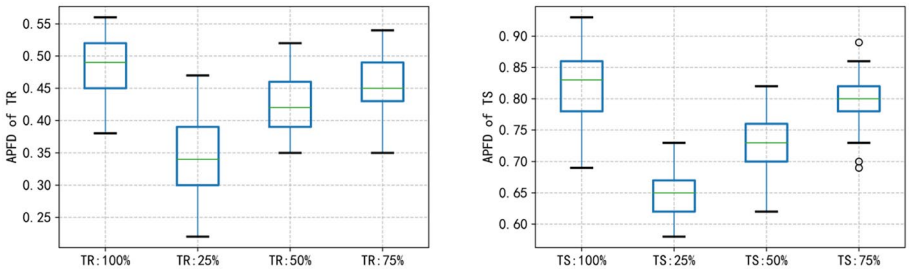

Fig. 4 Distribution of APFD values of *TR* (left) and *TS* (right) in all cases

Apparently, the APFD lower bound of *TS* is higher than the APFD upper bound of *TR*, which indicates that TCP-SR is better than random prioritization to improve fault detection rate. To intuitively compare the distribution of APFD values, Fig. 4 is drawn from the data. As shown in the two boxplots, for the scale 25%, 50%, 75%, and 100%, the average APFD values of *TR* are 0.35, 0.43, 0.46, and 0.48 while that of *TS* are 0.65, 0.73, 0.79, and 0.82, respectively. That is, the average APFD values of *TS* are almost as twice as those of *TS* for each scale of test cases, which indicates that the comparative advantage of TCP-SR to random prioritization is significant. This is because that the partial ordering relationships of services are mined out and used for test case prioritization rather than processing randomly. The answer to RQ1 is yes.

The APFD values and the prioritization time cost of experiment 2 are shown in Tables 4 and 5 respectively. As shown in Table 4, the APFD values of *TW* range from 0.55 to 0.82, while the APFD values of *TS* range from 0.59 to 0.83. The two ranges are basically the same, which indicates that the ability of the two techniques to improve fault detection rate is similar. Figure 5 is drawn from Table 4, which shows that for each scale, the average APFD values of *TS* are almost the same with those of *TW*. It indicates that the partial ordering relationships of services reflect the service relationships contained in the WSBL models in such two cases. That is, in terms of the fault detection rate, TCP-SR can be applied instead of the WSBL-based TCP technique in our experiments. On the other hand, the prioritization time cost is showed in Table 5, which is also plotted as the boxplots in Fig. 6. From them, the prioritization time cost of TCP-SR is not more than 0.22 min, while those of the WSBL based technique are more than 15 min at least. The time cost of WSBL-based technique mainly involves model generation, checking, updating, and mapping to test cases, while the characteristics of consistency, comprehensibility, integrity, granularity, etc. of the artifacts seriously affected the processing

Table 4 The APFD values of *TS* and *TW* with each scale in experiment 2

Subjects/versions		Prioritization Scale Ratio/APFD							
		25%		50%		75%		100%	
		TS	*TW*	*TS*	*TW*	*TS*	*TW*	*TS*	*TW*
m-Ticket	v2	0.63	0.59	0.70	0.71	0.73	0.76	0.73	0.76
	v3	0.65	0.62	0.76	0.75	0.80	0.82	0.80	0.82
	v4	0.65	0.65	0.75	0.75	0.83	0.81	0.83	0.81
	v5	0.73	0.72	0.82	0.79	0.82	0.81	0.82	0.81
z-Shop	v2	0.60	0.55	0.65	0.66	0.76	0.73	0.76	0.73
	v3	0.62	0.61	0.68	0.63	0.70	0.71	0.74	0.71
	v4	0.59	0.57	0.62	0.63	0.69	0.70	0.69	0.70

Table 5 The prioritization time cost with TCP-SR and the WSBL-based technique in experiment 2

Subjects/ versions		Prioritization Scale Ratio / time cost (min)							
		25%		50%		75%		100%	
		TS	*TW*	*TS*	*TW*	*TS*	*TW*	*TS*	*TW*
m-Ticket	v2	0.13	24.55	0.13	27.37	0.14	31.62	0.14	37.28
	v3	0.17	26.59	0.17	28.33	0.18	29.24	0.18	30.71
	v4	0.19	24.99	0.20	26.17	0.20	29.45	0.20	31.25
	v5	0.21	23.72	0.22	24.65	0.22	25.31	0.22	27.63
z-Shop	v2	0.09	16.37	0.09	18.23	0.10	19.13	0.10	20.22
	v3	0.10	15.63	0.10	16.59	0.11	17.18	0.11	17.82
	v4	0.12	16.50	0.12	17.75	0.13	18.12	0.13	19.03

performance. But with TCP-SR, the time cost mainly involves the log collection and mining, which are implemented automatically completely without maintaining any intermediate data and requiring any other artifacts. With the similar APFD values, the significant differences of prioritization time costs indicate that TCP-SRs are much more efficient than the WSBL-based technique for such two cases in our experiments. The answer to RQ2 is also yes.

