Component Rank: Relative Significance Rank for Software Component Search

Katsuro Inoue [†], Reishi Yokomori [†], Hikaru Fujiwara[†],
Tetsuo Yamamoto [††], Makoto Matsushita [†] and Shinji Kusumoto [†]
[†] Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
[††] Japan Science and Technology Corporation, 4-1-8, Honmachi,
Kawaguchi, Saitama 332-8531, Japan
{inoue, yokomori, t-yamamt, matusita, kusumoto}@ist.osaka-u.ac.jp

Abstract

Collections of already developed programs are important resources for efficient development of reliable software systems. In this paper, we propose a novel method of ranking software components, called Component Rank*, based on analyzing actual use relations among the components and propagating the significance through the use relations. We have developed a component-rank computation system, and applied it to various Java programs. The result is promising such that non-specific and generic components are ranked high. Using the Component Rank system as a core part, we are currently developing* Software Product Archiving, analyzing, and Retrieving System *named* SPARS.

1. Introduction

Computer systems are becoming core infrastractures of effective and efficient activities of everyday life. Programs involved in computer systems are getting larger and more complex, and a great number of software engineers are engaged in various kinds of software development projects, producing many kinds of programs day by day.

It would be imagined that similar programs have been developed independently in different locations of the world or in different times in the history, without sharing knowledge of other programs. It is considered that well-organized collection of programs or program components will improve productivity of the development and quality of the developed software products.

There are many kinds of software repositories available on the Internet such as Jumbo![13] or ZDNet[25]. These are mostly for software users who want to obtain already developed tools or systems, but not for software developers.

There is also a large software repository called SourceForge[21] where various software development projects are carried out with an open-source development model[20]. In this site, a search feature is provided to find a proper development project, but there is little support to explore program components, libraries, portions of codes, or abstracted algorithms, which are important resources for new development of programs. Reading, understanding, and duplicating well-written software legacies are indispensable activities for contemporary software development. We may want to keep all software legacies in a single repository; however, structuring and keeping consistency of the huge repository would be an impractically complicate and a painful work.

In this paper, we propose a novel method of ranking software components, called *Component Rank*, based on analyzing actual use relations of components and propagating the significance through the use relations.

There have been many researches related on software reusability[6, 9, 17]; however, the significance by actual usage has not been explored yet. The idea behind Component Rank originates from computing fair impact factors (called *influence weights*) of published papers[19]. This approach has been extended to ranking Web documents in the Internet[18].

We will present the Component Rank model for ranking software components, and show a system for computing Component Rank. In this model, a collection of software components is represented as a weighted directed graph whose nodes correspond to the components and edges correspond to the usage relations. Similar components are clustered into one node so that effect of simply duplicated nodes is removed. The nodes in the graph are ranked by their weights which are defined as the elements of the eigenvector of an adjacent matrix for the directed graph.

The Component Rank computation system targets Java source files containing a class definition as components. This system has been applied to various collections of Java programs, such as JDK, software engineering tools developed by ourselves, business applications and their framework developed by a mid-size software company, and a collection of XML and other tools.

The results show that stable classes frequently invoked or inherited by other classes have generally high ranks. Examples of such classes are for fundamental and standard data structures and for typical exception handers. Non-standard and special classes are listed with low ranks in many cases.

Since this empirical validation of Component Rank is very promising, we are currently developing Software Product Archiving, analyzing, and Retrieving System called *SPARS*, by which software engineers will interactively retrieve software components with various queries.

In Section 2, we will propose Component Rank model. Section 3 will show an implementation of the Component Rank model. The results of applications will be presented in Section 4. Various issues on the model and the implementation will be discussed in Section 5. Finally, we will conclude our discussion with future work in Section 6.

2. Component Rank Model

2.1. Component Graph

Software systems are modeled by a weighted directed graph, called a *Component Graph*. A node in a graph represents a software component, and a directed edge e_{xy} from node x to y represents a *use relation* meaning that component x uses component y.

Here, we do not restrict our discussion to a specific kind of component in the graph. A component may be a source-code module, a link library, or one section of document. The following discussion will hold for any kind of nodes. Also, the use relation is left unspecified here for the extendibility of the model. In Section 4, we will show our concrete implementation for those abstracted nodes and edges, such that the components are Java class files, and that the use relations are the class inheritance, method invocation, and abstract class implementation.

A software system is generally modeled with a component graph that is weakly connected (assuming that there is no redundant component). A set of software systems is also modeled with one component graph. This graph is disconnected if there are no sharing components.

Figure 1 shows a component graph for two software systems X and Y. X consists of 5 components A – E, and Y consists of 4 components F – I. This graph also shows that component C uses both A and B, and D and E use C. Also, H and I use G, and G and F mutually use one another.

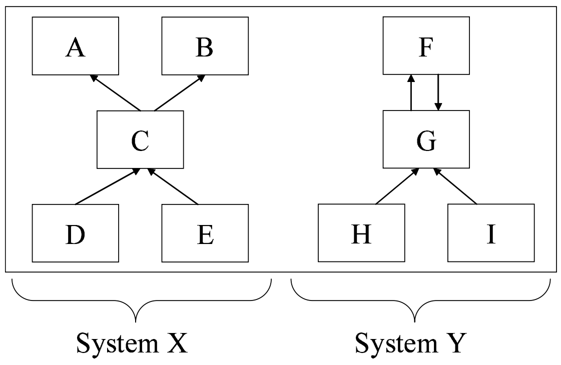

Figure 1. An example of component graph

2.2. Weight of Node

Each node v in component graph G has a non-negative weight value $w(v)$ where $0 \leq w(v) \leq 1$.

