Ranking Software Artifacts

Fabrizio Perin, Lukas Renggli, Jorge Ressia

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

***Abstract*—Reengineering and integrated development platforms typically do not list search results in a particularly useful order. PageRank is the algorithm prominently used by the Google internet search engine to rank the relative importance of elements in a set of hyperlinked documents. To determine the relevance of objects, classes, attributes, and methods we propose to apply PageRank to software artifacts and their relationship (reference, inheritance, access, and invocation). This paper presents various experiments that demonstrate the usefulness of the ranking algorithm in software (re)engineering.**

I. Introduction to Ranking

Reengineering platforms, such as Moose [1] and inFusion [2], as well as integrated development environments (IDEs), such as Eclipse and Smalltalk, do not list search results in a particularly useful order. Ordering search results alphabetically or according to traditional code metrics [3] is reasonable if the system is small and well understood. However, these approaches do not scale for large result sets or if the system is unknown.

We propose to use the PageRank algorithm [4] to assign weights to code entities relative to their importance in the system. PageRank was originally developed by Larry Page and Sergei Brin to rank search results of the Google internet search engine. The algorithm is inspired by the work of Eugene Garfield on citation analysis [5]. The idea has been applied in a variety of research fields, such as social networks, biogenetics, and lexical semantics [6]. In fact, the algorithm can be applied to any directed graph. However, to our knowledge, PageRank has not been used in the field of software analysis and development to determine the 'importance' of software artifacts.

The *PageRank algorithm* assigns an importance metric to a directed graph of websites (nodes) and navigation links (edges): the more links from 'important' sites point to a site the more 'important' that site gets. This iterative process is expressed in the following recursive formula:

$$r_{i,0} = \frac{1}{N}$$

$$r_{i,t+1} = \frac{1-d}{N} + d \sum_{\forall j \to i} \frac{r_{j,t}}{C_j}$$

The rank $r_{i,t+1}$ of a node i is calculated from the ranks of the previous iteration t of all the nodes $r_{j,t}$ that refer to node i. If node j points to C_j different nodes, then the rank $r_{j,t}$ is distributed evenly among all referenced nodes. N is the total number of nodes under analysis. The damping factor d is typically set to 0.85 and avoids that nodes that have no incoming references lose all their weight. In our experience, around 100 iteration steps t are sufficient to reach a stable ranking.

In the context of software systems the nodes are represented by software artifacts such as objects, packages, namespaces, classes, attributes, annotations, methods, and variables; and the edges by reference, containment, inheritance, access, instantiation, invocation, *etc.* We have implemented the PageRank algorithm in Smalltalk and applied it to several large case studies in Pharo Smalltalk and on the Moose reengineering platform. We discovered that PageRank is a useful metric and inexpensive to calculate. It can reliably point out relevant software artifacts.

The remainder of this paper is structured as follows: Section II discusses several usage scenarios of applying PageRank in the context of software (re)engineering. Section III discusses related work, and Section IV concludes and points out ideas for further work.

II. Ranking in Practice

In this section we apply the PageRank algorithm on the classes and methods of Smalltalk system, and then we explore various Java and Smalltalk frameworks using the Moose reengineering environment.

A. Ranking Classes and Methods in Smalltalk

For our experiments we took the default distribution of Pharo Smalltalk 1.1rc4 containing the core libraries and numerous additional development tools.

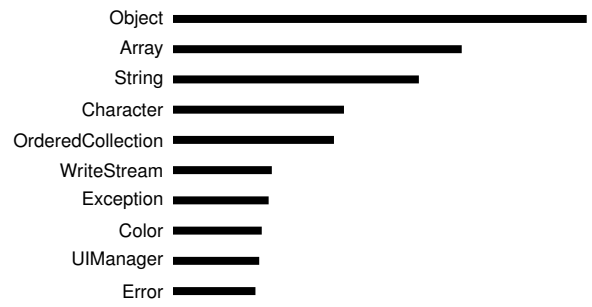

Fig. 1. Top 10 ranking classes in Pharo Smalltalk.

1) Ranking Classes: In our first experiment we use the 2 500 classes as nodes, and class-inheritance and class-references as edges. On a Macbook Pro 2.8 GHz the ranking of all the

classes in the system takes 4 seconds and yields the ranking depicted in Figure 1. The absolute numbers do not matter, only the relative ordering of the classes. The root class of the Smalltalk class hierarchy Object is clearly the most important class in the system, followed by common collection classes like Array, String, and OrderedCollection. Also the stream class WriteStream and several exception classes Error and Exception appear in the top 10.