The data of experiment 3 is shown in Table 6. In order to show the trends of APFD values changing with versions of four strategies, broken line graphs for each case are shown in Fig. 7. In Fig. 7, for the case m-Ticket with horizontal business expansion, the APFD values of the four strategies are not significantly different. For the case z-Shop with vertical expansion, as the version changes from facility concerned to application concerned, the APFD value of *TS-MaxR* decreases significantly, while that of *TS-Min* increases significantly. This difference is also presented in the case JOA and DIS similarly, but not as large as the case z-Shop. This is consistent with our analysis, that is, when applied to multiple versions of microservice systems with business hierarchical characteristics, the effects of MaxR and MinR will change significantly with services emphasizing on different levels. On the contrary, the APFD values of *TS-MaxR&D* and *TS-MinR&D* did not fluctuate significantly with versions changing in the four cases, indicating that the multi-objective strategy is more adaptable and applicable to all versions.

5.2 Threats to validity

As with most empirical studies, there are some risks to apply the conclusions directly of experiments above, and the threats to validity mainly are manifested in two aspects:

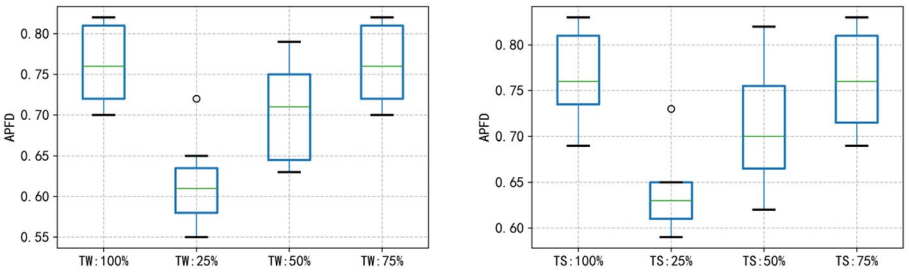

Fig. 5 APFD values of *TW* (left) and *TS* (right) in the case m-Ticket and case z-Shop

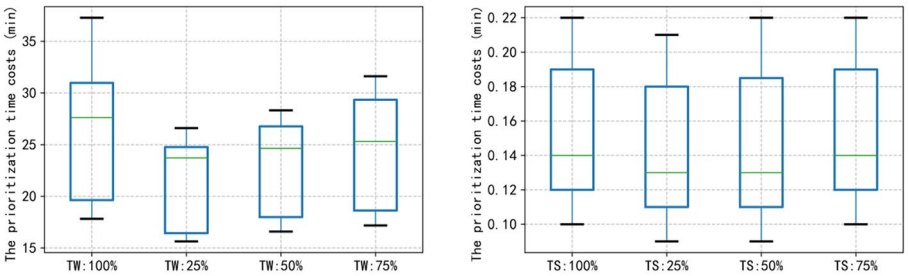

Fig. 6 Distribution of prioritization time cost with TCP-SR in the case m-Ticket and case z-Shop

1. Cases selection. Although the experiment considers factors such as size, complexity, and type when selecting cases, there may be some limitations that do not cover all types of microservice systems. At the same time, the integrity of logs, the way test cases are generated, and whether the faults records are comprehensive or not may also affect the

Table 6 The APFD values with each strategies in experiment 3

Subjects/versions		*TS-MaxR*	*TS-MinR*	*TS-MaxR&D*	*TS-MinR&D*
m-Ticket	v2	0.76	0.64	0.73	0.78
	v3	0.75	0.69	0.80	0.75
	v4	0.77	0.68	0.83	0.82
	v5	0.66	0.78	0.82	0.76
z-Shop	v2	0.84	0.17	0.76	0.71
	v3	0.39	0.56	0.74	0.77
	v4	0.13	0.75	0.69	0.82
JOA	v2	0.77	0.51	0.81	0.73
	v3	0.82	0.47	0.79	0.75
	v4	0.72	0.59	0.78	0.82
	v5	0.68	0.72	0.75	0.81
	v6	0.60	0.77	0.73	0.78
	v7	0.62	0.75	0.81	0.75
	v8	0.46	0.80	0.78	0.79
	v9	0.42	0.84	0.79	0.80
DIS	v2	0.79	0.63	0.86	0.83
	v3	0.78	0.68	0.84	0.80
	v4	0.83	0.65	0.87	0.79
	v5	0.82	0.72	0.89	0.82
	v6	0.85	0.75	0.93	0.85
	v7	0.78	0.69	0.83	0.86
	v8	0.80	0.82	0.84	0.85
	v9	0.80	0.75	0.83	0.80
	v10	0.81	0.73	0.85	0.79
	v11	0.82	0.77	0.88	0.90
	v12	0.81	0.84	0.87	0.85
	v13	0.83	0.83	0.88	0.82
	v14	0.81	0.79	0.86	0.80
	v15	0.82	0.80	0.85	0.83

