

Organizing and Visualizing Software Repositories Using the Growing Hierarchical Self-Organizing Map

SONGSRI TANGSRIPAIROJ AND M. H. SAMADZADEH*

Department of Computer Science, Faculty of Science

Mahidol University

Bangkok 10400, Thailand

E-mail: ccsts@mahidol.ac.th

**Department of Computer Science*

Oklahoma State University

Stillwater, OK 74078, U.S.A.

E-mail: samad@cs.okstate.edu

A software repository, a place where reusable components are stored and searched for, is a key ingredient for instituting and popularizing software reuse. It is vital that a software repository should be well-organized and provide efficient tools for developers to locate reusable components that meet their requirements. The growing hierarchical self-organizing map (GHSOM), an unsupervised learning neural network, is a powerful data mining technique for the clustering and visualization of large and complex data sets. The resulting maps, serving as retrieval interfaces, can be beneficial to developers in obtaining better insight into the structure of a software repository and increasing their understanding of the relationships among software components. The GHSOM, which is an improvement over the basic self-organizing map (SOM), can adapt its architecture during its learning process and expose the hierarchical structure that exists in the original data. In this paper, we demonstrate the potential of the GHSOM for the organization and visualization of a collection of reusable components stored in a software repository, and compare the results with the ones obtained by using the traditional SOM.

Keywords: software repository, self-organizing map, growing hierarchical self-organizing map, software reuse, data mining

1. INTRODUCTION

Software reuse, the process of developing new software systems from previously constructed software components, has been widely accepted as a promising means for the improvement of software quality, productivity, reliability, and maintainability [13, 18, 33]. For software reuse to be successful, it is necessary to set up and maintain a software repository containing a large number of reusable components over a wide spectrum of application domains. These reusable components should be classified and retrieved in a cost effective and efficient way. There should be tools for developers to find the desired reusable components quickly and easily, and hence to make better decisions in selecting the right components for reuse [14].

Received July 1, 2005; accepted November 24, 2005.

Communicated by Sung Shin.

Numerous methods have been proposed to organize software repositories and to facilitate the process of retrieving software components. Most existing methods are “either too ineffective to be useful or too intractable to be usable” [24] and seem to work properly for a small software repository. However, in a software repository that is possibly large and ever-growing, the process of specifying, locating, and retrieving reusable components can be complex and time consuming, and hence frustrating if the software repository is not well-organized.

The self-organizing map (SOM), an unsupervised learning neural network, is a powerful data mining technique for clustering and visualization of a large and complex data set [17]. Its most remarkable capability is to produce a mapping of high-dimensional input space onto a low-dimensional (usually two-dimensional) map, where similar input data can be found on nearby regions of the map. The resulting map offers a better insight into the interrelationships among the input data and helps identify clustering tendencies. The SOM has successfully been used in a variety of applications such as organizing massive document collections [16], analyzing financial data [7], clustering climate data [31], and classifying DNA sequences [25].

Achievements of the traditional SOM are limited due to two significant drawbacks. One drawback is that it has a fixed network architecture since the map size and the arrangement of the neurons have to be defined in advance. This may not be feasible for some applications and may result in a significant limitation on the final mapping [1, 3, 4, 11, 12]. The other drawback is that it lacks the ability to reflect hierarchical structure in the original data. Several dynamic SOM models, e.g., incremental grid growing [4], the growing cell structures [11], the growing grid [12], the hypercubical SOM [3], and the growing SOM [1], have been proposed to solve the first drawback. Differing from the other dynamic SOM models, the growing hierarchical self-organizing map (GHSOM) [9, 30] is able to overcome both drawbacks.

The GHSOM is a dynamic SOM model that can build a hierarchy of multiple layers of several growing SOMs, where the size of these SOMs and the depth of the hierarchy are determined during its learning process according to the requirements of the input data. By organizing maps in this way, the inherent hierarchical structure of data will be explicitly exhibited. The GHSOM has successfully been applied to organize large document archives [9, 30] and recognize handwritten digits [2].

The main objective of the work reported in this paper was to demonstrate the potential of the GHSOM for organization and visualization of a collection of reusable components stored in a software repository, and to compare its results with the ones obtained by using the traditional SOM.