Definition 1 (Total Weights of Nodes) *For simplicity of following calculation, we assume that the sum of the weights of all nodes in G is 1, i.e.,*

$$\sum_{v \in G} w(v) = 1$$

Computing the weight for each node under computation policies described below is our objective of this work. The order of the nodes sorted by the weights is called *Component Rank* of the components.

We introduce several definitions to define $w(v)$ for component graph $G = (V, E)$.

Definition 2 (Weight of Edge) *For computation of the weights of nodes, we introduce the weight $w'(e_{ij})$ of an edge $e_{ij} = (v_i, v_j)$, such that*

$$w'(e_{ij}) = d_{ij} \times w(v_i)$$

Figure 2 (a) depicts this definition. Here, d_{ij} is called a *distribution ratio*, where $0 \leq d_{ij} \leq 1$ and the total of d_{ij} for each j is 1. If there is no edge from v_i to v_j, $d_{ij} = 0$. The distribution ratio d_{ij} is used for determining the forwarding weights of v_i to an adjacent node v_j.

Definition 3 (Weight of Node) *The weight of a node v_i is defined as the sum of the weights of all incoming edges e_{ki}, such that*

$$w(v_i) = \sum_{e_{ki} \in \mathrm{IN}(v_i)} w'(e_{ki})$$

Here, IN(v_i) is the set of the incoming edges of v_i. Figure 2 (b) shows this definition.

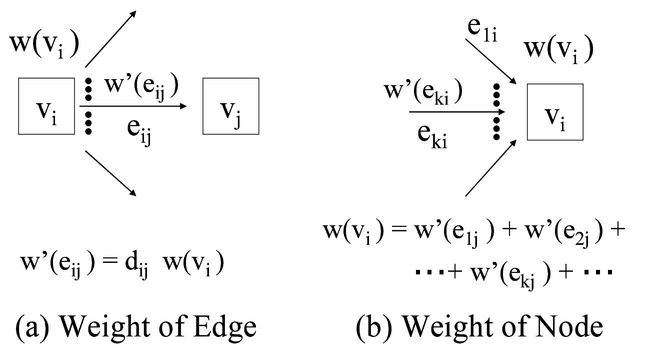

Figure 2. Definition of weights

2.3. Computation of Weights

Based on these definitions, we have $n(= |V|)$ simultaneous equations for $w(v_i)$,

$$w(v_i) = \sum_{e_{ki} \in \mathrm{IN}(v_i)} d_{ki} \times w(v_k)$$

Assume that W is a vector of node's weights,

$$W = \begin{pmatrix} w(v_1) \\ w(v_2) \\ \vdots \\ w(v_n) \end{pmatrix}$$

Also, D is a matrix of the distribution ratios,

$$D = \begin{pmatrix} d_{11} & d_{12} & ... & d_{1n} \\ d_{21} & d_{22} & ... & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & ... & d_{nn} \end{pmatrix}$$

So the simultaneous equations can be rewritten by,

$$W = D^t W \tag{1}$$

where D^t is the transposed matrix of D.

Together with Definition 1, formula (1) can be solved by computing the eigenvector with eigenvalue 1.

Instead of computing the eigenvector, we can also compute the weights of each node by a repeated computation such that, we give initial ad-hoc weights to each node(e.g., $1/n$ to each node), and then propagate them to adjacent nodes through directed edges. The weights are repeatedly recomputed until the all weights become stable.

Figure 3 shows a component graph with computed weights. v_1 has two outgoing edges, and weight 0.4 is evenly divided to two outgoing edges with 0.2 each (i.e., $d_{12} = d_{13}$ = 0.5). v_3 has two incoming edges, each with weight 0.2, so that the weight of v_3 is 0.4.

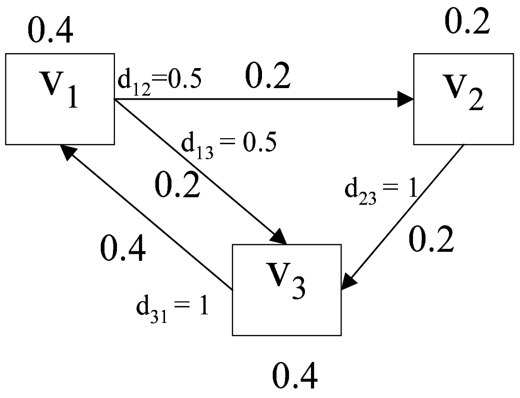

Figure 3. An example of stable weights assigned to nodes and edges

If we assume that the movement of software developer's focus on the target components is represented by a probabilistic state transition, the component graph is a Markov chain model. Thus, computing the weights of the nodes in the graph corresponds to getting a stationary distribution of the chain.

2.4. Convergence of Computation

In the case that a node v_i has no outgoing edge, all distribution ratios d_{ix} become zero. This does not satisfy the requirement of the total sum of the distribution ratio.

Also, if the graph is not strongly connected as shown in Figure 4 (a), the weight of v_1 is repeatedly added to the weight of v_2, causing non-termination in the repeated computation.

