CRA: Identifying Key Classes Using Markov-Chain-Based Ranking Aggregation

Xin Du 1, Weifeng Pan 1,*, Bo Jiang 1, Luyun Ding 1, Yun Pan 1, Chengxiang Yuan 1 and Yiming Xiang 2,*

1 School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China
2 School of Management and E-Business, Zhejiang Gongshang University, Hangzhou 310018, China
* Correspondence: wfpan@zjgsu.edu.cn (W.P.); yx_mail@126.com (Y.X.)

Abstract: Researchers have proposed many approaches to identify key classes in software from the perspective of complex networks, such as CONN-TOTAL-W, PageRank, and ElementRank, which can effectively help developers understand software. However, these approaches tend to rely on a single metric when measuring the importance of classes. They do not consider the aggregation of multiple metrics to select the winner classes that rank high in majority metrics. In this work, we propose a key class identification approach using Markov-Chain-based ranking aggregation, namely CRA. First, CRA constructs a weighted directed class coupling network (WDCCNet) to describe the software and further applies existing approaches on WDCCNet to calculate class importance. Second, CRA filters out some metrics according to specific rules and uses the Markov chain to aggregate the remaining metrics. When the state probability distribution reaches a fixed point and does not change anymore, the classes in the software are sorted in a descending order according to the probability distribution, and the top-15% classes are treated as key classes. To evaluate the CRA approach, we compare it with 10 baseline approaches available on 6 pieces of software. Empirical results show that our approach is superior to the baselines according to the average ranking of the Friedman Test.

Keywords: key class identification; complex network; Markov chain; ranking aggregation

1. Introduction

The development of software cannot be separated from the evolution of software—“the dynamic behavior of software systems as they are maintained and enhanced over their lifetimes” [1,2]. In practical scenarios, software needs to be constantly updated to meet new user requirements and working environments [3,4], such as code modification, functionality increase, and bug fixes. After iterations and updates, a software’s scale will become larger, and its function will be more complex, which makes the software more difficult to understand. Especially in the context of real development, when a developer unfamiliar with the development process of the software is confronted with thousands of lines of code, where does he start to understand such a complex software? In other words, what are the effective technologies to help developers deal with the great challenges they face?

We focus our attention on how to help developers understand the software. Object-Oriented (OO) languages have become the mainstream language in software development; in OO paradigms, software systems are composed of various elements, such as methods, classes, and packages. Complex networks provide a useful perspective to represent software by abstracting these components of the software as nodes and the coupling relationships between these elements as edges in the network. These essential components are closely coordinated to achieve the important function of the software. Therefore, developers only need to find the important nodes on the network and then regard them as the starting point for understanding the software. In recent works, researchers utilized complex networks to represent software at the class granularity and named such networks...
as Software Network (SN) [5–9]. SN, another type of complex network, also possesses two properties that a complex network has [9], i.e., \textit{scale-free} (the degree distribution of nodes in the network obeys a power-law distribution) [10] and \textit{small-world} (a node requires only a short path to reach another node) [11].

In the field of complex networks, researchers have proposed many methods for measuring important nodes in networks, such as k-shell [12], degree centrality [13], and PageRank [14]. Most of these methods are applied to unweighted and undirected networks. However, a software network is a weighted directed network whose components interact with different closeness. These methods cannot be applied directly to SN. Therefore, software engineering researchers improved these methods and proposed new metrics, such as \textit{Core}_{wk} [15], \textit{CONN-TOTAL-W} [6], \textit{PageRank}_{BR} [16], \textit{ElementRank} [17], and \textit{ClassRank} [18] etc., to identify key class nodes in the SN. Although these metrics obtained satisfying performance in key class identification, there is still a small part of key classes that are not successfully identified. Meanwhile, using a single metric cannot measure the importance of a class comprehensively.

Motivation: Social voting systems often determine preferred candidates in elections based on preferential ballots and pairwise comparison counts. During the comparison, the object wins a majority of the counts in a head-to-head battle with other competitors. That is, if there is a candidate who ranks high in most of the voting rankings, then the object is called “Condorcet winner”. This phenomenon is not only presented in social voting systems but is also applied to the field of key class identification. We can consider the above approaches as voters, and classes are treated as candidates. Aggregation of multiple metrics generated by these approaches is achieved by pairwise comparison of the performance (ranking) of each class in them. Specifically, when one class ranks higher than the others by more than half metrics, it is regarded as a winner candidate and should be ranked higher in the final aggregated sequence. Therefore, in this work, we attempt to aggregate multiple metrics in a “the majority rule” way to obtain a new metric and check the ranking of the entity in this sequence to identify the key class.