The remainder of this paper is organized as follows. Section 2 provides a brief review of the previous related work concerning software repository organization and the existing applications of SOM to software repositories. Section 3 gives an introduction to the SOM, followed by a description of the GHSOM methodology in section 4. Section 5 explains about the experiments and the results. Section 6 is the conclusion of the paper.

2. RELATED WORK

This section provides a brief review of previous research on software repository or-

ganization and SOM applications to the organization and visualization of software repositories.

2.1 Software Repository Organization

A software repository, a place where reusable software components are stored and searched, is a valuable resource in reuse-based software development. A reusable component can be any software document or work product generated during the software development process, examples include requirement analysis documents, architectural designs, code modules, test plans, test cases, and documentation [10, 33]. It is crucial that the software repository should be well-structured such that the reusable components closest to the developers' needs are easy to discover [15].

A large amount of research effort on techniques for software repository organization that assist developers in locating appropriate components, has been suggested in the literature. Three of the best-known techniques are faceted classification, free-text indexing, and knowledge-based indexing. *Faceted classification systems* use a set of facets, each of which has several predefined terms. Software components are categorized by synthesizing the facet terms selected from faceted lists [29]. This technique was implemented in REBOOT [34]. *Free-text indexing systems* automatically extract keywords from natural language documentation (such as manual pages and code comments) by using classic information retrieval techniques, and use these keywords to recognize software components. GURU [20] and RSL [5] adopted this technique. *Knowledge-based indexing systems* perform syntactic and semantic analysis on the natural language specification, and also store semantic information about the application domain and the natural language itself in a knowledge base. LASSIE applied this technique [8].

More recently, many researchers have explored the use of machine learning and data mining techniques to improve the performance and obtain better retrieval results. Damiani and Fugini [6] introduced a fuzzy classification model which is flexible and useful when the information about the reusable components is ill-defined. Pedrycz and Waletzky [27] proposed a fuzzy clustering scheme, based on fuzzy techniques and SOM. Lee and his colleagues [19] applied genetic algorithms for optimization of multi-way clustering and retrieval. Ugurel and his colleagues [35] demonstrated a support vector machine (SVM) approach to classify the archived source code according to application topics and programming languages.

2.2 Existing Applications of SOM to Software Repositories

In the literature, a number of researchers have demonstrated the usefulness of the SOM methodology in constructing knowledge maps for visualizing software repositories. The resulting maps, serving as retrieval interfaces, help developers to visualize the structure of software repositories and increase their understanding of the relationships among software components in terms of their geographical closeness.

Merkl and his colleagues [21, 22] demonstrated the ability of SOM in organizing software repositories according to the semantic similarity or functional similarity of the stored software components. They used a small set of 36 MS-DOS commands and the NIH C++ class library as test cases. A binary single-term indexing scheme was used to

represent the software components. A similar approach was adopted by Ye and Lo [36], where a collection of 97 UNIX commands was used in their experiments. In addition, the quality of the library's software component indexing was improved by means of weighted single-term identification, phrase formation, and thesaurus construction. More recently, Pedrycz and his colleagues [26] utilized SOM in a new dimension, particularly to analyze software measure data such as lines of code, number of methods, depth of inheritance tree, and number of children. In their study, a sample of 643 JAVA classes was used as experimental data.

One of the main reasons for the limited success of all of these SOM applications is the use of the traditional SOM which may not be practical when the number of software components stored in a software repository is large. Intensive and iterative training on different sized networks is required in order to find the network architecture that provides satisfying results [1, 30]. Therefore, applying dynamic SOM models to software repository organization seems to be a more promising alternative.

3. SELF-ORGANIZING MAP

The Self-Organizing Map (SOM), first introduced by Kohonen in 1981, is one of the major unsupervised learning paradigms in the family of artificial neural networks [17]. The SOM network typically consists of an input layer and an output or competitive layer. The input layer is composed of a set of n -dimensional input vectors $x = [x_1, x_2, \dots, x_n]^T$, where n indicates the number of features that each input vector contains. The output layer is an m -dimensional (usually two-dimensional) grid consisting of a set of neurons, each associated with an n -dimensional weight vector $w_i = [w_{i1}, w_{i2}, \dots, w_{in}]^T$ (the same dimension as the input vector). The arrangement of the neurons can be rectangular or hexagonal. The architecture of a 4×5 SOM is shown in Fig. 1.