In these cases, the eigenvector cannot be solved either. To guarantee the solution of the simultaneous equations, i.e., the termination of repeated computation of the weights, we introduce pseudo use relations between all nodes as shown in Figure 4 (b), and define an amended distribution ratio for each node as follows.

Definition 4 (Amended Distribution Ratio)

$$d'_{ij} = \begin{cases} p \times d_{ij} + (1-p)/n & \textit{if } v_i \textit{ has outgoing edge } e_{ij} \\ 1/n & \textit{if } v_i \textit{ has no outgoing edges} \end{cases}$$

Here $p(0 < p < 1)$ is the ratio between the weight of real use relations and that of pseudo use relations, and we generally employ a fairly large value such as 0.85 as discussed later.

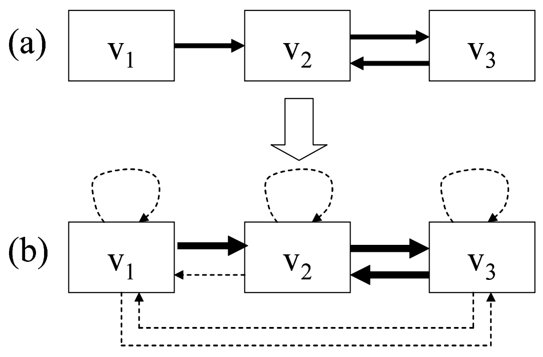

Figure 4. Introduction of pseudo use relation for convergence

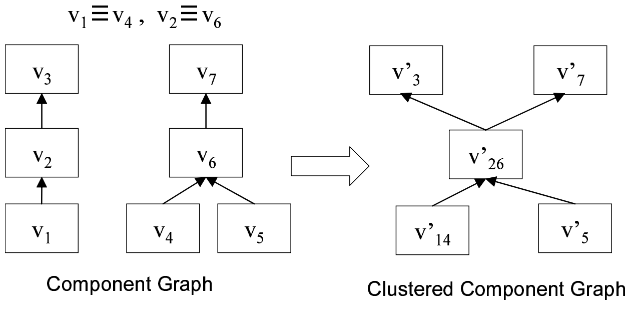

Figure 5. An example of clustered component graph

By using d'_{ij} instead of d_{ij}, we can always compute the weight for a given component graph. This pseudo use relation can be considered as a possible implicit reference of a component to all components (including itself).

2.5. Clustering Components

Software systems are often made by reusing already developed components. Some components are simply copied from previous systems, or some are constructed from those with minor or major modifications.

In the above mentioned method, those reused components are represented as multiple nodes in a component graph, and their weights are computed independently.

We would want to identify the similarity of components and to feed back the similarity information to the computation, in order to obtain more practical weight values.

Assume that we can measure similarity $S(v_1, v_2)$ of any two nodes (components) v_1 and v_2 in a component graph $G = (V, E)$, where $0 \leq S(v_1, v_2) \leq 1$, $S(v, v) = 1$, and $S(v, w) = 0$ if v and w are totally different.

Here, we say that an equivalence relation of v_1 and v_2 exists when $S(v_1, v_2) \geq t$, where t is a criterion of the equivalence. This equivalence relation partitions V into a set of equivalence classes, composing the quotient set V'.

We define the *clustered component graph* $G' = (V', E')$ for G as follows.

Definition 5 (Clustered Component Graph)
$G' = (V', E')$ such that V' is the quotient set of V, and $E' = \{(v'_i, v'_j) | (v_i, v_j) \in E\}$ where v'_i and v'_j are equivalence classes involving v_i and v_j, respectively.

Figure 5 shows an example of a clustered component graph. Two weakly connected subgraphs in the component graph are merged into one weakly connected graph as the clustered component graph. As shown in this example, there are cases such that several independent subgraphs in the component graph are merged into one, and the characteristics of each subgraph can be propagated to other subgraphs.

Using the clustered component graph instead of the component graph, we will compute the weights of each node as the above mentioned approach, including the amendment of the distribution ratio. We simply call *component graph* for *clustered component graph* hereafter, if there is no ambiguity.

3. Component Rank System

3.1. Implementation of Component Rank Model

Based on the Component Rank model discussed in Section 2, we have designed and implemented a system to compute Component Ranks for Java programs. The following features the implementation.

Component: A single Java source file with .java extension is considered as a component. In Java, it is inherently easy to extract components, since each class definition is independently defined as a single file with .java extension. Internal classes are not considered as components here.

Use relation: Class inheritance relation, method invocation, and abstract class implementation are the use relations. Here, we employ only statically detectable ones from the source programs.

Ratio p between real and pseudo use relations: We use p=0.85 which means that the total weight value 0.15 is assigned to the pseudo use relations.