In this paper, we are inspired by social choice theory and propose the CRA approach. It can generate a ranked list of classes for each software, in which the top-ranked classes are considered as key classes. Such classes can be used as a starting point for developers to understand the software. Specifically, first, we extract the software structure at the level of class granularity and construct the software network. Second, we calculate the importance of each node in the software network using 10 types of methods that were proposed and widely used. Then, we filter out several metrics according to specific rules and use the Markov chain to aggregate the remaining. Finally, all classes will be ranked according to the steady-state distribution of the Markov chain, and the top-ranked classes will be considered as key classes for the software comprehension process. In addition, CRA is empirically evaluated in six open source software. Through a series of experiments, we compare the performance of the CRA over the 10 methods. Empirical results show that our approach is superior to these baseline approaches.

The contribution of this paper can be summarized as follows:
- We propose a novel approach—CRA—to aggregate class metrics and obtain a ranking sequence of classes. Developers can treat the top-ranked classes as the starting point for software understanding;
- We evaluate CRA on six existing open source software and compare it with 10 baseline approaches. The results show that our approach achieves superior performance.

The rest of this paper is organized as follows: Related work is reviewed in Section 2. Our approach is described in detail in Section 3. Section 4 is theoretical and empirical evaluations, where Section 4.5 shows the results of our experiments. Section 5 is a summary of this work and discusses future work.
2. Related Work

In machine learning, approaches are classified into supervised and unsupervised learning based on whether labels need to be known beforehand. In the field of key class identification, researchers have also conducted extensive research using these two techniques.

Supervised techniques: The supervised technology uses a set of labeled data to learn a mapping from input to output and then apply this mapping relationship to unknown datasets to achieve the purpose of key class classification. Osman et al. used supervised learning to locate key classes by compressing class diagrams in open source projects [19]. Thung et al. extended the work by Osman et al. [20]. They combined design and network metrics as features and fed them into the classifier for training. However, these approaches suffered from some problems such as lack of data and class imbalance. To address this problem, researchers began experimenting with unsupervised learning to identify key classes in software.

Unsupervised techniques: Unlike supervised learning, unsupervised technology to identify key classes reveals the inherent characteristics and laws of data through learning these unlabeled samples. Zhou et al. [21] proposed weighted undirected class dependency networks, and then they used the h-index and its invariant metric to measure the importance of the nodes. Sora et al. [6] proposed CONN-TOTAL and CONN-TOTAL-W, where CONN-TOTAL is the total number of nodes connected to the specific node, and CONN-TOTAL-W is the sum of the weights on all link (in- and out-link) connected to the one node. Meyer constructed unweighted undirected class networks and used the K-Core decomposition algorithm to identify key classes in software [5]. Pan et al. extended Corek and proposed the Corewk approach to identify key classes [15]. The degree calculation is the biggest difference between Corewk and Corek. Specifically, the Corewk considers both the link direction and weight when calculating the degree, which Corek does not consider. Steidl et al. employed directed class dependency networks to represent a piece of software and applied PageRank [6,22–24], Betweenness, and HITS algorithms to measure the importance of nodes in the network [24]. PageRankBR was proposed by Sora et al. The difference between PageRankBR and PageRank is that the former takes back recommendation into account [16]. ElementRank [17] was proposed by Pan et al. They constructed multi-layer software networks at the class and package granularity of level, respectively. Moreover, the traditional PageRank algorithm only fits unweighted networks. Based on this, Pan et al. applied a weighted PageRank algorithm. Recently, Pan et al. proposed a PageRank-like algorithm called ClassRank. This work believed that classes with larger out- and in-degrees are tightly coupled with others. Thus, these classes are more accessible and important [18].

However, a single metric often does not provide a more comprehensive measure of the importance of a key class. In this work, we attempt to aggregate multiple existing metrics in a “minority rule” way. The purpose of doing this is to obtain a class importance metric that considers the advantages of multiple metrics. Our approach belongs to unsupervised learning in a strict sense.