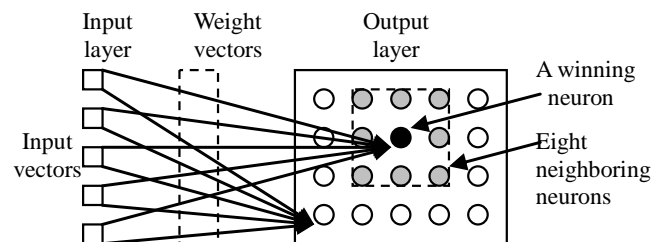


Fig. 1. The architecture of a 4×5 SOM [17].

Conceptually, SOM takes a set of inputs and maps them onto the neurons of a two-dimensional grid. The weight vectors are randomly initialized at the first stage. Then, the SOM network performs learning in two main steps as described below.

1) Determining the winning neuron

The SOM network determines the winning neuron for a given input vector, selected randomly from the set of all input vectors. For every neuron on the grid, its weight vector

is compared with the input vector by using some similarity measures, e.g., Euclidean distance. The neuron whose weight vector is closest to the input vector is selected to be the winning neuron. Eq. (1) shows how to determine the winning neuron c .

$$c : \|x - w_c\| = \min_i \|x - w_i\|. \quad (1)$$

2) Adjusting weights

After a winning neuron is determined, the weight vectors of the winning neuron and all of its neighboring neurons are adjusted by moving toward the input vectors according to the learning rule, as given in Eq. (2)

$$w_i(t+1) = w_i(t) + h_{ci}(t)[x(t) - w_i(t)], \quad (2)$$

where t is a discrete time constant denoting the current learning iteration. The neighborhood function $h_{ci}(t)$ is used to determine to which extent the neighboring neurons, lying within a certain radius of the winning neuron, will be updated. This function is a time decreasing function that converges to zero for large values of t . A typical smooth Gaussian neighborhood function is given in Eq. (3) below

$$h_{ci}(t) = \alpha(t) \exp(-\|r_c - r_i\|^2 / 2\sigma(t)^2), \quad (3)$$

where $\alpha(t)$ is the learning rate function which controls the amount of weight vector movement and gradually decreases over time, $\sigma(t)$ is the width of the Gaussian kernel, and $\|r_c - r_i\|^2$ is the distance between the winning neuron c and neuron i .

This learning process progresses repeatedly until it converges to a stable state where there are no further changes made to the weight vectors when they are presented with the given input vectors. After the training has been completed, an orderly map is formed in such a way that the topology of the data is preserved and becomes geographically explicit in that similar input data are mapped onto nearby regions of the map [7, 17].

4. GROWING HIERARCHICAL SELF-ORGANIZING MAP

The Growing Hierarchical Self-Organizing Map (GHSOM) [9, 30], which is an extension to the growing grid SOM [12] and hierarchical SOM [23], can build a hierarchy of multiple layers where each layer consists of several independent growing SOMs. The size of these SOMs and the depth of the hierarchy are determined during its learning process according to the requirements of the input data. As depicted in Fig. 2, the GHSOM architecture is similar to a tree structure where the SOM(s) at each layer can branch out to additional SOMs at the subsequent layer. The upper layers show a coarse organization of the major clusters in the data, whereas the lower layers offer a more detailed view of the data.

For the initial setup of the GHSOM, at Layer 0, a single-neuron SOM is created and the neuron's weight vector is initialized as the average of all input vectors. Then, the learning process starts at Layer 1 with a small SOM (usually a 2×2 grid) whose weight vectors are initialized to random values.

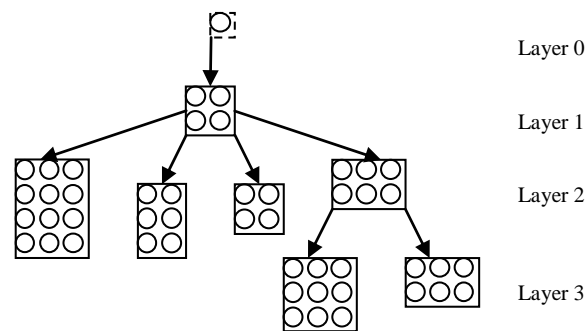


Fig. 2. The GHSOM architecture [30].