Distribution ratio: The distribution ratios of the real use relations, which go out from one node, are equally

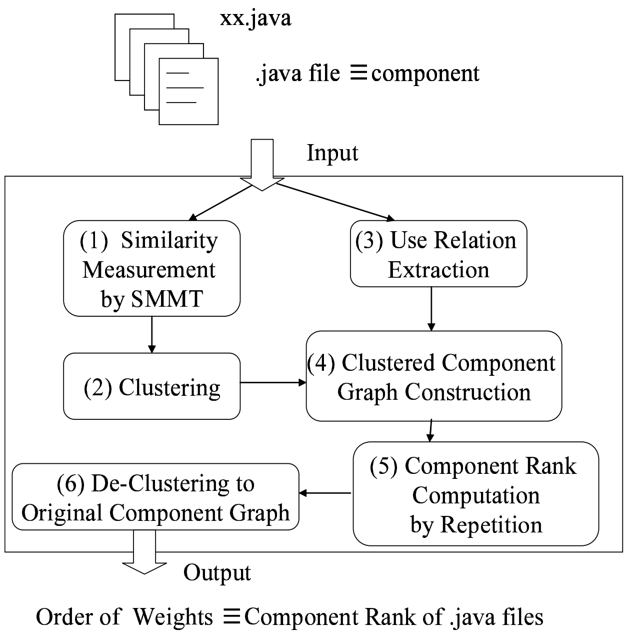
xx.java
.java file ≡component
Input
(1) Similarity Measurement by SMMT
(3) Use Relation Extraction
(2) Clustering
(4) Clustered Component Graph Construction
(5) Component Rank Computation by Repetition
(6) De-Clustering to Original Component Graph
Output
Order of Weights ≡Component Rank of .java files

Figure 6. Architecture of the component rank system

the same values. For example, if a node have 5 outgoing edges, the distribution ratio evenly divided for each edge is 0.17. The total of 5 edges is 0.85, and the rest 0.15 is the total of the pseudo use relations, which imaginarily go out to all nodes in the component graph.

Similarity t: Similarity metrics for source codes, proposed in [23], is adopted, and its measurement tool SMMT is used. The metrics is based on a combination of a fast code-clone detection by *CCFinder*[14] and a file-difference detection by *diff*[5]. The number of shared source code lines is divided by the number of total lines, producing the similarity value. We employ the similarity criterion $t = 0.80$ here.

3.2. Architecture of the Component Rank System

Figure 6 shows the architecture of our first prototype of the Component Rank system.

(1) The similarity of any two components (files) is computed by SMMT tool, for M input components.

(2) Two components with similarity higher than or equal to t=0.8 are merged into one component. By this clustering, we get $N(\leq M)$ equivalence classes.

(3) The syntax analyzer parses the java source files, and extracts class inheritances, method invocations, and abstract class implementations, as the use relations. The analyzer has been constructed by using Java syntax library ANTLR[1].

(4) The clustered component graph is constructed as a form of N x N amended distribution ratio matrix.

(5) The weights of nodes are computed by propagating initial ad-hoc values through edges until we get the convergence. This method is mostly faster than solving the eigenvector by a math library.

(6) The weighted nodes in the clustered component graph is mapped back to the original components, so that we can see the order of the weight of each component in M, i.e., Component Rank of the Java file, as the output.

The system computes Component Rank fairly efficiently even for a large set of programs. It takes about 7 minutes to compute Component Rank for JDK 1.3.0 (1877 components of totally 575,000 lines.) on a PC workstation (Pentium IV, 2GHz, 2GBytes memory).

4. Experiments

We have applied the Component Rank system to 4 sets of Java programs as follows.

4.1. JDK 1.3.0

All source programs of Java 2 Software Development Kit, Standard Edition 1.3.0[12] (simply referred to JDK 1.3.0 here) are the target of the first application. It is composed of 1877 .java files of totally 575,000 lines of code in Java. These files include the classes which are very important and fundamental ones to develop various Java applications.

Table 1 shows the resulting Component Rank values for each file, listed from the highest rank to the lowest one. The highest one, java.lang.Object class, is the superclass of any class in Java, so that this class is used directly or indirectly by any class, causing it on the top of the ranking.

Other highly ranked classes are also fundamental ones that are possibly invoked or inherited from many other classes. The 3rd class, java.lang.Throwable, is the superclass of any error or exception handlers so that it is used by many classes with error or exception handling.

There are 622 classes with the lowest (1256th) rank. These classes are not used by any other classes at all.

The overall result of Component Rank for JDK 1.3.0 matches to our intuition such that very general and core classes are ranked high, and specific and independent classes are ranked low.

Table 1. Component Rank for JDK 1.3.0

C.Rank	Class Name	Weight
1	java.lang.Object	0.16126
2	java.lang.Class	0.08712
3	java.lang.Throwable	0.05510
4	java.lang.Exception	0.03103
5	java.io.IOException	0.01343
6	java.lang.StringBuffer	0.01214
7	java.lang.SecurityManager	0.01169
8	java.io.InputStream	0.01027
9	java.lang.reflect.Field	0.00948
10	java.lang.reflect.Constructor	0.00936
⋮	⋮	⋮
1256	sunw.util.EventListener	0.00011
1256	⋮	⋮

Table 2. Component Rank for SE tools

C.Rank	Class Name	Weight
1	antlr.Token	0.10727
2	antlr.debug.Event	0.06189
2	antlr.debug.NewLineEvent	0.06189
4	antlr.collections.impl.Vector	0.05434
5	jp.gr.java_conf.keisuken. text.html.HtmlParameter	0.05246
6	jp.gr.java_conf.keisuken. net.server.ServerProperties	0.03699
7	Jama.Matrix	0.01564
8	jp.gr.java_conf.keisuken. util.IntegerArray	0.01390
8	jp.gr.java_conf.keisuken. util.LongArray	0.01390
10	jp.ac.osaka_u.es.ics.iip_lab.metrics. parser.IdentifierInfo	0.01365
⋮	⋮	⋮
418	cktool_new.examples.Main	0.00050

4.2. Collection of SE Tools

We have applied the Component Rank system to a collection of software engineering tools developed in our lab. There are 582 files in total. The following lists their overview.