3. The Proposed Method CRA

Figure 1 gives an overview of our proposed CRA method. It consists of four steps:

- Step 1: Building software network; in the first step, by compiling the source code of the Java software system, we extract its software structure and construct the software network;
- Step 2: Calculating Class Importance Metrics; in the second step, we apply the mainstream 10 key class identification methods to the constructed network and obtain 10 metrics respectively to measure the importance of classes;
- Step 3: Aggregation of class importance metrics; in the third step, we filter out some of the metrics by specific rules and aggregate the rest metrics. After the n-step transition, the system gradually converges to a fixed point, and, eventually, we can obtain the stationary distribution of the Markov chain;
• Step 4: we rely on this probability distribution to rank the classes, and top-ranked classes will be considered key as classes.

Input
Source Code (.java)

Steady-state distribution of the system when it reaches a constant value

Output
Filtering of indicators according to specific rules
Aggregation of the remaining metric using Markov chains

Figure 1. The overall process of the proposed approach.

In this work, we parse the source code of the java projects. The reasons are as follows: (1) Most applications are developed in Java. (2) Compared with other programming languages, we can more easily find software projects written in Java for all scales. Note that, our approach is shown with examples in Java, but the same approach can be extrapolated to other languages.

3.1. Building Software Network

The construction process of the software network is shown in Figure 2. Firstly, we developed SNAP (Software Network Analysis Platform [17,18]) to extract the entities and coupling types at a specific granularity level from the software system’s static source code; then, different weights are assigned to the different coupling types; finally, these components and the couplings between them form the software network.

The nature of the traditional network is a graph containing two parts, \((V, L)\), where \(V\) denotes the network nodes and \(L\) denotes the links between the nodes. The mapping from the software network to the traditional network is equivalent to abstracting entities and interactions of entities in the software into the nodes and edges in the network.

Structural Information Extraction: Entities in software contain multiple granularities level, including packages, classes, etc. Considering that our work is an aggregation strategy of class importance rankings generated by existing key class identification approaches, it is natural to think of extracting software structures at the class granularity level.

Coupling Types Extraction: Traditional networks only reflect the existence of links between nodes. However, there are various coupling relationships among the entities in software. In this work, we recognize nine types existing in software entities and assign directions links according to them [18].

Definition 1. (Coupling Types Between Classes). For any two classes node \(u\) and \(v(u \neq v)\), if there is a relationship as defined in Table 1, then we will recognize and extract it as \(u \rightarrow v\):
Table 1. Descriptions of the coupling relationships.

| Relationship                | Description                                      |
|-----------------------------|--------------------------------------------------|
| INheritance RELation (INR)  | $u$ inherits $v$ by keywords “extends”           |
| IMplementation Relation (IMR)| $u$ implements interface $v$ by keywords “implements” |
| INStantiates (INS)          | $u$ instantiates an object of $v$                |
| Local Variable Relation (LVR)| One method of $u$ contains a local variable of $v$ |
| Global Variable Relation (GVR)| $u$ has an attribute with type $v$               |
| ACCess (ACC)                | One of the $u$’s methods accesses a property of $v$ |
| Method Call Relation (MCR)  | One of the $u$’s methods calls other method on an object of $v$ |
| Return Type Relation (RTR)  | One of the $u$’s methods has a return type of $v$ |
| PARameter type (PAR)        | One of the $u$’s methods has a parameter type of $v$ |

Coupling Strength Calculation: Different coupling relationships in software have different coupling strengths. In this work, we adopt the objective weighting mechanism based on the distribution of coupling types, which was proposed by Abreu et al. [25]. The reasons can be divided into two aspects: (1) Internal reason: since our approach is based on existing approaches, to ensure fairness and consistency, the weights of entities in the CRA should be consistent with the assignment in the network constructed by existing approaches. (2) External reason: this weighting mechanism can objectively measure the closeness between classes and is widely used in several works.

In the previous step, we recognize nine types of couplings, let $R = \{\text{INR, IMR, INS, LVR, GVR, ACC, MCR, RTR, PAR}\}$. We use two $9 \times 1$ column vectors $w = [w_{\text{INR}} \cdots w_{\text{PAR}}]^{\top}$ and $o = [o_{\text{INR}} \cdots o_{\text{PAR}}]^{\top}$ to represent the weights and occurrence frequency of various coupling types on the link $u \rightarrow v$, respectively. The definition of weight on the link that connects class nodes $u$ and $v$, i.e., $W_{u \rightarrow v}$, can be considered as the dot product of two vectors $w_R$ and $o_R$.

$$W_{u \rightarrow v} = w \cdot o = [w_{\text{INR}} \cdots w_{\text{PAR}}]^{\top}[o_{\text{INR}} \cdots o_{\text{PAR}}] = w_{\text{INR}}o_{\text{INR}} + \cdots + w_{\text{PAR}}o_{\text{PAR}}$$  \hspace{1cm} (1)