The GHSOM grows in two dimensions: horizontally (by increasing the size of each SOM) and hierarchically (by increasing the number of layers). For horizontal growth, each SOM modifies itself in a systematic way very similar to the growing grid [12] so that each neuron does not represent too large an input space. For hierarchical growth, the principle is to periodically check whether the lowest layer SOMs have achieved sufficient coverage for the underlying input data. The basic steps of the horizontal growth and the hierarchical growth of the GHSOM are summarized in Table 1 below.

Table 1. Basic steps of the horizontal growth and the hierarchical growth of the GHSOM.

| |
|--|
| <p>Basic steps of horizontal growth:</p> <ol style="list-style-type: none"> 1. Initialize the weight vector of each neuron with random values. 2. Perform the traditional SOM learning algorithm for a fixed number λ of times. 3. Find the error unit e and its most dissimilar neighbor unit d. (Note that the error unit e is the neuron with the largest deviation between its weight vector and the input vectors it represents.) 4. Insert a new row or a new column between e and d. The weight vectors of these new neurons are initialized as the average of their neighbors. 5. Repeat steps 2-4 until the mean quantization error of the map $MQE_m < \tau_1 * qe_u$ where qe_u is the quantization error of the neuron u in the preceding layer of the hierarchy. <p>Basic steps of hierarchical growth:</p> <ol style="list-style-type: none"> 1. Check each neuron to find out if its $qe_i > \tau_2 * qe_0$, where qe_0 is the quantization error of the single neuron of Layer 0, then assign a new SOM at a subsequent layer of the hierarchy. 2. Train the SOM with input vectors mapped to this neuron. |
|--|

The growth process of the GHSOM is controlled by the following four important factors.

- The quantization error of a neuron i , qe_i , is calculated as the sum of the distance between the weight vector of neuron i and the input vectors mapped onto this neuron.

- The mean quantization error of the map (MQE_m) is the mean of all neurons' quantization errors in the map.
- The threshold τ_1 is for specifying the desired level of detail that is to be shown in a particular SOM.
- The threshold τ_2 is for specifying the desired quality of input data representation at the end of the learning process.

5. EXPERIMENTS

This section provides a brief description of the data set used in the experiments. Also, the procedure of the experiments is given. Lastly, the results of the application of the SOM and the GHSOM to the structured software repository are compared.

5.1 Data Set

The data set used in this study consisted of 273 samples of C/C++ program source code files gathered from three well-known textbooks that are widely used as major references in computer science classes. They are 1) Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss¹; 2) Information Retrieval Data Structures & Algorithms by Bill Frakes²; and 3) Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig³. 110 files are from the first source, 47 files are from the second source, and 116 files are from the third source. The lengths of the sample programs in the collection range from a few to several hundred lines of code.

5.2 Procedure

The procedure of the experiments was as follows. First, a set of software components for potential reuse (i.e., C/C++ program source code files) was gathered and saved as text files. Then preprocessing, including stop words elimination and stemming, was done on the collection in order to obtain high-quality features for describing the software components. Stop words (frequently-used words that don't help distinguish one component from the other such as "a," "an," "the," "is," "are," etc.) were eliminated. Words having the same common linguistic roots were grouped together according to the Porter Stemming algorithm [28]. For example, adjustable, adjustment, and adjust are grouped into adjust. After that, meaningful keywords were extracted from the software components by using a single-term free-text indexing scheme. During indexing, keywords occurring in fewer than 5 software components or in more than 218 software components were omitted. There were 542 keywords remaining for software component representation. By taking into account the importance of each keyword, these keywords are further weighted according to the term frequency multiplied by inverse document frequency ($tf \times idf$) weighting scheme. Each software component is then represented as a feature vector

¹ http://www.cs.fiu.edu/~weiss/dsaa_c++/code/.

² <http://www.dcc.uchile.cl/~rbaeza/iradsbook/irbook.html>.