C-K Metrics Measurement Tool 1: This tool measures C-Kmetrics[4] of Java programs. The package name is cktool, and the tool uses a syntax analyzer ANTLR[1].

C-K Metrics Measurement Tool 2: This is an upgraded tool of above, and is named cktool_new.

Component Rank System: A version of Component Rank system itself is also targeted. It uses library ANTLR, a matrix computation library JAMA[11], and a general utility library Caffe Cappuccino Class Library[3]. The package name is jp.ac.osaka_u.ics. iip_lab.metrics.

Libraries: The source files of above mentioned libraries, ANTLR, JAMA, and Caffe Cappuccino Class Library (the package name is jp.gr.java_conf.keisuken) are included as the target. JDK is excluded here.

Table 2 shows the result of those tools. There are two of the second classes, antlr.debug.Event and antlr,debug.NewLineEvent, having the same weights. These two are merged into a single node in the clustered component graph because of their similarity. Also, two classes of the 8th rank are merged into one.

We have noticed that library classes are generally in higher ranks, and our developed application classes are generally in lower ranks. Also, we knew that many data container classes are ranked high, such as antlr.Token at the first, and antlr.collections.impl.Vector at the 4th rank.

This result also suggests that Component Rank is an indicator of generality and specialty of classes from the view point of usage from other classes.

4.3. Framework and Applications Developed by a Software Company

Daiwa Computer, located in Osaka, Japan, is a software company with about 180 engineers. In that company, various kinds of new technologies in Software Engineering are actively studied and introduced. It holds ISO9001:2000 certificate and CMM Level3 assessment.

In that company, a shared framework for Java applications has been constructed, and various business applications have been developed on this framework.

Using this framework, five Web-based data management applications have been developed. Those applications have similar features but have distinction in their implementation such as databases and their interfaces (SQL, DB2, Oracle, ...).

These five applications and the framework itself are the target of the ranking. They contain 1538 components in total, and these components were clustered into 339 nodes.

The observation we got by this experiment is very similar to the previous two experiments. The classes defined in the framework generally have high ranks. For example, from the first to 10th ranked classes are for the framework. This means that those classes are frequently used by other classes. Also, we knew that classes defining data structures and their containers are highly ranked. The first ranked class is the definition of a record class for database management, which is defined in the framework. These results confirm

Table 3. Search Result Sorted by Component Rank

Order (C.Rank)	Class Name	weight
1(67)	enhydra3.1...dom.Node	0.029110
2(169)	saxon7_0...saxon.om.NodeInfo	0.000969
3(275)	saxon7_0...saxon.pattern.NodeTest	0.000437
4(316)	enhydra3.1...dom.DocumentImpl	0.000368
5(355)	saxon7_0...saxon.pattern.Pattern	0.000324
6(382)	saxon7_0...saxon.Controller	0.000296
7(437)	enhydra3.1...xslt.XSLTEngineImpl	0.000241
8(446)	enhydra3.1...dom.ElementImpl	0.000235
9(500)	saxon7_0...saxon.style.StyleElement	0.000202
10(506)	saxon7_0...saxon.tree.NodeImpl	0.000198
⋮	⋮	⋮
125(4441)	enhydra3.1...FuncID	0.000029
125(4441)	⋮	⋮

our approach, in the sense that core and fundamental components can be easily identified by the rank.

4.4. Searching Components with Component Rank

The rank values computed by our method are generic ones in the sense that any kind of components is ordered in one dimension. We would usually want to search a specific kind of components in which we are interested, and also we may want to sort the found components in the order of their ranks.

To have the first impression of such a search, we have ranked 7171 Java files for text editors *JEDIT* and *jext*, an application server *Enhydra*, an XSLT processor *saxon*, and a Gnutella client *phex* (all these handle XML documents), found in SourceForge[21], in addition to JDK 1.3.0 and the 4 tools shown in Section 4.2.

For these, we searched components with a keyword "getNodetype" by using UNIX tool *grep*, excluding the components having the keyword only in comments. We had 181 matched components, and they are sorted by Component Ranks, as shown in Table 3.

Method "getNodetype" is a method to get the information of the kind of the node in the DOM tree. At the first and second in the sorted order, the definitions of that method appear. Other components use that class. High-ranked components are mainly for the operation of elements and for the analysis of a style sheet, which are fairly generic operations. On the other hand, many low-ranked components are for non-generic and specific purpose ones such as classes for interpreting and performing the contents described by XML.

We would think that this is a very encouraging result, since the high-ranked components are ones which should be directly referred to or should be associated when we build codes related to DOM trees.

5. Discussions

5.1. Weight Computation

In our Component Rank model, the weights of nodes are propagated to other nodes through edges. We would consider another simpler model such that the weight of a node is determined by the number of incoming edges. This alternate model can be easily computed without complex computation for the convergence (or the eigenvector as we do).