$r$ belongs to $R$, which refers to any of the nine coupling types. $w_r$ denotes the weight assigned to a particular coupling type, and it is defined as follows:

$$w_r = \begin{cases} 
10 & N_{\text{intra}}^r \neq 0 \land N_{\text{inter}}^r = 0 \\
1 & N_{\text{intra}}^r = 0 \land N_{\text{inter}}^r = 0 \\
\text{Round}(0.5 + 10 \times \frac{N_{\text{intra}}^r}{N_{\text{intra}}^r + N_{\text{inter}}^r}) & \text{otherwise}
\end{cases}$$ \hspace{1cm} (2)

where $N_{\text{intra}}^r$ and $N_{\text{inter}}^r$ indicate the number of intra- and inter-module couplings, respectively [15].

Software Network Definition: After defining the coupling relationship (link direction in the network) and the coupling strength (link weight in the network), we construct the WDCCNet (Weighted Directed Class Coupling Network) to represent the software at the class level of granularity. It is defined as follows:

**Definition 2.** WDCCNet contains two parts, i.e., $(V, L)$. We regard class in software as nodes. Note that the term class refers to both classes and interfaces in software. If a relation exists between two classes in the software, there will be a link connect two nodes in the WDCCNet. For instance, $u$ and $v$ are two class nodes in WDCCNet. When they interact in the actual software with the above nine coupling types, there will be a link $L_{u \rightarrow v} = <u, v>$ (where $L_{u \rightarrow v} \in L$).
3.2. Calculating Class Importance Metrics

We count 10 key class identification methods that are the current mainstream approaches, i.e., \( h \)-index [21], \( a \)-index [21], CONN-TOTAL [6], CONN-TOTAL-W [6], Core\( k \) [26], Core\( w_k \) [15], PageRank [6,22–24], PageRank\( _{BR} \) [16], ElementRank, and ClassRank [18]. We make a brief introduction to them in Section 2.

We implement these approaches and apply them to the WDCCNet that we constructed, and we can then obtain 10 metrics for measuring the importance of classes. The larger values indicate that the class is more important, which applies to all approaches.

3.3. Aggregation of Class Important Metrics by Markov chain

3.3.1. Brief Review of Markov chain

The model of Markov chain refers to the stochastic process of transition from one state to another in the state space. Specifically, similar to [27], the Markov chain model is defined as follows:

**Definition 3.** Markov\( \left( S, \{ P, \pi^{(n)} \}_{n=1}^N \right) \) is composed of a state space \( S \), a set of \( N \) Markov chains \( \{ P, \pi^{(n)} \} \) defined over \( S \). Each \( \{ P, \pi^{(n)} \} \) is a Markov chain that is time-homogeneous, which means that the current state is only related to the previous one state and not to the past state. \( P \) is a transition matrix, also called stochastic matrix, whose element \( P_{ij}^{(n)} \) is the value of \( i \)-th row and \( j \)-th column of \( P \), which represents the probability that the chain that started in state \( i \) hits \( j \) in \( n \)-steps \((i, j \in S)\). \( P \) satisfies the following properties: (1) \( P_{ij}^{(n)} \geq 0 \) for all \( i,j \); (2) for each row \( i \), \( \sum_j P_{ij} = 1 \). \( \pi^{(n)} \) denotes the state probability vector after \( n \)-steps, where \( \pi = \pi_0, \pi_1, ..., \pi_{|S|-1} \). \( \pi^{(0)} \) is the initial state distribution of the Markov chain.

3.3.2. Refining Metrics

We use a group of agents to denote each component of the Markov chain. Let \( T \) be the number of classes (state). \( S = [s_1, ..., s_T] \in \mathbb{R}^{1 \times T} \) is a state space, where \( T = |S| \). Each class node in WDCCNet corresponds to a particular state \( s_t (s_t \in S) \). Let \( \tau = (\tau_1, \tau_2, ..., \tau_H) \), where \( \tau_h \) and \( H \) refer to a ranking returned by one specific class importance metric and the total number of metrics, respectively. Thus, the ranking position of each class is denoted as \( \tau_h (s_t) \) \((s_t \in S)\). Meanwhile, when \( s_u \) ranks before \( s_v \), we write this process as \( \tau_h^{(s_u)} \prec \tau_h^{(s_v)} \). However, one thing that needs to be considered is whether the ranking is a strong sorting, i.e., if class \( u \) and class \( v \) have the same value calculated by a specific method, we cannot compare their importance. Thus, a rule was proposed by us to filter out unsuitable metrics according to their repetition rates. The specific definition of the Repetition Rate (RR) is as follows:

**Definition 4.** Repetition Rate\( (RR) = 1 - \frac{T-r}{T} \), where \( T \) equals the total number of classes (state), and \( r \) denotes the count of recurring values. Suppose \( RR \geq 0.05 \), which means the current metric has a high repetition rate. We cannot rank the classes based on the metric values very well.