³ <http://www.cs.berkeley.edu/~russell/aima.html>.

in the Vector Space Model (VSM), where the $tf \times idf$ value of each keyword in each software component is recorded in a components-versus-keywords matrix [32]. Next, all of the feature vectors were used as input vectors for the construction of the SOM and the GHSOM. Finally, the resulting maps were used to visualize the structure of the software repository and the semantic relationships among the stored software components.

It is important to note that we used the feature extraction programs of the SOMLib⁴ Java package to extract keywords and create the feature vectors. Also, we utilized the MATLAB SOM Toolbox⁵ and GHSOM Toolbox⁶ for the construction of the SOM and the GHSOM.

5.3 Results

In order to compare the results of the application of the SOM and the GHSOM, two experiments were conducted.

In the first experiment, a 10×10 SOM was produced, as depicted in Fig. 3. By inspecting Fig. 3 visually, it is noticeable that there are three main groups of software components which are Data Structure (DS), Information Retrieval (IR), and Artificial Intelligence (AI), as labeled with cluster titles. Software components with similar

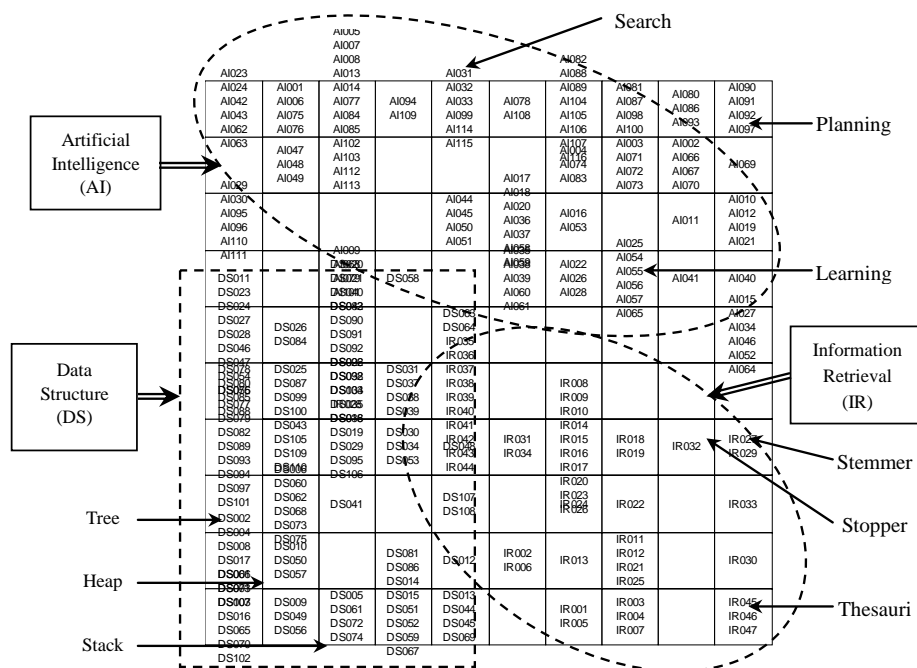


Fig. 3. The resulting 10×10 SOM.

⁴ <http://www.ifs.tuwien.ac.at/~andi/somlib/>.

⁵ <http://www.cis.hut.fi/projects/somtoolbox/>.

⁶ <http://www.ifs.tuwien.ac.at/~andi/ghsom/>.

features are apparently located on nearby regions of the map. For example in IR, a cluster of Stemmer programs can be found at the right side of the map, and next to it is a cluster of Stopper programs. As another example, a cluster of Thesauri programs shows up at the bottom right corner of the map.

In the second experiment, by setting the thresholds $\tau_1 = 0.8500$ and $\tau_2 = 0.0035$, a 4-layer GHSOM was generated, as illustrated in Fig. 4. It can be seen that the clusters are the areas with high data densities on the map that are further hierarchically expanded by growing SOMs. In the figure, the top layer maps are depicted in gray and the bottom layer maps are depicted in white. The first layer map, consisting of 3×3 neurons, shows the three major clusters of software components: DS, IR, and AI. Most neurons of the first layer SOM have been expanded in the second layer maps. Due to space limitation, only two of its sub maps are presented in Fig. 5. Fig. 5 (a) shows an AI sub map, consisting of 2×4 neurons, expanded from the neuron at the upper left corner. Fig. 5 (b) shows a DS sub map, consisting of 2×2 neurons, expanded from the neuron at the lower right corner. Let's have a look at Fig. 5 (b). Programs related to Tree structure, e.g., AVL Tree, Binary Search Tree, and Splay Tree are grouped together at the lower left corner of the map, and programs related to Heap structure, e.g., Leftist Heap, Pairing Heap, and Treap are grouped separately in the same vicinity.