However, this alternate model is very fragile to local special references. Consider that a component A is 10 times used by components B1 to B10, and B1 to B10 are not used by any component at all. In the alternate model, the weight of A becomes 10. In the Component Rank model, the weight of A is relatively determined by other components, but A is not ranked high as the alternate model, since the weights of B1 to B10 are relatively low. Thus, we consider that our model is very stable for such local references.

This argument is a similar one such as a comparison of Web search engines between Google and other simple search engines only counting incoming references[18]. The simple engines could easily cheated by intentionally-made local reference links, but Google is resistant to such links.

5.2. Clustering Policy

The clustering similar nodes is an important characteristics of our Component Rank model. If we would model multiple software systems in a component graph without clustering, the graph might be composed of multiple disconnected subgraphs and the weights would be independently computed inside each subgraph, without circulating weights over subgraphs (except for small interaction by the pseudo use relations).

The clustering has an effect that the weights of one software system are propagated to other systems through clustered nodes, resulting in Component Rank as a global ranking in all the software systems.

We have taken a clustering policy such that first similar nodes are clustered then the weight of each node is computed. An alternate policy would be that first we compute the weight of each node in the original component graph, then we cluster the similar nodes by adding similar node's weights as the clustered node's weight.

In our policy, simply copied and used components are not counted repeatedly. Consider a simple case shown in Figure 7 (we do not consider the pseudo use relations for simplicity here). In this case, components A and B are duplicated. In our policy, the result weights for A' and B' are 1/4, while those are 1/3 in the alternate policy. This means that in our policy the existence of simply copied components does not affect the resulting weight. In the alternate policy, if the number of copied components grows, the resulting weights for those components increase.

We believe that it is important not to count simply duplicated components, but to count modified and labored components.

Consider another case shown in Figure 8. In this case, component A is reused, but B is replaced with C. In the result, the weight of A' is 2/5 in our policy, which is higher than 1/3 in the alternate policy. The weight of A' in the alternate policy is 1/3, the same value as the case in Figure 7, even with its structural change.

These examples sharply present that the duplication with modification is properly counted in our policy, but it is not in the alternate policy.

5.3. Similarity Criterion t and Pseudo Use Relation Ratio p

We have investigated different t from 0.10 to 0.90, and knew that the resulting ranks are fairly insensitive to t[24]. There are some components which are clustered with 0.70 or lower, but not with 0.80 or higher. Exception handlers in JDK are examples of those. We thought that such components should be counted separately so we have employed t=0.80 here.

The ratio p between real and pseudo use relations has been explored in detail[7]. We knew that the resulting weight values are affected by p, but that the resulting Component Ranks are insensible to p. The ranks are fairly stable even if we have changed p from 0.75 to 0.95, so we have chosen p=0.85 here.

5.4. Distribution Ratios of Node

In the current implementation of the Component Rank system, we have employed an equal distribution ratios for any outgoing edge from one node. We can consider another policy such that we give some priority to specific outgoing edges which are considered to deserve. At this moment, we do not have a clue to determine the priority, and we think this is a further issue. However, our intuition is that if the component graph becomes huge enough, the priority parameter of distribution ratios would not affect to the result ranks so severely, as we have investigated variations of t and p above.

We have defined the Component Rank model such that the weight of a node is divided and distributed to the outgoing edges so that the total weight of the outgoing edges is the weight of the node (assuming no pseudo edges). We can consider another model where the weight of a node is directly the weights of each edge, without division by the number of the outgoing edges. This model means that nodes with more outgoing edges have higher influence to other nodes totally, than nodes with less outgoing edges. We thought that this is unfair, but we might need a further investigation.

5.5. Scalability

In order to get practically useful results, we need to collect a huge number of source programs. In that sense, the scalability of the system is important. Currently, our Component Rank system works under about 8,000 components due to the limitation of the similarity computation. The current similarity computation is based on a pair-wise execution of *diff*[5] for any two components. This should be changed to a more efficient method such as classification of components by characteristic metrics.

Compared to the similarity computation, the convergence computation is a much lighter process, and we think that it would not limit the scalability.

5.6. Related Works

The Component Rank model proposed here is based on our original intuition, such that, from a huge number of collections of legacy software, we might be able to effectively search software components, pieces of program codes, program patterns, or abstracted algorithms, by using a similar approach as Google[2, 8, 18] for Web pages, together with various program analysis techniques. Google computes the ranks (called *PageRanks*) for HTML documents available in the Internet. The resulting PageRank is practically very useful. A related analyses on the Web links have been discussed in detail in [16].

The Google's approach can be considered as an HTML extension of a method proposed for counting impact of publications, called *influence weight*[19]. The influence weight of a publication is determined by the sum of the weights of incoming references, as the Component Rank model does.

A major distinction of our model from PageRank and the influence weight is that our approach explores similarity between components before the weight computation. This is definitely important for software components. There would be so many copied or slightly modified software components when we simply collect software systems. Without proper handling of those similar components, we could not get reasonable and practical ranks.