As shown in Table 2, we calculate the RR of the 10 approaches on 6 software systems. According to the threshold of 0.05, we can filter out a portion of unsuitable measures. Eventually, five metrics are left for later aggregation operations. Therefore, \( H = 5 \). Note that the repetition rate is calculated based on the top-15% classes of the ranking list returned by each metric in a descending order. The reasons are two-fold: (1) There are many outlier nodes in WDCCNet, whose value is small, and they are mostly distributed in lower positions. Therefore, counting the repetition rate of these class nodes is meaningless. (2) Most software engineering researchers identified key classes in software only by checking the top-ranked classes [15,17,28,29].
Table 2. The results of the repetition rate. The cells highlighted in gray background denote the values lower than 0.05 and the corresponding metrics that are left.

| Systems   | h-Index | a-Index | CONN-TOTAL | CONN-TOTAL-W | Core<sub>h</sub> | Core<sub>wh</sub> | PageRank | PageRank<sub>GR</sub> | ElementRank | ClassRank |
|-----------|---------|---------|------------|--------------|----------------|----------------|-----------|---------------------|-------------|----------|
| Ant       | 0.9823  | 0.5778  | 0.8605     | 0.2388       | 1              | 0              | 0.0296    | 0.0296              | 0.0148      | 0.0      |
| Argo UML  | 0.9796  | 0.7953  | 0.8444     | 0.2835       | 1              | 0.1811         | 0.0315    | 0.0000              | 0.0          | 0.0      |
| jEdit     | 0.9733  | 0.5305  | 0.8929     | 0.2439       | 1              | 0              | 0.0000    | 0.0000              | 0.0          | 0.0      |
| JHotDraw  | 0.9756  | 0.3537  | 0.7750     | 0.0988       | 1              | 0.0244         | 0.0000    | 0.0000              | 0.0          | 0.0      |
| JMeter    | 0.9556  | 0.3077  | 0.7000     | 0.1538       | 1              | 0              | 0.0000    | 0.0000              | 0.0          | 0.0      |
| wro4j     | 0.9747  | 0.4941  | 0.8000     | 0.1860       | 1              | 0              | 0.0000    | 0.0000              | 0.0          | 0.0      |
| AVG       | 0.9735  | 0.5099  | 0.8121     | 0.2008       | 1              | 0.0343         | 0.0102    | 0.0102              | 0.0025      | 0.0000   |

3.3.3. Aggregating Metrics

In this part, we elaborate on how to aggregate metrics step-by-step. After the above operation, five metrics are left. We aggregate metrics mainly by constructing the transition matrix \( P' \), based on the following idea [30]: Assuming the current state is class \( s_u \). First, we need to evenly select another state (class) \( s_v \) from the remaining classes. Next, we need to compare the position about two classes ranked by metrics. If \( \tau_h^{(s_v)} \prec \tau_h^{(s_u)} \) in majority of ranking returned by metrics, then the current state \( s_u \) transfer to \( s_v \). Otherwise stay in state \( s_u \). Based on the above ideas, the element of the original transition matrix \( P \) can be constructed as follows:

\[
P'_{ij} = \begin{cases} 
0, & \text{if } \sum_h \left| \tau_h^{(i)} \prec \tau_h^{(j)} \right| > \frac{1}{2} \times H, \text{ s.t. } i \neq j \\
\frac{1}{T}, & \text{else}
\end{cases}
\]

Note that, there exists a constraint on the Equation (3) that \( i \neq j \). After a normalized operation, the diagonal part \((i = j)\) of the matrix \( P' \) can be supplemented according to one of the properties satisfied by the transfer matrix, i.e., for each row \( i \), \( \sum_j P_{ij} = 1 \). However, the constructed matrix \( P' \) has a small problem: there are two states that point to each other but not to the other state: once a state points to one of them, a loop transfer occurs. Thus, to avoid loop transfers between states, i.e., it is possible to go from one state to every state. We introduce the ergodic Markov chain:

\[
P = \alpha P' + (1 - \alpha)K,
\]

where \( P \) is the transition matrix for an ergodic Markov chain, \( \alpha (0 < \alpha < 1) \) is a coefficient that is typically set to 0.85 [17,22,23,30]. \( K \) is a \( T \times T \) matrix each of whose entries is \( \frac{1}{T} \).