We compared the set of 121 classes mentioned in the introductory book "Pharo by Example" [7] with the set of 121 top ranked classes returned by our approach. This results in an accuracy of 42% (F_1-score). We found that many classes discussed in the book are not used by the system itself and therefore do not appear in our ranking. We believe that if searches for classes would be sorted according to their rank, developers would find relevant results faster.

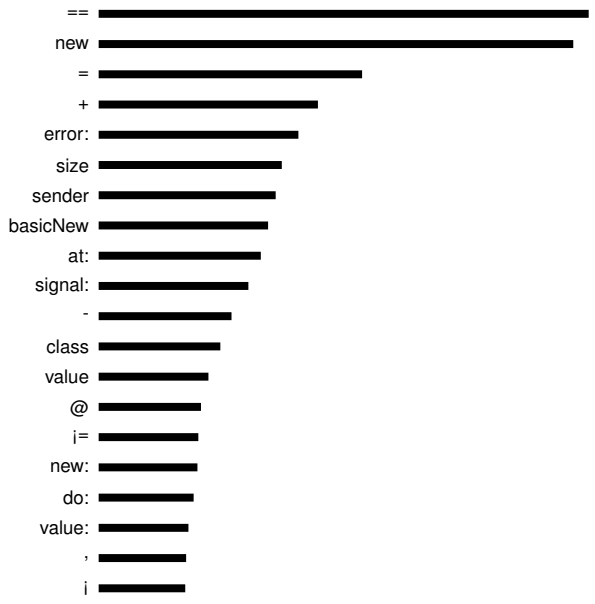

Fig. 2. Top 10 ranking method names in Pharo Smalltalk.

2) Ranking Methods: Similarly we rank all the method names. This dataset is much larger compared to the previous one. It takes into consideration over 25 000 method names (nodes) and over 180 000 method invocations (edges). The ranking of the method names takes about 25 seconds and the top 20 are listed in Figure 2. Again we can make some interesting observations: methods to compare objects ==, and =, are considered most important; followed by methods for object instantiation new, basicNew, and new:; and numerical operators such as +, -, ¡=, and ¡. We compared the set of 32 selector names that the designers of Smalltalk-80 [8] considered the most important and optimized with special bytecodes with the 32 highest ranked selectors of our study: this results in an accuracy of 56% (F_1-score).

B. Ranking Code Entities in Moose

In Moose we have a much more sophisticated model to represent code. Moose models software systems as a graph of objects independent of the language they are implemented in. This gives us the possibility to apply the ranking algorithm at different levels of granularity, *e.g.,* namespaces, packages, classes, attributes, or methods. For example, the query clientClasses returns all the classes a given code entity depends upon. The method considers class inheritance, class references, method invocations and attribute access; and it is independent of the analyzed programming language.

Moose does not provide a scalable way of highlighting the important parts of a system. For example, visualizations are of little use if the analyzed system is large, because the overview is proportionally large. Also code metrics or specific queries might be difficult to apply on unknown systems or yield many results.

The PageRank algorithm suggests that a code artifact that uses another code artifact casts a vote for that artifact. The tighter a code artifact is referenced the more important it is, and the more likely it is that we should have a look at it to understand the system. Similarly, code artifacts that are the least referenced can be interesting too: they can point us to parts of a system where the code is rotting.

To demonstrate the accuracy of the automatic ranking we run the algorithm on various well-known open-source Smalltalk and Java frameworks. The analysis uses the clientClasses to identify collaborating classes. The results are listed in Figure 3. The authors of the respective Smalltalk frameworks confirmed the correctness of the results, although they also reported some other core classes that did not show up in the top 5.

III. Related Work

a) Ranking Algorithms: Figure 4 lists the most common ranking algorithms known in the web mining community. The *Hubs and Authorities* or *Hyperlink-Induced Topic Search* (HITS) [9] algorithm identifies authoritative sources in a hyperlinked environment. The algorithm calculates two scores per entity, a hub and an authority metric. The *TrustRank* [10] algorithm ranks the quality of websites. It requires an initial seed of 'trustful' elements. The *Hilltop* [11] algorithm also requires an initial seed. Furthermore it is query specific, this means it requires a new ranking each time a query is issued.