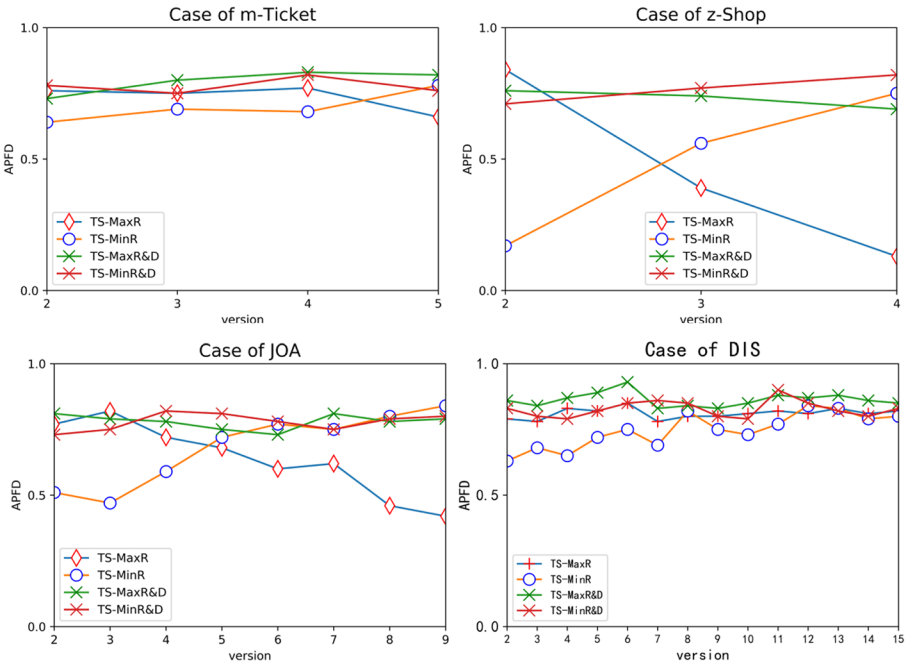

Fig. 7 The trends of APFD values changing with *TS-MaxR*, *TS-MinR*, *TS-MaxR&D*, and *TS-MinR&D*

experimental results. It needs to be validated by more cases in different areas, different scales, and different data distribution characteristics.

2. Comparison method selection. The methods used in this study to compare with TCP-SR come from related work, which also have validity risks that will be introduced into these experiments. At the same time, these methods do not represent all test case prioritization methods and need to be validated against with more methods.

6 Conclusions and future work

This paper presents a test case prioritization approach TCP-SR based on service ranking, describes the framework and each steps in detail, and verifies its validity through empirical studies. Instead of requiring artifacts such as codes, specifications, and models as inputs, TCP-SR generates service ranking vectors from the API gateway layer logs, and orders test cases based on the vector with appropriate strategies. The whole process can be fully automated and is applicable to the microservice systems regression testing in practice.

The future work will include the following two aspects:

1. TCP-SR emphasizes on service ranking rather than service dependency ranking, which makes it difficult to reflect the differences between combinations of the same set of services. We will extend the method to consider the service dependencies ranking,

establish the corresponding test case quantification model and strategies, and conduct experimental validation.

2. TCP-SR ranks service by mining log information, the essence of which is mining software architecture information from running data. And it requires all requests pass through the API gateway layer. To extend the feasibility of our approach, we will try to apply TCP-SR in different areas and different architecture patterns such as service mesh and collect more cases for empirical study.

References

Abele, S., & Göhner, P. (2014). Improving proceeding test case prioritization with learning software agents. In: *Proceedings of 6th International Conference on Agents and Artificial Intelligence*. *ICAART*, (*2*), 293-298.