3.3.4. An Example to Aggregate Metrics via Markov Chain

In this section, we describe the calculation process of our CRA approach in detail by giving an example. Suppose that the existing three metrics return the importance value of classes A, B, and C, as shown in Table 3. Firstly, we translate their value into rankings as shown in Table 4.

Table 3. An example of a simulation class importance calculation.

| Metric-1 | Metric-2 | Metric-3 |
|----------|----------|----------|
| A        | 0.3      | 0.2      | 0.3      |
| B        | 0.2      | 0.1      | 0.1      |
| C        | 0.1      | 0.3      | 0.2      |
Table 4. An example of a simulated class ranking.

|     | Metric-1 | Metric-2 | Metric-3 |
|-----|----------|----------|----------|
| A   | 1        | 2        | 1        |
| B   | 2        | 3        | 3        |
| C   | 3        | 1        | 2        |

Next, we construct the transition matrix according to Equations (3) and (4). For class A, two of the three metrics rank it higher than C (more than half of all metrics), and all metrics rank it higher than B. Therefore, the definition of the transition matrix \( P \) is as follows:

\[
P = 0.85 \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 1 \end{bmatrix} + (1 - 0.85) \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} = \begin{bmatrix} \frac{18}{20} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{13}{20} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{13}{20} \end{bmatrix}
\]

We set initial state \( \pi(0) \) so it equals \( \left[ \frac{1}{3} \frac{1}{3} \frac{1}{3} \right] \). Note that the initial state is not unique since the final stationary distribution is regardless of the initial distribution. Then, the next state probability distribution \( \pi(1) \) is the product of the \( \pi(0) \) and transition matrix \( P \). We describe this process as

\[
\pi(1) = \pi(0)P = \begin{bmatrix} 47 \\ 90 \\ 90 \end{bmatrix} \neq \pi(0)
\]

In such a manner, we can calculate \( \pi(n) \) until it reaches a fixed point and does not change anymore, i.e., \( \pi(n) = \pi(n-1) \). In this example, after 27 iterations of computation, the final state probability distribution \( \pi(27) \) converges to \( \left[ 0.7692 \ 0.0698 \ 0.161 \right] \), i.e., \( \pi(27) = \pi(28) \). Such a distribution that satisfies \( \pi(n) = \pi(n-1) \) is called a stationary distribution. We rank three classes according to the values of the sequence formed by the entities in the stationary distribution. Using the above process, we achieve the aggregation of multiple metrics to rank the classes.

### 3.4. Class-Level Ranking

We rank classes in a descending order according to the value of stationary distribution. Top-ranked classes are more important than lower-ranked classes. As with these efforts \([15,17,28,29]\), we use 15% as the threshold for identifying key classes. That is, the top-15% of classes were regarded as key classes, and these classes can be used as a starting point for developers to understand the software.

### 4. Experimental Validation

In this section, we describe the experimental setup that we follow to evaluate CRA. The experimental environment is a desktop computer equipped with AMD Ryzen 5 5600G with Radeon Graphics 3.90 GHz CPU and 16GB RAM, running Win10 (64-bit).

#### 4.1. Research Questions

In the experiments, we aim to address several research questions:

- **RQ1**: *Can our method improve how to distinguish the difference between classes?* We rank classes according to the probability of each state when they converge to a steady point. Thus, if two state have the same probability in stationary distribution, they cannot be ranked in theory. In this experiment, we need to judge whether we can obtain a strong ordering according stationary distribution;

- **RQ2**: *Can our method effectively identify key classes distributed in software systems?* Currently, many methods have been proposed and applied to nine key classes. In this experiment, we focus on our method’s performance compared to other baseline methods when the cut-off line equals 15%.
4.2. Subject Systems

In this work, we chose six pieces of software systems as our subject software. The reasons were twofold: (1) These systems were applied to other advanced research work, which proves that these systems are representative. (2) It can be more intuitive to compare our method with these mainstream methods on the same software systems.

We list the detailed information about these six systems in Table 5. This table consists of three columns. We provide the complete name in the left column. The second and third columns offer the version and the number of classes in each software system. In order to facilitate other researchers to download the software and reproduce our experiments, we list the specific URL of the software in the right column.