Algorithm	Initial Seed	Query Specific
Hilltop	●	●
Hubs and Authorities	○	○
PageRank	○	○
Trust Rank	●	○

Fig. 4. Web mining algorithms and their properties. The column *Initial Seed* marks the algorithms that require an initial seed of 'important' entries; and the column *Query Specific* marks the algorithms that require a new ranking for each query.

From the presented ranking algorithms, PageRank has various advantages over the other algorithms: PageRank does not require an initial seed, something that is often not easy to provide if the data is unknown. Furthermore, PageRank calculates a single ranking for all elements, there is no need to re-run the calculation for specific queries. The time complexity

System	NOC	Time	Top 5 Classes
Apache-Ant 1.8.1	1490	5.46	Ejbjar, JbossDeploymentTool, AggregateTransformer, XMLResultAggregator, Tar
Apache-Ant 1.6.1	1148	3.03	Ejbjar, JbossDeploymentTool, MacroDef, ProjectHelperImpl, Main
ArgoUML 0.28.1	2200	3.10	Main, ProjectBrowser, ElementPropPanelFactory, InitUmlUI, GenericArgoMenuBar
Eclipse Core	1928	5.29	PojoProperties, Workspace, SaveManager, FileSystemResourceManager, DataBindingContext
Eclipse e4	10129	34.53	XWTDesigner, XWTDesignerMenuProvider, ModelEditor, AddEventHandlerAction, EventMEnuManager
JEdit 4.3	890	1.44	jEdit, ManagePanel, HyperSearchResults, PluginManager, Parser
JMeter	980	1.53	JMeter, UserParametersGui, AddUserAction, ProxyControl, Proxy
Famix	65	0.05	FAMIXAbstractFile, FAMIXPackageGroup, FAMIXClass, FAMIXNamespace, FAMIXPackage
Glamour	156	0.16	GLMMorphicRenderer, GLMBrowser, GLMCompositePresentation, GLMRenderer, GLMDynamicPresentation
Magritte	203	0.28	MAObject, MAContainer, MARelationDescription, MADescription, MAStringWriter
Mondrian	153	0.18	MOEasel, MOViewRenderer, MOAbstractLayout, MOFormsBuilder, MOLocalSpringLayout
OmniBrowser	468	1.27	OBBrowser, OBCommand, OBCommandCluster, OBColumn, OBMetaNode
PetitParser	61	0.05	PPParser, PPSequenceParser, PPChoiceParser, PPDelegateParser, PPWrappingParser
Seaside	321	0.58	WAPresenter, WASession, WAApplication, WAConfiguration, WAResponse

Fig. 3. Various large Java and Smalltalk open-source systems, the number of classes in the respective systems (NOC), the time in seconds it took to rank the classes within the Moose system, and the resulting top 5 ranked classes.

of the PageRank algorithm is linear with the number of nodes, thus it can handle large graphs efficiently. Distributed [12] and incremental [13] variants of the algorithm perform even better. The implementation of PageRank is simple and around 20 lines of code in Smalltalk. Its complete implementation can be found in the Appendix.

b) Ranking of Software Artifacts: Zaidman *et al.* [14] use the HITS algorithm to find key classes from dynamic system traces. The authors report an average recall of 90% and a precision of 50% (which is equivalent to an F_1-Score of 64%), however these values are directly tied to their fixed (but arbitrary) retrieval rate of 15%. Their approach is extremely expensive: collecting the traces, calculating the metrics and ranking the entities is in the range of hours; while our static approach directly ranks a Moose model. In many cases using dynamic information is not feasible because the resulting data is unstable (it highly depends on what code is executed), expensive to collect and analyze the data (typically execution traces are large), or not possible at all (the system to be analyzed is not executable).

It is interesting to compare the differences between Zaidman *et al.* approach and our case-study applied to 'Apache Ant' and 'Jakarta JMeter'. The only top ranked class identified by both approaches is the class Main, the starting point of 'Apache Ant'. We explain this difference in the fact that the dynamic approach measures only a specific usage scenario, while the static approach looks at the complete codebase as a whole.

Robillard [15] proposes an approach where methods and attributes are ranked using their structural relationship. The algorithm requires an initial seed of 'interesting' code artifacts and yields a possible set of related elements. The author states that their algorithm could be applied to classes too, but that this is outside the scope of the algorithm.