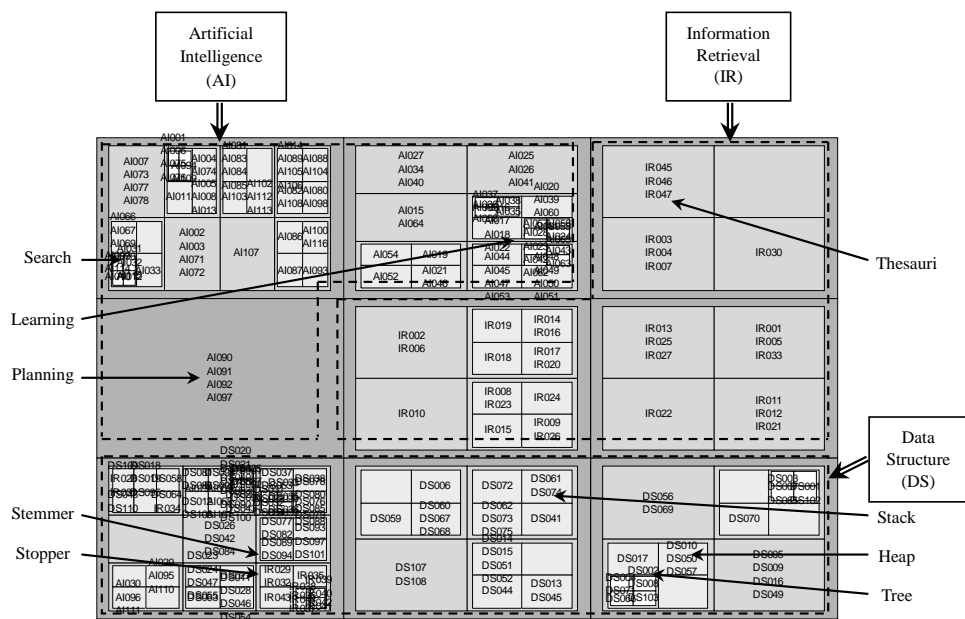


Fig. 4. The resulting 4-layer GHSOM.

According to the experimental results, we found that both SOM and GHSOM were successful in creating a topology-preserving representation of the topical clusters of the software components. However, when dealing with a large number of software components, GHSOM behaved better than SOM in the sense that its architecture was

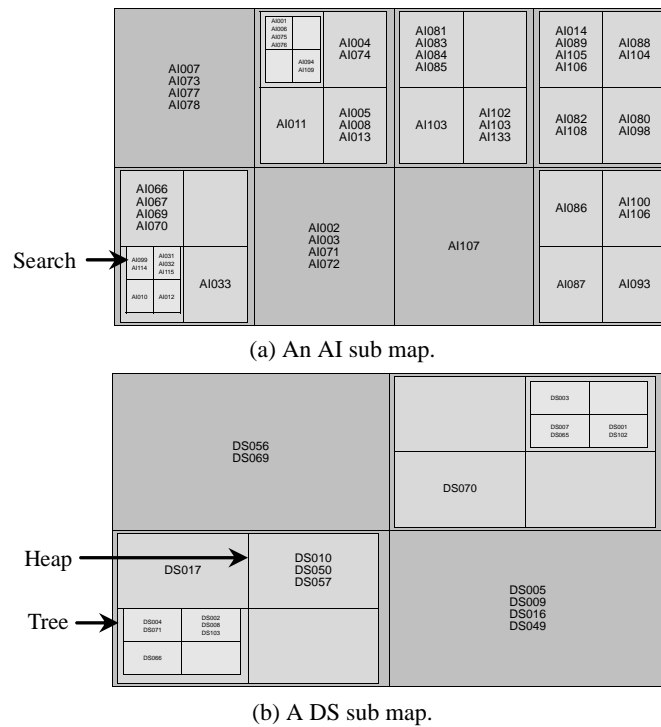


Fig. 5. Two sub maps of the resulting 4-layer GHSOM.

determined automatically during its learning process based on the requirement of the input data. Moreover, GHSOM was able to reveal the inherent hierarchical structure of the data into layers and provided the ability to select the granularity of the representation at different levels of the GHSOM.