Table 5. Basic information of the subject software systems.

| System   | Version | #C | URL                                      |
|----------|---------|----|------------------------------------------|
| Apache Ant | 1.6.1   | 900 | http://ant.apache.org                   |
| Argo UML  | 0.9.5   | 846 | https://argouml-tigris-org.github.io/    |
| jEdit     | 5.1.0   | 1082(9) | http://www.jedit.org                  |
| jHotDraw  | 6.0b.1  | 544 | http://www.jhotdraw.org                 |
| jMeter    | 2.0.1   | 258 | http://jmeter.apache.org               |
| wro4j     | 1.6.3   | 567(9) | https://code.google.com/p/wro4j (accessed on 9 August 2022) |

4.3. Baseline Approaches

There are 10 mainstream metrics described in Section 2. These approaches achieved excellent performance when applied to the field of key class identification. Thus, in this work, these metrics also play the role of baseline approaches compared with our proposed method. Since we briefly summarized these approaches above, we will not repeat them anymore here.

4.4. Evaluation Metrics

Classes can be divided into two classes, i.e., key and non-key classes. Key classes can be treated as positive samples while non-key classes are negative samples, indicating that identifying key classes is a binary classification problem. We adopt the evaluation metrics used by Refs. [5,6,16,18,21], which measure precision, recall, and F1 score over identification of key class. The recall is often used to correctly evaluate how many key classes are identified by our approach. Precision is used to evaluate the number of key classes in our retrieved classes. Note that most of the work is more concerned with the recall to evaluate the performance of the approach. For example, even if an approach identifies all key classes that only rely on the top-15% of classes, the base number of retrieved classes is still too large. Such an approach has a full recall but low precision, equivalent to automatically hurting the precision. Thus, we introduce the F1 score, a harmonic function about Recall and Precision, to use both together.

4.5. Experiment Results and Analysis

In this subsection, we focus on answering the RQs proposed in Section 4.1 by obtaining the results from a series of experiences in detail.

4.5.1. RQ1: Can Our Method Improve How to Distinguish the Difference between Classes?

We perform this experiment to evaluate whether the condition exists where we cannot rank classes due to their importance values being the same. Detailed steps are specifically expressed as follows. First, we apply 11 approaches to measure the classes’ importance value in 6 software networks. Then, we rank classes according to their importance value returned by 11 approaches and calculate the repetition rate of top-15% classes in this experiment; the threshold was still set at 0.05. The approach has a low repetition rate
when RR is equal to or less than 0.05, which indicates that the current approach can easily distinguish the class nodes in software networks.

The experimental result is shown in Figure 3. Every subfigure corresponds to a system. Its abscissa and ordinate represent 11 approaches and repetition rates, respectively. In order to display the comparison result more intuitively, we divide the threshold using a red dotted line. A value below the red line indicates that the approach performs well.

**Conclusion 1: Answer to RQ1**

Experimental result on six subject systems shows that our approach returns a low repetition rate of class importance value. That is to say, there are no classes with the same value in the top-15%.

![Figure 3. The comparison of repetition rate with baseline approaches for the top-15% classes.](image)

4.5.2. RQ2: Can Our Method Effectively Identify Key Classes Distributed in Software Systems?

We aim to evaluate our approach’s retrieval ability of key class by performing this experiment. Noticeably, it exists in some approaches that classes have the same importance value and cannot be sorted. Specifically, we use 15% as the cut-off line and check the top-15% of classes whether they are key classes. However, there are still classes with the same value of around 15%. Thus, we determine the specific number of retrieval classes according to the mean ranking of these indistinguishable classes. For example, we denote ith as the class of its ranking returned by one specific approach and assume them with the same importance value, whose ranking is obtained from 2th to 6th. While these five classes can be located in any of the five rankings due to the same value, we decide whether to append all of them by comparing the number of retrieval classes with their average location 4th (i.e., \( \frac{2+4+6}{3} \)th).

Table 4 visualizes the results of this experiment, which reflects the performance of the 11 methods through 3 evaluation metrics, namely Recall, Precision, and F1. The bold black values in the table indicate the approach that performs the best when we apply it to a particular software system.