IV. Conclusions and Future Work

When displaying search results or when examining an unknown system it is often unclear what parts to look at first. In this paper we proposed the use of web mining techniques, namely the PageRank algorithm, to find key software artifacts. While more empirical evidence is needed, our initial case studies do demonstrate the usefulness and accuracy of such an approach. We have further demonstrated that PageRank is scalable to millions of nodes and that it can be applied to any graph of objects, not necessarily related to code artifacts.

As future work we plan to integrate our ranking algorithm into *Moose* to give a meaning to the order in which elements are listed. Also we envision to integrate it with the *Pharo IDE* to guide developers to more important results when browsing for senders, implementors or other code entities. In such a setup an incremental calculation of the ranks could be beneficial [13].

While in this paper we only looked at the highest ranked elements, we also foresee interesting usage of the *lowest ranked* elements. This could help the developer to identify code artifacts that are rarely used and that could be refactored or even removed from a system.

Implementation

The complete implementation of the PageRank algorithm is given below:

```
rank: aNodeCollection references: aReferencesBlock damping:
      aDampingFloat iterations: anIterationInteger
  | previous |
  previous := IdentityDictionary new: aNodeCollection size.
  aNodeCollection
    do: [ :node | previous at: node put: 1.0 / aNodeCollection size ].
  anIterationInteger timesRepeat: [
    | current |
    current := IdentityDictionary new: aNodeCollection size.
    previous keysAndValuesDo: [ :node :rank |
      | children |
      current
        at: node
        ifAbsentPut: [ 1.0 - aDampingFloat ].
      (children := aReferencesBlock value: node) do: [ :child |
        current
          at: child
          put: (current
            at: child
            ifAbsent: [ 1.0 - aDampingFloat ])
              + (aDampingFloat * rank / children size) ] ].
    previous := current ].
  ^ previous
```

A significantly more performant version of the PageRank algorithm can be found at www.squeaksource.com/pagerank/.

It comes with examples and unit tests. Compared to the version above its responsibilities are refactored between various objects.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010) and the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" (Project no. 2234, Oct. 2007 – Sept. 2010). We also like to thank Oscar Nierstrasz for his feedback an earlier draft of this paper.

REFERENCES

[1] S. Ducasse, T. Gîrba, and O. Nierstrasz, "Moose: an agile reengineering environment," in *Proceedings of ESEC/FSE 2005*, Sep. 2005, pp. 99–102, tool demo. [Online]. Available: http://scg.unibe.ch/archive/papers/Duca05fMooseDemo.pdf

[2] C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An integrated platform for quality assessment of object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, 2005, pp. 77–80, tool demo.

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. [Online]. Available: http://www.springer.com/alert/urltracking.do?id=5907042

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[5] E. Garfield, "Citation indexes in sociological and historical research," *American Documentation*, vol. 14, no. 4, pp. 289–291, 1963.

[6] A. Esuli and F. Sebastiani, "PageRanking WordNet synsets: An application to opinion mining," in *Annual Meeting-Association for Computational Linguistics*, vol. 45, no. 1, 2007, p. 424.

[7] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009. [Online]. Available: http://pharobyexample.org

[8] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Addison Wesley, 1989.

[9] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *ACM*, vol. 46, no. 5, pp. 604–632, 1999.

[10] J. P. Zoltán Gyöngyi, Hector Garcia-Molina, "Combating Web Spam with TrustRank," in *30th International Conference on Very Large Data Bases*, vol. 30, 2004, p. 587.

[11] K. Bharat and G. A. Mihaila, "Hilltop: A search engine based on expert documents," 2000.

[12] Y. Wang and D. J. DeWitt, "Computing PageRank in a distributed internet search system," in *30th International Conference on Very Large Data Bases*, 2004, p. 431. [Online]. Available: http://www.vldb.org/conf/2004/RS11P1.PDF

[13] S. Kamvar, T. Haveliwala, and G. Golub, "Adaptive methods for the computation of PageRank," *Linear Algebra and its Applications*, vol. 386, pp. 51–65, 2004.

[14] A. Zaidman and S. Demeyer, "Automatic identification of key classes in a software system using webmining techniques," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 387–417, 2008. [Online]. Available: http://www.st.ewi.tudelft.nl/~zaidman/publications/azaidmanJSME2008b.pdf

[15] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *10th European Software Engineering Conference*. ACM, 2005, p. 20. [Online]. Available: http://www.st.cs.uni-saarland.de/edu/recommendation-systems/papers/esecfse2005a-1.pdf