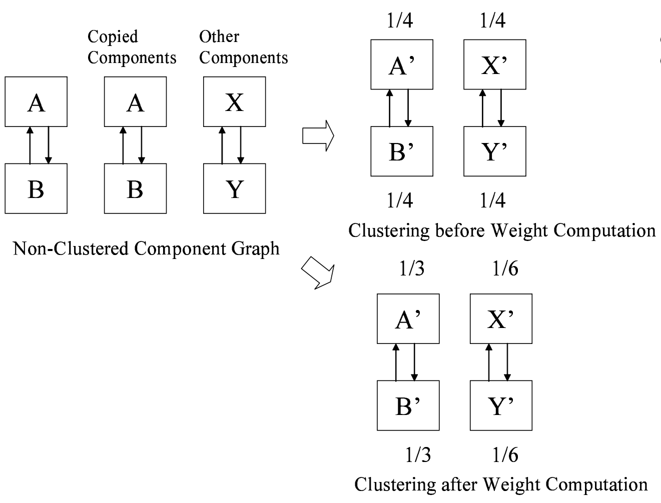
Copied Components
Other Components
A
B
A
B
X
Y
Non-Clustered Component Graph
1/4
1/4
A'
X'
B'
Y'
1/4
1/4
Clustering before Weight Computation
1/3
1/6
A'
X'
B'
Y'
1/3
1/6
Clustering after Weight Computation

Figure 7. Effect of simple copied components

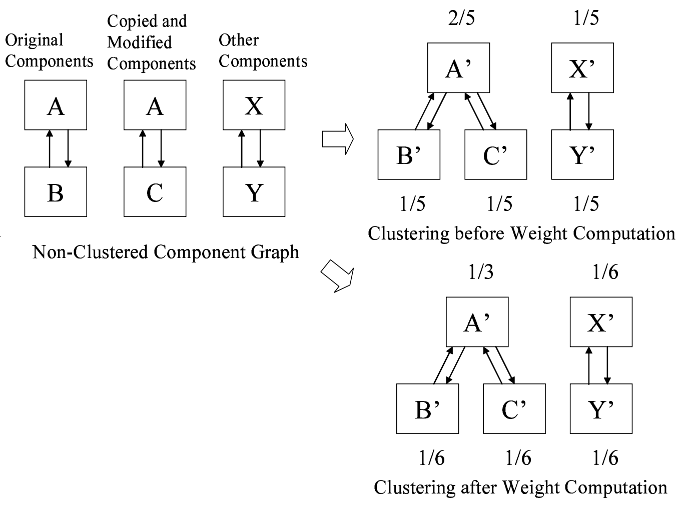
Original Components
Copied and Modified Components
Other Components
A
B
A
C
X
Y
Non-Clustered Component Graph
2/5
1/5
A'
X'
B'
C'
Y'
1/5
1/5
1/5
Clustering before Weight Computation
1/3
1/6
A'
X'
B'
C'
Y'
1/6
1/6
1/6
Clustering after Weight Computation

Figure 8. Effect of copied and modified components

Our Component Rank is a true measure of software reuse. There have been many researches on software reusability, and many literatures have been already published[9, 17]. Software reuse is a promising approach to improve the efficiency of software development and the quality of developed software[10, 15].

Many methods to measure reusability of software components have been proposed. Etzkorn's approach[6] is that various static metrics for C++ classes (components), such as cohesion and complexity, are measured and these values are normalized and added. Yamamoto's approach[22] is that class interface information is used to determine reusability. These approaches are based on measuring static properties of components.

Our approach does not measure the property of components, but only uses relation among components. We believe that the overall structure of the components represents much properly the usage history of the component, rather than the metrics of the components. In our model, if a component is repeatedly reused (by not simple copying), the result rank will be higher. However, the property measures are not affected by the repeated reuse.

5.7. SPARS

Using the Component Rank system as a core ranking engine, we are currently developing Software Product Archiving, analyzing, and Retrieving System called *SPARS*. Figure 9 shows SPARS architecture. Various Java source programs are collected, and they are stored in the raw component archive. Those components are ranked by the Component Rank engine and stored as the ranked component archive.

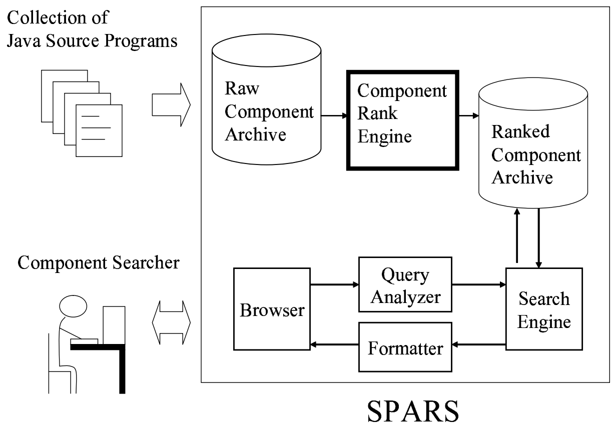

Figure 9. Architecture of SPARS

A component searcher, who is trying to build a software system, will give SPARS queries for some typical definition or typical usage of a class to build, by keywords or portions of source codes. These queries are analyzed and given to the search engine. It retrieves the ranked component archive. The matched results are listed by the ranks, and are returned to the component searcher.

6. Conclusion

We have proposed a novel ranking method of software components, Component Rank, and shown the first implementation of Component Rank. Using the implemented system, various Java programs have been ranked, and the characteristics of Component Rank have been investigated.

Although the results might be still preliminary, we believe that this method is very promising. Using the Component Rank method, we are currently developing a software component retrieval system, SPARS, which will be used in various situations of software development, such as searching, exploring, checking, investigating, reminding or referring to software components, as we use dictionaries and libraries when writing a composition. SPARS might be considered as a Google-like system for software engineers.