However, it can be seen from Table 6 that no approach has the best performance across all software, even the most state-of-the-art approach. Thus, an objective technique
Axioms 2022, 11, 491

is adopted to select one approach with relatively better performance by conducting a comprehensive evaluation of them. Friedman Test [31], a non-parametric statistical test, can offer an important basis for comparison between algorithms. At the same time, such a technique is also widely used in similar scenarios to rank approaches based on their performance on multiple datasets. Generally speaking, the smaller the value returned by the Friedman test is, the better the approach performs. Every column in Table 7 denotes the average ranking result for one specific evaluation metric. The bold black font indicates that the corresponding approach performs best.

Table 6. Recall, precision, and F1 comparison of CRA against other baseline approaches (cut-off = 15%).

Table 7. Average ranking of the CRA and other baseline approaches.

Conclusion 2: Answer to RQ2

Our experimental result on six subject systems denotes that CRA is superior to other baseline approaches (i.e., ElementRank, ClassRank, etc.) according to the average ranking returned by the Friedman test, which indicates that our approach has excellent key class retrieval capabilities.
4.6. Threats of Validity

Several factors may influence the validity of our experimental conclusion. We divide these factors into two threats, i.e., the internal and external threats.

4.6.1. Threats to Internal Validity

One internal threat is the accuracy of the WDCCNet. It was extracted by the SNAP tool we developed. Therefore, the final result will be influenced by the accuracy of the network. However, this threat was minimized due to the SNAP tool having been sufficiently tested in our published papers [7,15,17,18]. Another is the metrics selection. Although we filter some metrics with a high reputation rate, there are still tiny nodes in the remaining metrics that have the same importance and cannot be ranked. Since this part of the node’s position is relatively backward, we only focus on the ranking of the top class. Thus, it does not pose a threat to our conclusion.

4.6.2. Threats to External Validity

The second potentially limiting factor is the selection of programming language. In this work, we aim to analyze Java. Extending it to other programming languages may differ a little from our conclusions. We will actively explore key classes in other OO languages, such as C++.

5. Conclusions and Future Work

In this work, we propose a key class identification approach based on the existing metrics and Markov chain. Our approach addresses the problem that a single metric cannot reflect the importance of the class from multiple perspectives. Specifically, we construct weighted directed class coupling software networks and apply some mainstream key class identification approaches to them. Then, we filter out some metrics according to specific rules and use the Markov chain to aggregate the remaining metrics. The stationary distribution of the Markov chain reaching a stable state can be described as the ranking of class importance. Top-15% classes were regarded as a key class to help developers understand unfamiliar software. The empirical results of experiments conducted on 6 Java subject systems show that our approach is superior to the other 10 baseline approaches according to the average ranking of the Friedman test. In future work, we will propose more efficient approaches to the nine key classes in software to help developers understand a piece of the system and extend our approach to other OO languages.

Author Contributions: Conceptualization, W.P. and X.D.; methodology, W.P. and X.D.; software, W.P., L.D. and X.D.; supervision, B.J., Y.P., C.Y., and Y.X.; writing-original draft, X.D. and L.D.; writing-review and editing, W.P., X.D., and L.D.. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (Grant Nos. 62272412, 62032010, and 61832014), the Natural Science Foundation of Zhejiang Province (Grant Nos. LY22F020007 and LY21F020002), and the Key R&D Program of Zhejiang Province (Grant Nos. 2021C01162 and 2019C01004).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data used to support the findings of this study are available from the corresponding author upon request.

Acknowledgments: The authors gratefully acknowledge all the reviewers for their positive and valuable comments and suggestions regarding our manuscript.

Conflicts of Interest: The authors declare that there are no conflicts of interest regarding the publication of this paper.
28. Zaidman, A.; Demeyer, S. Automatic identification of key classes in a software system using webmining techniques. *J. Softw. Maint. Res. Pract.* **2008**, *20*, 387–417. https://doi.org/10.1002/smr.370.

29. Jiang, S.J.; Ju, X.L.; Wang, X.Y.; Li, H.Y.; Liu, Y.Q. Measuring the importance of classes using UIO sequence. *Acta Electronica Sin.* **2015**, *43*, 2062.

30. Dwork, C.; Kumar, R.; Naor, M.; Sivakumar, D. Rank aggregation methods for the Web. In Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, 1–5 May 2001; Shen, V.Y., Saito, N., Lyu, M.R., Zurko, M.E., Eds.; ACM: New York, NY, USA, 2001; pp. 613–622. https://doi.org/10.1145/371920.372165.

31. García, S.; Fernández, A.; Luengo, J.; Herrera, F. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Inf. Sci.* **2010**, *180*, 2044–2064. https://doi.org/10.1016/j.ins.2009.12.010.