6. CONCLUSIONS

In this paper, we demonstrated the potential of the GHSOM for organization and visualization of a software repository and compared its results with the ones obtained by the traditional SOM. GHSOM was introduced as an alternative technique for clustering a collection of software components and generating knowledge maps which can be helpful to developers in obtaining a better insight into the structure of a software repository and increasing their understanding of the semantic relationships among software components. By using the resulting maps, developers can find the needed software components more easily and quickly, and make better decisions in selecting the most suitable software components to reuse. The results of the experiments conducted in this study lead us to the conclusion that the GHSOM is more promising than the traditional SOM owing to its adaptive architecture and the ability to expose the hierarchical structure of data. For future work, we will explore the use of GHSOM combined with other machine learning or data mining techniques such as fuzzy sets theory, genetic algorithms, and mining asso-

ciation rules in an attempt to improve the performance and obtain better results. Furthermore, we will conduct experiments based on larger collections of software components.

REFERENCES

1. D. Alahakoon, S. K. Halgamuge, and B. Srinivasan, "Dynamic self-organizing maps with controlled growth for knowledge discovery," *IEEE Transactions on Neural Networks*, Vol. 11, 2000, pp. 601-614.
2. L. B. Batista, H. M. Gomes, and R. F. Herbster, "Application of growing hierarchical self-organizing map in handwritten digit recognition," in *Proceedings of 16th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, 2003, pp. 1539-1545.
3. H. Bauer and T. Villmann, "Growing a hypercubical output space in a self-organizing feature map," *IEEE Transactions on Neural Networks*, Vol. 8, 1997, pp. 218-226.
4. J. Blackmore and R. Miikkulainen, "Incremental grid growing: encoding high-dimensional structure into a two-dimensional feature map," in *Proceedings of the IEEE International Conference on Neural Networks*, Vol. 1, 1993, pp. 450-455.
5. B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler, and L. A. Mayes, "The reusable software library," *IEEE Software*, Vol. 4, 1987, pp. 25-33.
6. E. Damiani and M. G. Fugini, "Design and code reuse based on fuzzy classification of components," *ACM SIGAPP Applied Computing Review*, Vol. 4, 1996, pp. 26-32.
7. G. Deboeck and T. Kohonen, *Visual Explorations in Finance with Self-Organizing Maps*, Springer, New York, 1998.
8. P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "LaSSIE: a knowledge-based software information system," *Communications of the ACM*, Vol. 34, 1991, pp. 34-49.
9. M. Dittenbach, D. Merkl, and A. Rauber, "The growing hierarchical self-organizing map," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, Vol. 6, 2000, pp. 15-19.
10. P. Freeman, "Reusable software engineering: concepts and research directions," *Tutorial: Software Reusability*, 1987, pp. 10-23.
11. B. Fritzke, "Growing cell structures – a self-organizing network for unsupervised and supervised learning," *Neural Networks*, Vol. 7, 1994, pp. 1441-1460.
12. B. Fritzke, "Growing grid – a self-organizing network with constant neighborhood range and adaptation strength," *Neural Processing Letters*, Vol. 2, 1995, pp. 9-13.
13. W. B. Frakes and C. J. Fox, "Sixteen questions about software reuse," *Communications of the ACM*, Vol. 38, 1995, pp. 75-87.
14. 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*, 2000, pp. 92-100.
15. S. Henninger, "An evolutionary approach to constructing effective software reuse repositories," *ACM Transactions on Software Engineering and Methodology*, Vol. 6, 1997, pp. 111-140.