There are several further issues. The validity of Component Rank will be explored further.

One approach is to evaluate the rank values directly. However, since we do not know any objective measure inherently correlating to Component Rank, we would use, for a validation, a subjective measure such as questioners to software experts.

Another approach would be to investigate precision and recall of the search results. To do this, we will need to build a proper experimental setting, including correct answers to test queries.

In the Component Rank model, we employed only statical use relations. It is possible to extend to use relations such as method invocation and dynamic class binding. This approach would be more preferable since we can compute the ranks of components without their source codes. Distinction between statical and dynamical ranks is a very intriguing research issue to explore.

We have discussed the Component Rank model only for component retrieval; however, we are planning to apply Component Rank to automatic software architecture composition. There are many literatures for the structuring software architectures from the results of statical analyses of source codes and libraries, or from the results of dynamic analyses of object code execution. We think that those analysis results could be more stabilized by the propagation and convergence computation as the Component Rank model.

Our Component Rank system is currently implemented to accept Java programs only. We will extend to other procedural languages such as C and C++. To do this, we have to define the component and use relation for those languages. Functions and procedure invocations would be practically feasible candidates, but we need further investigation and validation.

Acknowledgments

We express our great thanks to Daiwa Computer Co., Ltd. for the data collection. We also thank Prof. Kumiyo Nakakoji of University of Tokyo / SRA-KTL and Dr. Yunwen Ye of University of Colorado / SRA-KTL for their helpful comments and discussions.

This project is supported by Japan Science and Technology Corporation, Research and Development for Applying Advanced Computational Science and Technology.

References

[1] ANTLR. “http://www.antlr.org/”.

[2] S. Brin and L. Page. “The Anatomy of a Large-scale Hypertextual Web Search Engine”. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[3] Caffe Cappuccino class library. ”http://cappuccino.ne.jp /keisuken/java/capp/api/overview-summary.html”.

[4] S. Chidamber and C. Kemerer. “A Metrics Suite for Object-Oriented Design”. *IEEE Transactions. Software Engineering*, 20(6):476–493, 1994.

[5] Diffutls. “http://www.gnu.org/software/diffutils/”.

[6] L. H. Etzkorn, W. E. H. Jr., and C. G. Davis. “Automated Reusability Quality Analysis of OO Legacy Software”. *Information and Software Technology*, 43(5):295–308, 2001.

[7] H. Fujiwara. “ A New Reusability Metric Based on Reuse Results and Similarity among Programs”. Master’s thesis, Dept. of Informatics and Mathematical Science, Osaka University, 2002. (In Japanese).

[8] Google. “http://www.google.com”.

[9] J. Guo and Luqi. “A Survey of Software Reuse Repositories”. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 92–100, Edinburgh, Scotland, April 2000.

[10] S. Isoda. “Experience Report on a Software Reuse Project: Its Structure, Activities, and Statistical Results”. In *Proceedings of 14th International Conference on Software Engineering*, pages 320–326, Melbourne, Australia, 1992.

[11] JAMA : A Java Matrix Package. “http://math.nist.gov/ javanumerics/jama/”.

[12] Java 2 Software Development Kit, Standard Edition 1.3.0. “http://java.sun.com/j2se/”.

[13] JUMBO! “http://www.jumbo.com/”.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. “CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code”. *IEEE Transactions. Software Engineering*, 28(7):654–670, 2002.

[15] B. Keepence and M. Mannion. “Using Patterns to Model Variability in Product Families”. *IEEE Software*, 16(4):102–108, 1999.

[16] J. Kleinberg. “Authoritative Sources in a Hyperlinked Environment”. *Journal of the ACM*, 46(5):604–632, 1999.

[17] C. Krueger. “Software Reuse”. *ACM Computing Surveys*, 24(2):131–183, 1992.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd. “The PageRank Citation Ranking: Bringing Order to the Web”. Technical Report of Stanford Digital Library Technologies Project, 1998.“ http://www-db.stanford.edu/backrub/ pageranksub.ps”.

[19] G. Pinski and F. Narin. “Citation Influence for Journal Aggregates of Scientific Publications: Theory, with Application to the Literature of Physics”. *Information Processing and Management*, 12(5):297–312, 1976.

[20] E. S. Raymond. "The Cathedral and the Bazaar". "http://www.tuxedo.org/ esr/writings/cathedral-bazaar/".

[21] SOURCEFORGE.net. "http://sourceforge.net/".

[22] H. Yamamoto, H. Washizaki, and Y. Fukazawa. "The Proposal and Verification of Component Metrics Based on the Reuse Characteristic". In *Workshop of Foundation of Software Engineering (FOSE2001)*, 2001. (In Japanese).

[23] T. Yamamoto, M. Matsusita, T. Kamiya, and K. Inoue. "Measuring Similarity of Large Software Systems Based on Source Code Correspondence". Technical Report of Dept. of ICS, IIP-03-03-02, 2002.

[24] R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsusita, S. Kusumoto, and K. Inoue. "Proposal of Reusability Evaluation Method Using Relations among Software Components". In *Proceedings of Software Symposium 2002*, pages 216–225, Matsue, Japan, July 2002. (In Japanese).

[25] ZDNet. "http://www.zdnet.com".