Al-Hajjaji, M., Thüm, T., Lochau, M., Meinicke, J., & Saake, G. (2019). Effective product-line testing using similarity-based product prioritization. *Software and Systems Modeling, 18*(1), 499–521. https://doi.org/10.1007/s10270-016-0569-2

Azizi, M., & Do, H. (2018a). A collaborative filtering recommender system for test case prioritization in web applications. In *Proceedings of the 33rd annual ACM symposium on applied computing,* (pp. 1560-1567). https://doi.org/10.1145/3167132.3167299

Azizi, M., & Do, H. (2018b). Graphite: a greedy graph-based technique for regression test case prioritization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (pp. 245-251).

Canfora, G., & Di Penta, M. (2009). Service-oriented architectures testing: A survey Software Engineering. Springer, 78–105. https://doi.org/10.1007/978-3-540-95888-84

Catal, C., & Mishra, D. (2013). Test case prioritization: A systematic mapping study. *Software Quality Journal, 21*(3), 445–478. https://doi.org/10.1007/s11219-012-9181-z

Chen, L., Wang, Z., Xu, L., Lu, H., & Xu, B. (2010). Test case prioritization for web service regression testing. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering,* (pp. 173-178). IEEE.

de S. Campos Junior, H., Araújo, M. A. P., David, J. M. N., Braga, R., Campos, F., & Ströele, V. (2017). Test case prioritization: a systematic review and mapping of the literature. In *Proceedings of the 31st Brazilian Symposium on Software Engineering* (pp. 34-43).

Diefenbach, D., & Thalhammer, A. (2018). PageRank and generic entity summarization for RDF knowledge bases. In: Proceedings of European Semantic Web Conference. Springer.

Fan, C. Y., & Ma, S. P. (2017). Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 IEEE International Conference on AI & Mobile Services (AIMS)* (pp. 109-112). IEEE. https://doi.org/10.1109/AIMS.2017.23

Gao, C., Zheng, W., Deng, Y., & Lo, D., (2019). Emerging app issue identification from user feedback: Experience on wechat. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 279-288). IEEE.

Gökçe, N., & Eminli, M. (2014). Model-based test case prioritization using neural network classification. *Computer Science & Engineering: An International Journal (CSEIJ)., 4*(1), 15–25. https://doi.org/10.5121/cseij.2014.4102

Horváth, F., Gergely, T., Beszédes, Á., Tengeri, D., Balogh, G., & Gyimóthy, T. (2019). Code coverage differences of Java bytecode and source code instrumentation tools. *Software Quality Journal, 27*(1), 79–123. https://doi.org/10.1007/s11219-017-9389-z

Kaczanowski, T. (2013). Practical Unit Testing with JUnit and Mockito. In: https://site.mockito.org/

Khan, T. A., & Heckel, R. (2011). On model-based regression testing of web-services using dependency analysis of visual contracts. In *International Conference on Fundamental Approaches to Software Engineering,* (pp. 341-355).

Kim, M., Sumbaly, R., & Shah, S. (2013). Root cause detection in a service-oriented architecture. *In ACM SIGMETRICS Performance Evaluation Review, ACM, 41*, 93–104.

Konsaard, P., & Ramingwong, L. (2015). Total coverage based regression test case prioritization using genetic algorithm. *Proceedings of 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*.

Korel, B., Koutsogiannakis, G., & Tahat, L. H. (2008). Application of system models in regression test suite prioritization. *Proceedings of International Conference on Software Maintenance*, 247–256.

Korel, B., Tahat, L. H., & Harman, M. (2005). Test prioritization using system models. *Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 559–568.

Larrucea, X., Santamaria, I., Colomo-Palacios, R., & Ebert, C. (2018). Microservices. *IEEE Software, 35*(3), 96–100.

Lewis, J., & Fowler, M. (2014). Microservices: a definition of this new architectural term. In: http://martinfowler.com/articles/microservices.html

Li, B., Qiu, D., Leung, H., & Wang, D. (2012). Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph. *Journal of Systems and Software*, *85*(6), 1300-1324. https://doi.org/10.1016/j.jss.2012.01.036

Li, Z. J., Tan, H. F., Liu, H. H., Zhu, J. M., & N. M. (2008). Business-process-driven gray-box SOA testing. IBM. *System, 47*(3), 457–472. https://doi.org/10.1147/sj.473.0457

Liu, H., Li, Z., Zhu, J., & Tan, H. (2007). Business process regression testing. *Proceedings of the 5th International Conference on Service-Oriented Computing*, 157–168.