16. T. Kohonen, S. Kaski, K. Lagus, J. Salojärvi, J. Honkela, V. Paatero, and A. Saarela, "Self-organization of a massive document collection," *IEEE Transactions on Neural Networks*, Vol. 11, 2000, pp. 574-585.
17. T. Kohonen, *Self-Organizing Maps*, 3rd ed., Springer, New York, 2001.
18. C. W. Krueger, "Software reuse," *ACM Computing Surveys*, Vol. 24, 1992, pp. 131-183.
19. B. Lee, B. Moon, and C. Wu, "Optimization of multi-way clustering and retrieval using genetic algorithms in reusable class library," in *Proceedings of the Asia Pacific Software Engineering Conference*, 1998, pp. 4-11.
20. Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp. 800-813.
21. D. Merkl, A. M. Tjoa, and G. Kappel, "Learning the semantic similarity of reusable software components," in *Proceedings of 3rd International Conference on Software Reuse: Advances in Software Reusability*, 1994, pp. 33-41.
22. D. Merkl, "Self-organizing maps and software reuse," *Computational Intelligence in Software Engineering*, W. Pedrycz and J. F. Peters (eds.), 1998.
23. R. Miikkulainen, "Script recognition with hierarchical feature maps," *Connection Science*, Vol. 2, 1990, pp. 83-101.
24. A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, Vol. 5, 1998, pp. 349-414.
25. T. Naenna, R. A. Bress, and M. J. Embrechts, "DNA classifications with self-organizing maps (SOMs)," in *Proceedings of the IEEE International Workshop on Soft Computing in Industrial Applications*, 2003, pp. 151-154.
26. W. Pedrycz, G. Succi, M. Reformat, P. Musilek, and X. Bai, "Self organizing maps as a tool for software analysis," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, 2001, pp. 93-97.
27. W. Pedrycz and J. Waletzky, "Fuzzy clustering in software reusability," *Software-Practice and Experience*, Vol. 27, 1997, pp. 245-270.
28. M. F. Porter, "An algorithm for suffix stripping," *Program*, Vol. 14, 1980, pp. 130-137.
29. R. Prieto-Diaz, "Implementing faceted classification for software reuse," *Communications of the ACM*, Vol. 34, 1991, pp. 88-97.
30. A. Rauber, D. Merkl, and M. Dittenbach, "The growing hierarchical self-organizing map: exploratory analysis of high-dimensional data," *IEEE Transactions on Neural Networks*, Vol. 13, 2002, pp. 1331-1341.
31. I. S. Reljin, B. D. Reljin, and G. Jovanovi, "Clustering of climate data in yugoslavia by using the SOM neural network," in *Proceedings of 6th Seminar on Neural Network Applications in Electrical Engineering*, 2002, pp. 203-206.
32. G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley Publishing Company, Reading, MA, 1989.
33. M. H. Samadzadeh and M. K. Zand, "Software houses," *Encyclopedia of Electrical and Electronics Engineering*, J. G. Webster (eds.), John Wiley and Sons, Inc., New York, Vol. 19, 1999, pp. 473-483.
34. G. Sindre, R. Conradi, and E. Karlsson, "The REBOOT approach to software reuse,"

Journal of Systems Software, Vol. 30, 1995, pp. 201-212.

35. S. Ugurel, R. Krovetz, C. L. Giles, D. M. Pennock, E. J. Glover, and H. Zha, "What's the code? Automatic classification of source code archives," in *Proceedings of 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002, pp. 632-638.
36. H. Ye and B. W. N. Lo, "Towards a self-structuring software library," *IEE Proceedings - Software*, Vol. 148, 2001, pp. 45-55.



Songsri Tangsripairoj is a lecturer at the Department of Computer Science, faculty of Science, Mahidol University, Bangkok, Thailand. She received the B.S., M.S., and Ph.D. degrees in Computer Science from Thammasat University, Bangkok, Thailand in 1994, Mahidol University, Bangkok, Thailand in 1996, and Oklahoma State University in 2004, respectively. Her research interests include software engineering (software reuse), data warehousing, and data mining.



M. H. Samadzadeh is a Professor of Computer Science at the Computer Science Department of Oklahoma State University, Stillwater, Oklahoma, U.S.A. His research interests include software engineering (software metrics, reuse, debugging, program comprehension), operating systems (multiprocessor scheduling, synchronization, memory management), parallel computing, formal languages, and automata theory. He is a member of the ACM and IEEE-CS.