Mahdian, A., Andrews, A. A., & Pilskalns, O. J. (2009). Regression testing with UML software designs: A survey. *Journal of Software Maintenance and Evolution: Research and Practice, 21*(4), 253–286. https://doi.org/10.1002/smr.v21:4

Mece, E. K., Binjaku, K., & Paci, H. (2020). The application of machine learning in test case prioritization - a review. *European Journal of Electrical and Computer Engineering, 4*(1), 1–9. https://doi.org/10.24018/ejece.2020.4.1.128

Meszaros, G. (2007). *xUnit: Test Patterns Refactoring Test Code*. Boston, Addison-Wesley.

Mohd Shafie, M. L., & Wan Kadir, W. M. N. (2018). Model-based test case prioritization: a systematic literature review. *Journal of Theoretical and Applied Information Technology (JATIT & LLS), 96*(14), 4548–4573.

Mirshokraie, S., Mesbah, A., & PaŠabiraman, K. (2015). Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering, 41*(5), 429–444. https://doi.org/10.1109/TSE.2014.2371458

Naik, A., Maeda, H., Kanojia, V., & Fujita, S. (2018). Scalable Twitter user clustering approach boosted by Personalized PageRank. *International Journal of Data Science and Analytics*, *6*(4), 297-309. https://doi.org/10.1007/s41060-017-0089-3

Newman, S. (2015). Building microservices: designing fine-grained systems. O'Reilly Media.

Ouriques, J. F. S., Cartaxo, E. G., & Machado, P. D. L. (2018). Test case prioritization techniques for model based testing: A replicated study. *Software Quality Journal, 26*(4), 1451–1482.

Prado, J. A., Lima, S. R., & Vergilio. (2020). Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*. https://doi.org/10.1016/j.infsof.2020.106268

Qiu, D., Li, B., Ji, S., & Leung, H. (2014). Regression testing of web service: a systematic mapping study. *ACM Computing Surveys (CSUR)*, *47*(2), 1-46.

Richardson, M., & Domingos, P. (2002). The intelligent surfer: Probabilistic combination of link and content information. *PageRank Advances in Neural Information Processing Systems, 14*, 673–680.

Roberto, P. (2020). On the testing resource allocation problem: Research trends and perspectives. *Journal of Systems and Software*. https://doi.org/10.1016/j.jss.2019.110462

Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017). Reinforcement learning for automatic test case prioritization and selection in continuous integration. *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 12–22.

Wang, X., Zeng, H., Gao, H., Miao H., & Lin, W. (2019) Location-based test case prioritization for software embedded in mobile devices using the law of gravitation. *Mobile Information Systems*, 1–14. https://doi.org/10.1155/2019/9083956

Wu, G., & Wei, Y. (2007). A power-arnold algorithm for computing PageRank. *Numerical Linear Algebra with Applications, 14*(7), 521–546. https://doi.org/10.1002/nla.531

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability, 22*(2), 67–120. https://doi.org/10.1002/stv.430

Zhang, M., Li, X., Zhang, L., & Khurshid, S. (2017). Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis,* (pp. 261-272).

Zhao, X., Wang, Z., Fan, X., Wang, Z. (2015). A clustering-bayesian network based approach for test case prioritization. *Proceedings of IEEE 39th Annual Computer Software and Applications Conference, Taichung*.

Zhongsheng, Q. (2010). Test case generation and optimization for user session-based web application testing. *Journal of Computers*, *5*(11), 1655-1662. https://doi.org/10.4304/jcp.5.11.1655-1662

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

LIZHE CHEN receivedhis master degree in computer application technology from National DefenseUniversity of Science and Technology, Hefei, China, in 2010. He is currently aPh.D candidate in the oftware Engineering Institute(SEI) at Beihang University.His research interests include model driven engineering, data mining, machinelearning, model based safety analysis, and software testing.

JI WU is associateprofessor of software engineering at Beihang University. He received his PhDdegree from Beihang University in 2003 and MS degree from the Second ResearchInstitute of the China Aerospace Science and Industry Group in 1999. Hisresearch interests include embedded system and software modeling andverification, software requirement and architecture modeling and verification,safety and reliability assessment, and software testing.

HAIYAN YANG is lecture of software engineering at Beihang University. She received her master degree in Computer Software and Theory at Beihang University in 2000. Her research interests include software modeling and verification, software testing, software measurement.

KUI ZHANG receivedhis master degree in information management and information system from BeijingInstitute of Technology, Beijing, China, in 2010. He is currently a Ph.Dcandidate in the oftware Engineering Institute(SEI) at Beihang University. Hisresearch interests include model driven engineering, model based real timeanalysis, airworthiness certification, model base safety analysis, and generalmodel based software engineering.